Ce livre est dédié à
Robert Jourdain,
John Socha,
Ralf Brown
et Peter Abel

Reverse Engineering pour Débutants

(Comprendre le langage d'assemblage)

Dennis Yurichev

Reverse Engineering pour Débutants

(Comprendre le langage d'assemblage)

Pourquoi deus titres? Lire ici: on page x.

Dennis Yurichev
<dennis@yurichev.com>

Version du texte (12 août 2018).

La dernière version (et édition en russe) de ce texte est accessible sur beginners.re.

La couverture a été réalisée par Andy Nechaevsky: facebook.

A la recherche de traducteurs !

Vous souhaitez peut-être m'aider en traduisant ce projet dans d'autres langues, autres que l'anglais et le russe. Il vous suffit de m'envoyer les portions de texte que vous avez traduites (peu importe leur longueur) et je les intégrerai à mon code source LaTeX.

Lire ici.

Nous avons déjà quelque chose en allemand, français, un peu en italien, portugais et polonais.

La vitesse de traduction n'est pas importante, puisqu'il s'agit d'un projet open-source après tout. Votre nom sera mentionné en tant que contributeur au projet. Les traductions en coréen, chinois et persan sont réservées par mes éditeurs.

Les versions anglaise et russe ont été réalisées par moi-même. Toutefois, mon anglais est toujours horrible et je vous serais très reconnaissant pour toute éventuelle remarque sur la grammaire, etc... Même mon russe est imparfait, donc je vous serais également reconnaissant pour toute remarque sur la traduction en russe !

N'hésitez donc pas à me contacter : `dennis@yurichev.com`.

Contenus abrégés

Table des matières

[1] Floating-Point Unit

[2]Redundant Array of Independent Disks

Préface

C'est quoi ces deux titres?

Le livre a été appelé "Reverse Engineering for Beginners" en 2014-2018, mais j'ai toujours suspecté que ça rendait son audience trop réduite.

Les gens de l'infosec connaissent le "reverse engineering", mais j'ai rarement entendu le mot "assembleur" de leur part.

De même, le terme "reverse engineering" est quelque peu cryptique pour une audience générale de programmeurs, mais qui ont des connaissances à propos de l'"assembleur".

En juillet 2018, à titre d'expérience, j'ai changé le titre en "Assembly Language for Beginners" et publié le lien sur le site Hacker News[3], et le livre a été plutôt bien accueilli.

Donc, c'est ainsi que le livre a maintenant deux titres.

Toutefois, j'ai changé le second titre à "Understanding Assembly Language", car quelqu'un a déjà écrit le livre "Assembly Language for Beginners". De même, des gens disent que "for Beginners" sonne sarcastique pour un livre de ~1000 pages.

Les deux livres diffèrent seulement par le titre, le nom du fichier (UAL-XX.pdf versus RE4B-XX.pdf), l'URL et quelques-une des première pages.

À propos de la rétro-ingénierie

Il existe plusieurs définitions pour l'expression « ingénierie inverse ou rétro-ingénierie reverse engineering » :

1) L'ingénierie inverse de logiciels : examiner des programmes compilés;

2) Le balayage des structures en 3D et la manipulation numérique nécessaire afin de les reproduire;

3) Recréer une structure de base de données.

Ce livre concerne la première définition.

Prérequis

Connaissance basique du C LP[4]. Il est recommandé de lire: 8.1.3 on page 648.

Exercices et tâches

...ont été déplacés sur un site différent : http://challenges.re.

[3] https://news.ycombinator.com/item?id=17549050
[4] Langage de programmation

A propos de l'auteur

Dennis Yurichev est un ingénieur expérimenté en rétro-ingénierie et un programmeur. Il peut être contacté par email : **dennis@yurichev.com**.

Éloges de ce livre

- « Now that Dennis Yurichev has made this book free (libre), it is a contribution to the world of free knowledge and free education. » Richard M. Stallman, Fondateur de GNU, militant pour la liberté des logiciels
- « It's very well done .. and for free .. amazing. »[5] Daniel Bilar, Siege Technologies, LLC.
- « ... excellent and free »[6] Pete Finnigan, gourou de la sécurité Oracle RDBMS.
- « ... [the] book is interesting, great job! » Michael Sikorski, auteur de *Practical Malware Analysis: The Hands-On Guide to Dissecting Malicious Software*.
- « ... my compliments for the very nice tutorial! » Herbert Bos, professeur à temps complet à Vrije Universiteit Amsterdam, co-auteur de *Modern Operating Systems (4th Edition)*.
- « ... It is amazing and unbelievable. » Luis Rocha, CISSP / ISSAP, Technical Manager, Network & Information Security at Verizon Business.
- « Thanks for the great work and your book. » Joris van de Vis, spécialiste SAP Netweaver & Security .
- « ... [a] reasonable intro to some of the techniques. »[7] Mike Stay, professeur au Federal Law Enforcement Training Center, Georgia, US.
- « I love this book! I have several students reading it at the moment, [and] plan to use it in graduate course. »[8] Sergey Bratus, Research Assistant Professor dans le Département Informatique du Dartmouth College
- « Dennis @Yurichev has published an impressive (and free!) book on reverse engineering »[9] Tanel Poder, expert en optimisation des performances Oracle RDBMS .
- « This book is a kind of Wikipedia to beginners... » Archer, Chinese Translator, IT Security Researcher.
- « [A] first-class reference for people wanting to learn reverse engineering. And it's free for all. » Mikko Hyppönen, F-Secure.

Remerciements

Pour avoir patiemment répondu à toutes mes questions : Andrey « herm1t » Baranovich, Slava « Avid » Kazakov, SkullC0DEr.

Pour m'avoir fait des remarques par rapport à mes erreurs ou manques de précision : Stanislav « Beaver » Bobrytskyy, Alexander Lysenko, Alexander « Solar Designer » Peslyak, Federico Ramondino, Mark Wilson, Xenia Galinskaya, Razikhova Meiramgul Kayratovna, Anatoly Prokofiev, Kostya Begunets, Valentin "netch"

[5] twitter.com/daniel_bilar/status/436578617221742593
[6] twitter.com/petefinnigan/status/400551705797869568
[7] reddit
[8] twitter.com/sergeybratus/status/505590326560833536
[9] twitter.com/TanelPoder/status/524668104065159169

Nechayev, Aleksandr Plakhov, Artem Metla, Alexander Yastrebov, Vlad Golovkin[10], Shell Rocket, Zhu Ruijin, Changmin Heo, Vitor Vidal, Stijn Crevits, Jean-Gregoire Foulon[11], Ben L., Etienne Khan, Norbert Szetei[12], Marc Remy, Michael Hansen, Derk Barten, The Renaissance[13], Hugo Chan, Emil Mursalimov..

Pour m'avoir aidé de toute autre manière : Andrew Zubinski, Arnaud Patard (rtp on #debian-arm IRC), noshadow on #gcc IRC, Aliaksandr Autayeu, Mohsen Mostafa Jokar.

Pour avoir traduit le livre en chinois simplifié : Antiy Labs (antiy.cn), Archer.

Pour avoir traduit le livre en coréen : Byungho Min.

Pour avoir traduit le livre en néerlandais : Cedric Sambre (AKA Midas).

Pour avoir traduit le livre en espagnol : Diego Boy, Luis Alberto Espinosa Calvo, Fernando Guida, Diogo Mussi, Patricio Galdames.

Pour avoir traduit le livre en portugais : Thales Stevan de A. Gois, Diogo Mussi.

Pour avoir traduit le livre en italien : Federico Ramondino[14], Paolo Stivanin[15], twyK.

Pour avoir traduit le livre en français : Florent Besnard[16], Marc Remy[17], Baudouin Landais, Téo Dacquet[18], BlueSkeye@GitHub[19].

Pour avoir traduit le livre en allemand : Dennis Siekmeier[20], Julius Angres[21], Dirk Loser[22], Clemens Tamme.

Pour avoir traduit le livre en polonais: Kateryna Rozanova, Aleksander Mistewicz, Wiktoria Lewicka.

Pour avoir traduit le livre en japonais: shmz@github[23].

Pour la relecture : Alexander « Lstar » Chernenkiy, Vladimir Botov, Andrei Brazhuk, Mark "Logxen" Cooper, Yuan Jochen Kang, Mal Malakov, Lewis Porter, Jarle Thorsen, Hong Xie.

Vasil Kolev[24] a réalisé un gros travail de relecture et a corrigé beaucoup d'erreurs.

Pour les illustrations et la couverture : Andy Nechaevsky.

Merci également à toutes les personnes sur github.com qui ont contribué aux remarques et aux corrections[25].

De nombreux packages LaTeX ont été utilisé : j'aimerais également remercier leurs auteurs.

Donateurs

Ceux qui m'ont sontenu lorsque j'écrivais le livre :

2 * Oleg Vygovsky (50+100 UAH), Daniel Bilar ($50), James Truscott ($4.5), Luis Rocha ($63), Joris van de Vis ($127), Richard S Shultz ($20), Jang Minchang ($20), Shade Atlas (5 AUD), Yao Xiao ($10), Pawel Szczur (40 CHF), Justin Simms ($20), Shawn the R0ck ($27), Ki Chan Ahn ($50), Triop AB (100 SEK), Ange Albertini (€10+50), Sergey Lukianov (300 RUR), Ludvig Gislason (200 SEK), Gérard Labadie (€40), Sergey Volchkov (10 AUD), Vankayala Vigneswararao ($50), Philippe Teuwen ($4), Martin Haeberli ($10), Victor Cazacov (€5), Tobias Sturzenegger (10 CHF), Sonny Thai ($15), Bayna AlZaabi ($75), Redfive B.V. (€25), Joona Oskari Heikkilä (€5), Marshall Bishop ($50), Nicolas Werner (€12), Jeremy Brown ($100), Alexandre Borges ($25), Vladimir Dikovski (€50), Jiarui Hong (100.00 SEK), Jim Di (500 RUR), Tan Vincent ($30), Sri Harsha Kandrakota (10 AUD), Pillay Harish (10 SGD), Timur Valiev (230 RUR), Carlos Garcia Prado (€10), Salikov Alexander (500 RUR), Oliver Whitehouse (30 GBP), Katy Moe ($14), Maxim Dyakonov ($3), Sebastian Aguilera (€20), Hans-Martin Münch (€15), Jarle Thorsen (100 NOK), Vitaly Osipov ($100), Yuri Romanov

[10] goto-vlad@github
[11] https://github.com/pixjuan
[12] https://github.com/73696e65
[13] https://github.com/TheRenaissance
[14] https://github.com/pinkrab
[15] https://github.com/paolostivanin
[16] https://github.com/besnardf
[17] https://github.com/mremy
[18] https://github.com/T30rix
[19] https://github.com/BlueSkeye
[20] https://github.com/DSiekmeier
[21] https://github.com/JAngres
[22] https://github.com/PolymathMonkey
[23] https://github.com/shmz
[24] https://vasil.ludost.net/
[25] https://github.com/DennisYurichev/RE-for-beginners/graphs/contributors

(1000 RUR), Aliaksandr Autayeu (€10), Tudor Azoitei ($40), Z0vsky (€10), Yu Dai ($10), Anonymous ($15), Vladislav Chelnokov ($25), Nenad Noveljic ($50), Ryan Smith ($25), Andreas Schommer (€5).

Un énorme merci à chaque donateur !

mini-FAQ

Q: Quels sont les prérequis nécessaires avant de lire ce livre ?

R: Une compréhension de base du C/C++ serait l'idéal.

Q: Dois-je apprendre x86/x64/ARM et MIPS en même temps ? N'est-ce pas un peu trop ?

R: Je pense que les débutants peuvent seulement lire les parties x86/x64, tout en passant/feuilletant celles ARM/MIPS.

Q: Puis-je acheter une version papier du livre en russe / anglais ?

R: Malheureusement non, aucune maison d'édition n'a été intéressée pour publier une version en russe ou en anglais du livre jusqu'à présent. Cependant, vous pouvez demander à votre imprimerie préférée de l'imprimer et de le relier.

Q: Y a-il une version ePub/Mobi ?

R: Le livre dépend majoritairement de TeX/LaTeX, il n'est donc pas évident de le convertir en version ePub/Mobi.

Q: Pourquoi devrait-on apprendre l'assembleur de nos jours ?

R: A moins d'être un développeur d'OS[26], vous n'aurez probablement pas besoin d'écrire en assembleur—les derniers compilateurs (ceux de notre décennie) sont meilleurs que les êtres humains en terme d'optimisation. [27].

De plus, les derniers CPU[28]s sont des appareils complexes et la connaissance de l'assembleur n'aide pas vraiment à comprendre leurs mécanismes internes.

Cela dit, il existe au moins deux domaines dans lesquels une bonne connaissance de l'assembleur peut être utile : Tout d'abord, pour de la recherche en sécurité ou sur des malwares. C'est également un bon moyen de comprendre un code compilé lorsqu'on le debug. Ce livre est donc destiné à ceux qui veulent comprendre l'assembleur plutôt que d'écrire en assembleur, ce qui explique pourquoi il y a de nombreux exemples de résultats issus de compilateurs dans ce livre.

Q: J'ai cliqué sur un lien dans le document PDF, comment puis-je retourner en arrière ?

R: Dans Adobe Acrobat Reader, appuyez sur Alt + Flèche gauche. Dans Evince, appuyez sur le bouton "<".

Q: Puis-je imprimer ce livre / l'utiliser pour de l'enseignement ?

R: Bien sûr ! C'est la raison pour laquelle le livre est sous licence Creative Commons (CC BY-SA 4.0).

Q: Pourquoi ce livre est-il gratuit ? Vous avez fait du bon boulot. C'est suspect, comme nombre de choses gratuites.

R: D'après ma propre expérience, les auteurs d'ouvrages techniques font cela pour l'auto-publicité. Il n'est pas possible de se faire beaucoup d'argent d'une telle manière.

Q: Comment trouver du travail dans le domaine de la rétro-ingénierie ?

R: Il existe des topics d'embauche qui apparaissent de temps en temps sur Reddit, dédiés à la rétro-ingénierie (cf. reverse engineering ou RE)[29] (2016). Jetez un oeil ici.

Un topic d'embauche quelque peu lié peut être trouvé dans le subreddit « netsec » : 2016.

Q: Comment puis-je apprendre à programmer en général?

R: Si vous pouvez maîtriser à la fois le language C et LISP, votre vie de programmeur en sera beaucoup plus facile. Je recommanderais de résoudre les exercices de [Brian W. Kernighan, Dennis M. Ritchie, *The C Programming Language*, 2ed, (1988)] et SICP[30].

[26] Système d'exploitation (Operating System)
[27] Un très bon article à ce sujet : [Agner Fog, *The microarchitecture of Intel, AMD and VIA CPUs*, (2016)]
[28] Central Processing Unit
[29] reddit.com/r/ReverseEngineering/
[30] Structure and Interpretation of Computer Programs

Q: J'ai une question...

R: Envoyez la moi par email (dennis@yurichev.com).

Comment apprendre à programmer

Beaucoup de gens me l'ont demandé.

Il n'y a pas de "voie royale", mais il y a quelques chemins efficaces.

De ma propre expérience, ceci est juste: résoudre les exercices de:

- Brian W. Kernighan, Dennis M. Ritchie, *The C Programming Language*, 2ed, (1988)
- Harold Abelson, Gerald Jay Sussman, Julie Sussman – Structure and Interpretation of Computer Programs
- Donald E. Knuth, *The Art of Computer Programming*
- Niklaus Wirth's books
- Brian W. Kernighan, Rob Pike, *Practice of Programming*, (1999)

... en pur C et LISP. Vous pourriez ne jamais utiliser du tout ces langages de programmation dans le futur. Presqu'aucun des programmeurs de métier le les utilisent. Mais l'expérience du développement en C et en LISP aide énormément sur la long terme.

Vous pouvez également passer leur lecture elle-même, parcourez-les seulement lorsque vous sentez que vous devez comprendre quelque chose que vous ne comprenez pas dans l'exercice que vous faites.

Ceci peu prendre des années, ou une vie entière, mais c'est toujours plus rapide que de se précipiter entre les modes.

Le succès de ces livres est sans doute lié au fait que les auteurs sont des enseignants et tous ce matériel a d'abord été amélioré avec les étudiants.

Pour LISP, je recommande personnellement Racket (dialecte Scheme). Mais c'est une affaire de goût.

Certaines personnes disent que comprendre le langage d'assemblage est très utile, même si vous ne l'utiliserez jamais. Ceci est vrai. Mais ceci est une voie pour les geeks les plus persévérants, et ça peut être mis de côté au début.

De même, les autodidactes (incluant l'auteur de ces lignes) ont souvent le problème de travailler dur sur des problèmes difficiles, passant sur les plus faciles. Ceci est une grosse erreur. Comparez avec le sport ou le musique – personne ne commence à un poids de 100kg, ou Paganini's Caprices. Je dirais – essayez de résoudre un problème si vous pouvez en esquisser la solution dans votre tête.

> I think the art of doing research consists largely of asking questions, and sometimes answering them. Learn how to repeatedly pose miniquestions that represent special cases of the big questions you are hoping to solve.
>
> When you begin to explore some area, you take baby steps at first, building intuition about that territory. Play with many small examples, trying to get a complete understanding of particular parts of the general situation.
>
> In that way you learn many properties that are true and many properties that are false. That gives guidance about directions that are fruitful versus directions to avoid.
>
> Eventually your brain will have learned how to take larger and larger steps. And shazam, you'll be ready to take some giant steps and solve the big problem.
>
> But don't stop there! At this point you'll be one of very few people in the world who have ever understood your problem area so well. It will therefore be your responsibility to discover what else is true, in the neighborhood of that problem, using the same or similar methods to what your brain can now envision. Take your results to their "natural boundary" (in a sense analogous to the natural boundary where a function of a complex variable ceases to be analytic).
>
> My little book Surreal Numbers provides an authentic example of research as it is happening. The characters in that story make false starts and useful discoveries in exactly the same order as I myself made those false starts and useful discoveries, when I first studied John Conway's fascinating axioms about number systems — his amazingly simple axioms that go significantly beyond real-valued numbers.

> (One of the characters in that book tends to succeed or fail by brute force and patience; the other is more introspective, and able to see a bigger picture. Both of them represent aspects of my own activities while doing research. With that book I hoped to teach research skills “by osmosis”, as readers observe a detailed case study.)
>
> Surreal Numbers deals with a purely mathematical topic, not especially close to computer science; it features algebra and logic, not algorithms. When algorithms become part of the research, a beautiful new dimension also comes into play: Algorithms can be implemented on computers!
>
> I strongly recommend that you look for every opportunity to write programs that carry out all or a part of whatever algorithms relate to your research. In my experience the very act of writing such a program never fails to deepen my understanding of the problem area.

(Donald E. Knuth – https://theorydish.blog/2018/02/01/donald-knuth-on-doing-research/)

Bonne change!

À propos de la traduction en Coréen

En Janvier 2015, la maison d’édition Acorn (www.acornpub.co.kr) en Corée du Sud a réalisé un énorme travail en traduisant et en publiant mon livre (dans son état en Août 2014) en Coréen.

Il est désormais disponible sur leur site web.

Le traducteur est Byungho Min (twitter/tais9). L’illustration de couverture a été réalisée l’artiste, Andy Nechaevsky, un ami de l’auteur: facebook/andydinka. Ils détiennent également les droits d’auteurs sur la traduction coréenne.

Donc si vous souhaitez avoir un livre *réel* en coréen sur votre étagère et que vous souhaitez soutenir ce travail, il est désormais disponible à l’achat.

Á propos de la traduction en Farsi/Perse

En 2016, ce livre a été traduit par Mohsen Mostafa Jokar (qui est aussi connu dans la communauté iranienne pour sa traduction du manuel de Radare[31]). Il est disponible sur le site web de l’éditeur[32] (Pendare Pars).

Extrait de 40 pages: https://beginners.re/farsi.pdf.

Enregistrement du livre à la Bibliothèque Nationale d’Iran: http://opac.nlai.ir/opac-prod/bibliographic/4473995.

Á propos de la traduction en Chinois

En avril 2017, la traduction en Chinois a été terminée par Chinese PTPress. Ils sont également les détenteurs des droits de la traduction en Chinois.

La version chinoise est disponible à l’achat ici: http://www.epubit.com.cn/book/details/4174. Une revue partielle et l’historique de la traduction peut être trouvé ici: http://www.cptoday.cn/news/detail/3155.

Le traducteur principal est Archer, à qui je dois beaucoup. Il a été très méticuleux (dans le bon sens du terme) et a signalé la plupart des erreurs et bugs connus, ce qui est très important dans le genre de littérature de ce livre. Je recommanderais ses services à tout autre auteur!

Les gens de Antiy Labs ont aussi aidé pour la traduction. Voici la préface écrite par eux.

[31] http://rada.re/get/radare2book-persian.pdf
[32] http://goo.gl/2Tzx0H

Chapitre 1

Pattern de code

1.1 La méthode

Lorsque l'auteur de ce livre a commencé à apprendre le C, et plus tard, le C++, il a pris l'habitude d'écrire des petits morceaux de code, de les compiler et de regarder le langage d'assemblage généré. Cela lui a permis de comprendre facilement ce qui se passe dans le code qu'il écrit. [1] Il l'a fait si souvent que la relation entre le code C++ et ce que le compilateur produit a été imprimée profondément dans son esprit. Ça lui est facile d'imaginer immédiatement l'allure de la fonction et du code C. Peut-être que cette méthode pourrait être utile à d'autres.

Parfois, des anciens compilateurs sont utilisés, afin d'obtenir des extraits de code le plus court (ou le plus simple) possible.

À propos, il y a un bon site où vous pouvez faire la même chose, avec de nombreux compilateurs, au lieu de les installer sur votre système. Vous pouvez également l'utiliser: https://godbolt.org/.

Exercices

Lorsque l'auteur de ce livre étudiait le langage d'assemblage, il a souvent compilé des petites fonctions en C et les a ensuite récrites peu à peu en assembleur, en essayant d'obtenir un code aussi concis que possible. Cela n'en vaut probablement plus la peine aujourd'hui, car il est difficile de se mesurer aux derniers compilateurs en terme d'efficacité. Cela reste par contre un excellent moyen d'approfondir ses connaissances de l'assembleur. N'hésitez pas à prendre n'importe quel code assembleur de ce livre et à essayer de le rendre plus court. Toutefois, n'oubliez pas de tester ce que vous aurez écrit.

Niveau d'optimisation et information de débogage

Le code source peut être compilé par différents compilateurs, avec des niveaux d'optimisation variés. Un compilateur en a typiquement trois, où le niveau 0 désactive l'optimisation. L'optimisation peut se faire en ciblant la taille du code ou la vitesse d'exécution. Un compilateur sans optimisation est plus rapide et produit un code plus compréhensible (quoique verbeux), alors qu'un compilateur avec optimisation est plus lent et essaye de produire un code qui s'exécute plus vite (mais pas forcément plus compact). En plus des niveaux d'optimisation, un compilateur peut inclure dans le fichier généré des informations de débogage, qui produit un code facilitant le débogage. Une des caractéristiques importante du code de 'debug', est qu'il peut contenir des liens entre chaque ligne du code source et les adresses du code machine associé. D'un autre côté, l'optimisation des compilateurs tend à générer du code où des lignes du code source sont modifiées, et même parfois absentes du code machine résultant. Les rétro-ingénieurs peuvent rencontrer n'importe quelle version, simplement parce que certains développeurs mettent les options d'optimisation, et d'autres pas. Pour cette raison, et lorsque c'est possible, nous allons essayer de travailler sur des exemples avec les versions de débogage et finale du code présenté dans ce livre.

[1] En fait, il fait encore cela lorsqu'il ne comprend pas ce qu'un morceau de code fait.

1.2 Quelques bases

1.2.1 Une courte introduction sur le CPU

Le CPU est le système qui exécute le code machine que constitue le programme.

Un court glossaire:

Instruction : Une commande CPU primitive. Les exemples les plus simples incluent: déplacement de données entre les registres, travail avec la mémoire et les opérations arithmétiques primitives. Généralement, chaque CPU a son propre jeu d'instructions (ISA[2]).

Code machine : Code que le CPU exécute directement. Chaque instruction est codée sur plusieurs octets.

Langage d'assemblage : Code mnémotechnique et quelques extensions comme les macros qui facilitent la vie du programmeur.

Registre CPU : Chaque CPU a un ensemble de registres d'intérêt général (GPR[3]). ≈ 8 pour x86, ≈ 16 pour x86-64, ≈ 16 pour ARM. Le moyen le plus simple de comprendre un registre est de le voir comme une variable temporaire non-typée. Imaginez que vous travaillez avec un LP de haut niveau et que vous pouvez utiliser uniquement huit variables de 32-bit (ou de 64-bit). C'est malgré tout possible de faire beaucoup de choses en les utilisant!

On pourrait se demander pourquoi il y a besoin d'une différence entre le code machine et un LP. La réponse est que les humains et les CPUs ne sont pas semblables—c'est beaucoup plus simple pour les humains d'utiliser un LP de haut niveau comme C/C++, Java, Python, etc., mais c'est plus simple pour un CPU d'utiliser un niveau d'abstraction de beaucoup plus bas niveau. Peut-être qu'il serait possible d'inventer un CPU qui puisse exécuter du code d'un LP de haut niveau, mais il serait beaucoup plus complexe que les CPUs que nous connaissons aujourd'hui. D'une manière similaire, c'est moins facile pour les humains d'écrire en langage d'assemblage à cause de son bas niveau et de la difficulté d'écrire sans faire un nombre énorme de fautes agaçantes. Le programme qui convertit d'un LP haut niveau vers l'assemblage est appelé un *compilateur*.

Quelques mots sur les différents ISAs

Le jeu d'instructions ISA x86 a toujours été avec des instructions de taille variable. Donc quand l'époque du 64-bit arriva, les extensions x64 n'ont pas impacté le ISA très significativement. En fait, le ISA x86 contient toujours beaucoup d'instructions apparues pour la première fois dans un CPU 8086 16-bit, et que l'on trouve encore dans beaucoup de CPUs aujourd'hui. ARM est un CPU RISC[4] conçu avec l'idée d'instructions de taille fixe, ce qui présentait quelques avantages dans le passé. Au tout début, toutes les instructions ARM étaient codés sur 4 octets[5]. C'est maintenant connu comme le « ARM mode ». Ensuite ils sont arrivés à la conclusion que ce n'était pas aussi économique qu'ils l'avaient imaginé sur le principe. En réalité, la majorité des instructions CPU utilisées[6] dans le monde réel peuvent être encodées en utilisant moins d'informations. Ils ont par conséquent ajouté un autre ISA, appelé Thumb, où chaque instruction était encodée sur seulement 2 octets. C'est maintenant connu comme le « Thumb mode ». Cependant, toutes les instructions ne peuvent être encodées sur seulement 2 octets, donc les instructions Thumb sont un peu limitées. On peut noter que le code compilé pour le mode ARM et pour le mode Thumb peut, évidemment, coexister dans un seul programme. Les créateurs de ARM pensèrent que Thumb pourrait être étendu, donnant naissance à Thumb-2, qui apparut dans ARMv7. Thumb-2 utilise toujours des instructions de 2 octets, mais a de nouvelles instructions dont la taille est de 4 octets. Une erreur couramment répandue est que Thumb-2 est un mélange de ARM et Thumb. C'est incorrect. Plutôt, Thumb-2 fut étendu pour supporter totalement toutes les caractéristiques du processeur afin qu'il puisse rivaliser avec le mode ARM—un objectif qui a été clairement réussi, puisque la majorité des applications pour iPod/iPhone/iPad est compilée pour le jeu d'instructions de Thumb-2 (il est vrai que c'est largement dû au fait que Xcode le faisait par défaut). Plus tard, le ARM 64-bit sortit. Ce ISA a des instructions de 4 octets, et enlevait le besoin d'un mode Thumb supplémentaire. Cependant, les prérequis de 64-bit affectèrent le ISA, résultant maintenant au fait que nous avons trois jeux d'instructions ARM: ARM mode, Thumb mode (incluant Thumb-2) et ARM64. Ces ISAs s'intersectent partiellement, mais on peut dire que ce sont des ISAs différents, plutôt

[2] Instruction Set Architecture
[3] General Purpose Registers
[4] Reduced Instruction Set Computing
[5] D'ailleurs, les instructions de taille fixe sont pratiques parce qu'il est possible de calculer l'instruction suivante (ou précédente) sans effort. Cette caractéristique sera discutée dans la section de l'opérateur switch() (1.15.2 on page 173).
[6] Ce sont MOV/PUSH/CALL/Jcc

que des variantes du même. Par conséquent, nous essayerons d'ajouter des fragments de code dans les trois ISAs de ARM dans ce livre. Il y a, d'ailleurs, bien d'autres ISAs RISC avec des instructions de taille fixe de 32-bit, comme MIPS, PowerPC et Alpha AXP.

1.2.2 Systèmes de numération

Les Hommes ont probablement pris l'habitude d'utiliser la numérotation décimale parce qu'ils ont 10 doigts. Néanmoins, le nombre 10 n'a pas de signification particulière en science et en mathématiques. En électronique, le système de numérotation est le binaire : 0 pour l'absence de courant dans un fil et 1 s'il y en a. 10 en binaire est 2 en décimal; 100 en binaire est 4 en décimal et ainsi de suite.

Si le système de numération a 10 chiffres, il est en *base* 10. Le système binaire est en *base* 2.

Choses importantes à retenir:

1) Un *nombre* est un nombre, tandis qu'un *chiffre* est un élément d'un système d'écriture et est généralement un caractère

2) Un nombre ne change pas lorsqu'on le convertit dans une autre base; seule sa représentation écrite change (et donc la façon de le représenter en RAM[7]).

1.2.3 Conversion d'une base à une autre

La notation positionnelle est utilisée dans presque tous les systèmes de numération, cela signifie qu'un chiffre a un poids dépendant de sa position dans la représentation du nombre. Si 2 se situe le plus à droite, c'est 2. S'il est placé un chiffre avant celui le plus à droite, c'est 20.

Que représente 1234 ?

$10^3 \cdot 1 + 10^2 \cdot 2 + 10^1 \cdot 3 + 1 \cdot 4$ = 1234 ou $1000 \cdot 1 + 100 \cdot 2 + 10 \cdot 3 + 4 = 1234$

De même pour les nombres binaires, mais la base est 2 au lieu de 10. Que représente 0b101011 ?

$2^5 \cdot 1 + 2^4 \cdot 0 + 2^3 \cdot 1 + 2^2 \cdot 0 + 2^1 \cdot 1 + 2^0 \cdot 1 = 43$ ou $32 \cdot 1 + 16 \cdot 0 + 8 \cdot 1 + 4 \cdot 0 + 2 \cdot 1 + 1 = 43$

Il existe aussi la notation non-positionnelle comme la numération romaine[8]. Peut-être que l'humanité a choisi le système de numération positionnelle parce qu'il était plus simple pour les opérations basiques (addition, multiplication, etc.) à la main sur papier.

En effet, les nombres binaires peuvent être ajoutés, soustraits et ainsi de suite de la même manière que c'est enseigné à l'école, mais seulement 2 chiffres sont disponibles.

Les nombres binaires sont volumineux lorsqu'ils sont représentés dans le code source et les dumps, c'est pourquoi le système hexadécimal peut être utilisé. La base hexadécimale utilise les nombres 0..9 et aussi 6 caractères latins : A..F. Chaque chiffre hexadécimal prend 4 bits ou 4 chiffres binaires, donc c'est très simple de convertir un nombre binaire vers l'hexadécimal et inversement, même manuellement, de tête.

hexadécimal	binaire	décimal
0	0000	0
1	0001	1
2	0010	2
3	0011	3
4	0100	4
5	0101	5
6	0110	6
7	0111	7
8	1000	8
9	1001	9
A	1010	10
B	1011	11
C	1100	12
D	1101	13
E	1110	14
F	1111	15

[7] Random-Access Memory

[8] À propos de l'évolution du système de numération, voir [Donald E. Knuth, *The Art of Computer Programming*, Volume 2, 3rd ed., (1997), 195–213.]

Comment savoir quelle est la base utilisée dans un cas particulier?

Les nombres décimaux sont d'ordinaire écrits tels quels, i.e, 1234. Certains assembleurs permettent d'accentuer la base décimale, et les nombres peuvent alors s'écrire avec le suffixe "d" : 1234d.

Les nombres binaires sont parfois préfixés avec "0b" : 0b100110111 (GCC[9] a une extension de langage non-standard pour ça [10]). Il y a aussi un autre moyen : le suffixe "b", par exemple : 100110111b. J'essaierai de garder le préfixe "0b" tout le long du livre pour les nombres binaires.

Les nombres hexadécimaux sont préfixés avec "0x" en C/C++ et autres LPs : 0x1234ABCD. Ou ils ont le suffixe "h" : 1234ABCDh - c'est une manière commune de les représenter dans les assembleurs et les débogueurs. Si le nombre commence par un A..F, un 0 est ajouté au début : 0ABCDEFh. Il y avait une convention répandue à l'ère des ordinateurs personnels 8-bit, utilisant le préfixe $, comme $ABCD. J'essaierai de garder le préfixe "0x" tout le long du livre pour les nombres hexadécimaux.

Faut-il apprendre à convertir les nombres de tête? La table des nombres hexadécimaux de 1 chiffre peut facilement être mémorisée. Pour les nombres plus gros, ce n'est pas la peine de se tourmenter.

Peut-être que les nombres hexadécimaux les plus visibles sont dans les URL[11]s. C'est la façon d'encoder les caractères non-Latin. Par exemple: `https://en.wiktionary.org/wiki/na%C3%AFvet%C3%A9` est l'URL de l'article de Wiktionary à propos du mot « naïveté ».

Base octale

Un autre système de numération a été largement utilisé en informatique est la représentation octale. Elle comprend 8 chiffres (0..7), et chacun occupe 3 bits, donc c'est facile de convertir un nombre d'une base à l'autre. Il est maintenant remplacé par le système hexadécimal quasiment partout mais, chose surprenante, il y a encore une commande sur *NIX, utilisée par beaucoup de personnes, qui a un nombre octal comme argument : `chmod`.

Comme beaucoup d'utilisateurs *NIX le savent, l'argument de `chmod` peut être un nombre à 3 chiffres. Le premier correspond aux droits du propriétaire du fichier, le second correspond aux droits pour le groupe (auquel le fichier appartient), le troisième est pour tous les autres. Et chaque chiffre peut être représenté en binaire:

décimal	binaire	signification
7	111	**rwx**
6	110	**rw-**
5	101	**r-x**
4	100	**r--**
3	011	**-wx**
2	010	**-w-**
1	001	**--x**
0	000	**---**

Ainsi chaque bit correspond à un droit: lecture (r) / écriture (w) / exécution (x).

L'importance de `chmod` est que le nombre entier en argument peut être écrit comme un nombre octal. Prenons par exemple, 644. Quand vous tapez `chmod 644 file`, vous définissez les droits de lecture/écriture pour le propriétaire, les droits de lecture pour le groupe et encore les droits de lecture pour tous les autres. Convertissons le nombre octal 644 en binaire, ça donne `110100100`, ou (par groupe de 3 bits) `110 100 100`.

Maintenant que nous savons que chaque triplet sert à décrire les permissions pour le propriétaire/groupe/autres : le premier est `rw-`, le second est `r--` et le troisième est `r--`.

Le système de numération octal était aussi populaire sur les vieux ordinateurs comme le PDP-8 parce que les mots pouvaient être de 12, 24 ou de 36 bits et ces nombres sont divisibles par 3, donc la représentation octale était naturelle dans cet environnement. Aujourd'hui, tous les ordinateurs populaires utilisent des mots/taille d'adresse de 16, 32 ou de 64 bits et ces nombres sont divisibles par 4, donc la représentation hexadécimale était plus naturelle ici.

[9] GNU Compiler Collection
[10] https://gcc.gnu.org/onlinedocs/gcc/Binary-constants.html
[11] Uniform Resource Locator

Le système de numération octal est supporté par tous les compilateurs C/C++ standards. C'est parfois une source de confusion parce que les nombres octaux sont notés avec un zéro au début. Par exemple, 0377 est 255. Et parfois, vous faites une faute de frappe et écrivez "09" au lieu de 9, et le compilateur renvoie une erreur. GCC peut renvoyer quelque chose comme ça:
`erreur: chiffre 9 invalide dans la constante en base 8.`

De même, le système octal est assez populaire en Java. Lorsque IDA[12] affiche des chaînes Java avec des caractères non-imprimables, ils sont encodés dans le système octal au lieu d'hexadécimal. Le décompilateur Java JAD se comporte de la même façon.

Divisibilité

Quand vous voyez un nombre décimal comme 120, vous en déduisez immédiatement qu'il est divisible par 10, parce que le dernier chiffre est zéro. De la même façon, 123400 est divisible par 100 parce que les deux derniers chiffres sont zéros.

Pareillement, le nombre hexadécimal 0x1230 est divisible par 0x10 (ou 16), 0x123000 est divisible par 0x1000 (ou 4096), etc.

Un nombre binaire 0b1000101000 est divisible par 0b1000 (8), etc.

Cette propriété peut être souvent utilisée pour déterminer rapidement si l'adresse ou la taille d'un bloc mémoire correspond à une limite. Par exemple, les sections dans les fichiers PE[13] commencent quasiment toujours à une adresse finissant par 3 zéros hexadécimaux: 0x41000, 0x10001000, etc. La raison sous-jacente est que la plupart des sections PE sont alignées sur une limite de 0x1000 (4096) octets.

Arithmétique multi-précision et base

L'arithmétique multi-précision utilise des nombres très grands et chacun peut être stocké sur plusieurs octets. Par exemple, les clés RSA, tant publique que privée, utilisent jusqu'à 4096 bits et parfois plus encore.

Dans [Donald E. Knuth, *The Art of Computer Programming*, Volume 2, 3rd ed., (1997), 265] nous trouvons l'idée suivante: quand vous stockez un nombre multi-précision dans plusieurs octets, le nombre complet peut être représenté dans une base de $2^8 = 256$, et chacun des chiffres correspond à un octet. De la même manière, si vous sauvegardez un nombre multi-précision sur plusieurs entiers de 32 bits, chaque chiffre est associé à l'emplacement de 32 bits et vous pouvez penser à ce nombre comme étant stocké dans une base 2^{32}.

Comment prononcer les nombres non-décimaux

Les nombres dans une base non décimale sont généralement prononcés un chiffre à la fois : "un-zéro-zéro-un-un-...". Les mots comme "dix", "mille", etc, ne sont généralement pas prononcés, pour éviter d'être confondus avec ceux en base décimale.

Nombres à virgule flottante

Pour distinguer les nombres à virgule flottante des entiers, ils sont souvent écrits avec avec un ".0" à la fin, comme 0.0, 123.0, etc.

1.3 Fonction vide

La fonction la plus simple possible est sans doute celle qui ne fait rien:

Listing 1.1: Code C/C++

```
void f()
{
        return;
```

[12] Désassembleur interactif et débogueur développé par Hex-Rays
[13] Portable Executable

```
};
```

Compilons-la!

1.3.1 x86

Voici ce que les compilateurs GCC et MSVC produisent sur une plateforme x86:

Listing 1.2: GCC/MSVC avec optimisation (résultat en sortie de l'assembleur)

```
f :
        ret
```

Il y a juste une instruction: RET, qui détourne l'exécution vers l'appelant.

1.3.2 ARM

Listing 1.3: avec optimisation Keil 6/2013 (Mode ARM) ASM Output

```
f       PROC
        BX       lr
        ENDP
```

L'adresse de retour n'est pas stockée sur la pile locale avec l'ISA ARM, mais dans le "link register" (registre de lien), donc l'instruction BX LR force le flux d'exécution à sauter à cette adresse—renvoyant effectivement l'exécution vers l'appelant.

1.3.3 MIPS

Il y a deux conventions de nommage utilisées dans le monde MIPS pour nommer les registres: par numéro (de $0 à $31) ou par un pseudo-nom ($V0, $A0, etc.).

La sortie de l'assembleur GCC ci-dessous liste les registres par numéro:

Listing 1.4: GCC 4.4.5 avec optimisation (résultat en sortie de l'assembleur)

```
        j       $31
        nop
```

...tandis qu'IDA le fait—avec les pseudo noms:

Listing 1.5: GCC 4.4.5 avec optimisation (IDA)

```
        j       $ra
        nop
```

La première instruction est l'instruction de saut (J ou JR) qui détourne le flux d'exécution vers l'appelant, sautant à l'adresse dans le registre $31 (ou $RA).

Ce registre est similaire à LR[14] en ARM.

La seconde instruction est NOP[15], qui ne fait rien. Nous pouvons l'ignorer pour l'instant.

Une note à propos des instructions MIPS et des noms de registres

Les registres et les noms des instructions dans le monde de MIPS sont traditionnellement écrits en minuscules. Cependant, dans un souci d'homogénéité, nous allons continuer d'utiliser les lettres majuscules, étant donné que c'est la convention suivie par tous les autres ISAs présentés dans ce livre.

[14] Link Register
[15] No Operation

1.3.4 Fonctions vides en pratique

Bien que les fonctions vides soient inutiles, elles sont assez fréquentes dans le code bas niveau.

Tout d'abord, les fonctions de débogage sont assez populaires, comme celle-ci:

Listing 1.6: Code C/C++

```
void dbg_print (const char *fmt, ...)
{
#ifdef _DEBUG
        // open log file
        // write to log file
        // close log file
#endif
};

void some_function()
{
        ...

        dbg_print ("we did something\n");

        ...
};
```

Dans une compilation en non-debug (e.g., "release"), `_DEBUG` n'est pas défini, donc la fonction `dbg_print()`, bien qu'elle soit appelée pendant l'exécution, sera vide.

Un autre moyen de protection logicielle est de faire plusieurs compilations: une pour les clients, une de démonstration. La compilation de démonstration peut omettre certaines fonctions importantes, comme ici:

Listing 1.7: Code C/C++

```
void save_file ()
{
#ifndef DEMO
        // actual saving code
#endif
};
```

La fonction `save_file()` peut être appelée lorsque l'utilisateur clique sur le menu `Fichier->Enregistrer`. La version de démo peut être livrée avec cet item du menu désactivé, mais même si un logiciel cracker pourra l'activer, une fonction vide sans code utile sera appelée.

IDA signale de telles fonctions avec des noms comme `nullsub_00`, `nullsub_01`, etc.

1.4 Valeur de retour

Une autre fonction simple est celle qui retourne juste une valeur constante:

La voici:

Listing 1.8: Code C/C++

```
int f()
{
        return 123;
};
```

Compilons la!

1.4.1 x86

Voici ce que les compilateurs GCC et MSVC produisent sur une plateforme x86:

Listing 1.9: GCC/MSVC avec optimisation (résultat en sortie de l'assembleur)

```
f :
        mov     eax, 123
        ret
```

Il y a juste deux instructions: la première place la valeur 123 dans le registre EAX, qui est par convention le registre utilisé pour stocker la valeur renvoyée d'une fonction et la seconde est RET, qui retourne l'exécution vers l'appelant.

L'appelant prendra le résultat de cette fonction dans le registre EAX.

1.4.2 ARM

Il y a quelques différences sur la platforme ARM:

Listing 1.10: avec optimisation Keil 6/2013 (Mode ARM) ASM Output

```
f       PROC
        MOV       r0,#0x7b  ; 123
        BX        lr
        ENDP
```

ARM utilise le registre R0 pour renvoyer le résultat d'une fonction, donc 123 est copié dans R0.

Il est à noter que l'instruction MOV est trompeuse pour les plateformes x86 et ARM ISAs.

La donnée n'est en réalité pas *déplacée (moved)* mais *copiée*.

1.4.3 MIPS

La sortie de l'assembleur GCC ci-dessous indique les registres par numéro:

Listing 1.11: GCC 4.4.5 avec optimisation (résultat en sortie de l'assembleur)

```
        j       $31
        li      $2,123                  # 0x7b
```

...tandis qu'IDA le fait—avec les pseudo noms:

Listing 1.12: GCC 4.4.5 avec optimisation (IDA)

```
                jr      $ra
                li      $v0, 0x7B
```

Le registre $2 (ou $V0) est utilisé pour stocker la valeur de retour de la fonction. LI signifie "Load Immediate" et est l'équivalent MIPS de MOV.

L'autre instruction est l'instruction de saut (J ou JR) qui retourne le flux d'exécution vers l'appelant.

Vous pouvez vous demander pourquoi la position de l'instruction d'affectation de valeur immédiate (LI) et l'instruction de saut (J ou JR) sont échangées. Ceci est dû à une fonctionnalité du RISC appelée "branch delay slot" (slot de délai de branchement).

La raison de cela est du à une bizarrerie dans l'architecture de certains RISC ISAs et n'est pas importante pour nous. Nous gardons juste en tête qu'en MIPS, l'instruction qui suit une instruction de saut ou de branchement est exécutée *avant* l'instruction de saut ou de branchement elle-même.

Par conséquent, les instructions de branchement échangent toujours leur place avec l'instruction qui doit être exécutée avant.

1.4.4 En pratique

Les fonctions qui retournent simplement 1 (*true*) ou 0 (*false*) sont vraiment fréquentes en pratique.

Les plus petits utilitaires UNIX standard, */bin/true* et */bin/false* renvoient respectivement 0 et 1, comme code de retour. (un code retour de zéro signifie en général succès, non-zéro une erreur).

1.5 Hello, world!

Utilisons le fameux exemple du livre [Brian W. Kernighan, Dennis M. Ritchie, *The C Programming Language*, 2ed, (1988)]:

```
#include <stdio.h>

int main()
{
    printf("hello, world\n");
    return 0;
}
```

1.5.1 x86

MSVC

Compilons-le avec MSVC 2010:

```
cl 1.cpp /Fa1.asm
```

(L'option /Fa indique au compilateur de générer un fichier avec le listing en assembleur)

Listing 1.13: MSVC 2010

```
CONST   SEGMENT
$SG3830 DB      'hello, world', 0AH, 00H
CONST   ENDS
PUBLIC  _main
EXTRN   _printf :PROC
; Function compile flags : /Odtp
_TEXT   SEGMENT
_main   PROC
        push    ebp
        mov     ebp, esp
        push    OFFSET $SG3830
        call    _printf
        add     esp, 4
        xor     eax, eax
        pop     ebp
        ret     0
_main   ENDP
_TEXT   ENDS
```

MSVC génère des listings assembleur avec la syntaxe Intel. La différence entre la syntaxe Intel et la syntaxe AT&T sera discutée dans 1.5.1 on page 11.

Le compilateur a généré le fichier object `1.obj`, qui sera lié dans l'exécutable `1.exe`. Dans notre cas, le fichier contient deux segments: `CONST` (pour les données constantes) et _TEXT (pour le code).

La chaîne `hello, world` en C/C++ a le type `const char[]`[Bjarne Stroustrup, *The C++ Programming Language, 4th Edition*, (2013)p176, 7.3.2], mais elle n'a pas de nom. Le compilateur doit pouvoir l'utiliser et lui défini donc le nom interne `$SG3830` à cette fin.

C'est pourquoi l'exemple pourrait être récrit comme suit:

```
#include <stdio.h>

const char $SG3830[]="hello, world\n";

int main()
{
    printf($SG3830);
    return 0;
}
```

Retournons au listing assembleur. Comme nous le voyons, la chaîne est terminée avec un octet à zéro, ce qui est le standard pour les chaînes C/C++.

Dans le segment de code, _TEXT, il n'y a qu'une seule fonction: `main()`. La fonction `main()` débute par le code du prologue et se termine par le code de l'épilogue (comme presque toutes les fonctions) [16].

Après le prologue de la fonction nous voyons l'appel à la fonction `printf()` :
`CALL _printf`. Avant l'appel, l'adresse de la chaîne (ou un pointeur sur elle) contenant notre message est placée sur la pile avec l'aide de l'instruction PUSH.

Lorsque la fonction `printf()` rend le contrôle à la fonction `main()`, l'adresse de la chaîne (ou un pointeur sur elle) est toujours sur la pile. Comme nous n'en avons plus besoin, le pointeur de pile (stack pointer le registre ESP) doit être corrigé.

`ADD ESP, 4` signifie ajouter 4 à la valeur du registre ESP.

Pourquoi 4? puisqu'il s'agit d'un programme 32-bit, nous avons besoin d'exactement 4 octets pour passer une adresse par la pile. S'il s'agissait d'un code x64, nous aurions besoin de 8 octets. `ADD ESP, 4` est effectivement équivalent à `POP register` mais sans utiliser de registre[17].

Pour la même raison, certains compilateurs (comme le compilateur C++ d'Intel) peuvent générer POP ECX à la place de ADD (e.g., ce comportement peut être observé dans le code d'Oracle RDBMS car il est compilé avec le compilateur C++ d'Intel. Cette instruction a presque le même effet mais le contenu du registre ECX sera écrasé. Le compilateur C++ d'Intel utilise probablement `POP ECX` car l'opcode de cette instruction est plus court que celui de `ADD ESP, x` (1 octet pour POP contre 3 pour ADD).

Voici un exemple d'utilisation de POP à la place de ADD dans Oracle RDBMS :

Listing 1.14: Oracle RDBMS 10.2 Linux (app.o file)

```
.text :0800029A                push    ebx
.text :0800029B                call    qksfroChild
.text :080002A0                pop     ecx
```

Après l'appel de `printf()`, le code C/C++ original contient la déclaration `return 0` —renvoie 0 comme valeur de retour de la fonction `main()`.

Dans le code généré cela est implémenté par l'instruction `XOR EAX, EAX`.

XOR est en fait un simple « OU exclusif (eXclusive OR »[18] mais les compilateurs l'utilisent souvent à la place de `MOV EAX, 0`—à nouveau parce que l'opcode est légèrement plus court (2 octets pour XOR contre 5 pour MOV).

Certains compilateurs génèrent `SUB EAX, EAX`, qui signifie *Soustraire la valeur dans* EAX *de la valeur dans* EAX, ce qui, dans tous les cas, donne zéro.

La dernière instruction RET redonne le contrôle à l'appelant. Habituellement, c'est ce code C/C++ CRT[19], qui, à son tour, redonne le contrôle à l'OS.

GCC

Maintenant compilons le même code C/C++ avec le compilateur GCC 4.4.1 sur Linux: `gcc 1.c -o 1`. Ensuite, avec l'aide du désassembleur IDA, regardons comment la fonction `main()` a été créée. IDA, comme MSVC, utilise la syntaxe Intel[20].

Listing 1.15: code in IDA

```
main            proc near

var_10          = dword ptr -10h

                push    ebp
                mov     ebp, esp
                and     esp, 0FFFFFFF0h
                sub     esp, 10h
                mov     eax, offset aHelloWorld ; "hello, world\n"
```

[16] Vous pouvez en lire plus dans la section concernant les prologues et épilogues de fonctions (1.6 on page 30).
[17] Les flags du CPU, toutefois, sont modifiés
[18] Wikipédia
[19] C Runtime library
[20] GCC peut aussi produire un listing assembleur utilisant la syntaxe Intel en lui passant les options `-S -masm=intel`.

```
                mov     [esp+10h+var_10], eax
                call    _printf
                mov     eax, 0
                leave
                retn
main            endp
```

Le résultat est presque le même. L'adresse de la chaîne `hello, world` (stockée dans le segment de donnée) est d'abord chargée dans le registre EAX puis sauvée sur la pile.
En plus, le prologue de la fonction comprend `AND ESP, 0FFFFFFF0h` —cette instruction aligne le registre ESP sur une limite de 16-octet. Ainsi, toutes les valeurs sur la pile seront alignées de la même manière (Le CPU est plus performant si les adresses avec lesquelles il travaille en mémoire sont alignées sur des limites de 4-octet ou 16-octet)[21].

`SUB ESP, 10h` réserve 16 octets sur la pile. Pourtant, comme nous allons le voir, seuls 4 sont nécessaires ici.

C'est parce que la taille de la pile allouée est alignée sur une limite de 16-octet.

L'adresse de la chaîne est (ou un pointeur vers la chaîne) est stockée directement sur la pile sans utiliser l'instruction PUSH. *var_10* —est une variable locale et est aussi un argument pour `printf()`. Lisez à ce propos en dessous.

Ensuite la fonction `printf()` est appelée.

Contrairement à MSVC, lorsque GCC compile sans optimisation, il génère `MOV EAX, 0` au lieu d'un opcode plus court.

La dernière instruction, `LEAVE` —est équivalente à la paire d'instruction `MOV ESP, EBP` et `POP EBP` —en d'autres mots, cette instruction déplace le pointeur de pile (ESP) et remet le registre EBP dans son état initial. Ceci est nécessaire puisque nous avons modifié les valeurs de ces registres (ESP et EBP) au début de la fonction (en exécutant `MOV EBP, ESP` / `AND ESP, …`).

GCC: Syntaxe AT&T

Voyons comment cela peut-être représenté en langage d'assemblage avec la syntaxe AT&T. Cette syntaxe est bien plus populaire dans le monde UNIX.

Listing 1.16: compilons avec GCC 4.7.3

```
gcc -S 1_1.c
```

Nous obtenons ceci:

Listing 1.17: GCC 4.7.3

```
        .file   "1_1.c"
        .section        .rodata
.LC0 :
        .string "hello, world\n"
        .text
        .globl  main
        .type   main, @function
main :
.LFB0 :
        .cfi_startproc
        pushl   %ebp
        .cfi_def_cfa_offset 8
        .cfi_offset 5, -8
        movl    %esp, %ebp
        .cfi_def_cfa_register 5
        andl    $-16, %esp
        subl    $16, %esp
        movl    $.LC0, (%esp)
        call    printf
        movl    $0, %eax
        leave
        .cfi_restore 5
```

[21] Wikipedia: Alignement en mémoire

```
        .cfi_def_cfa 4, 4
        ret
        .cfi_endproc
.LFE0 :
        .size   main, .-main
        .ident  "GCC : (Ubuntu/Linaro 4.7.3-1ubuntu1) 4.7.3"
        .section        .note.GNU-stack,"",@progbits
```

Le listing contient beaucoup de macros (qui commencent avec un point). Cela ne nous intéresse pas pour le moment.

Pour l'instant, dans un souci de simplification, nous pouvons les ignorer (excepté la macro *.string* qui encode une séquence de caractères terminée par un octet nul, comme une chaîne C). Ensuite nous verrons cela[22] :

Listing 1.18: GCC 4.7.3

```
.LC0 :
        .string "hello, world\n"
main :
        pushl   %ebp
        movl    %esp, %ebp
        andl    $-16, %esp
        subl    $16, %esp
        movl    $.LC0, (%esp)
        call    printf
        movl    $0, %eax
        leave
        ret
```

Quelques-une des différences majeures entre la syntaxe Intel et AT&T sont:

- Opérandes source et destination sont écrites dans l'ordre inverse.

  En syntaxe Intel: <instruction> <opérande de destination> <opérande source>.

  En syntaxe AT&T: <instruction> <opérande source> <opérande de destination>.

  Voici un moyen simple de mémoriser la différence: lorsque vous avez affaire avec la syntaxe Intel, vous pouvez imaginer qu'il y a un signe égal (=) entre les opérandes et lorsque vous avez affaire avec la syntaxe AT&T imaginez qu'il y a un flèche droite (→) [23].

- AT&T: Avant les noms de registres, un signe pourcent doit être écrit (%) et avant les nombres, un signe dollar ($). Les parenthèses sont utilisées à la place des crochets.
- AT&T: un suffixe est ajouté à l'instruction pour définir la taille de l'opérande:
  - q — quad (64 bits)
  - l — long (32 bits)
  - w — word (16 bits)
  - b — byte (8 bits)

Retournons au résultat compilé: il est identique à ce que l'on voit dans IDA. Avec une différence subtile: `0FFFFFFF0h` est représenté avec `$-16`. C'est la même chose: 16 dans le système décimal est `0x10` en hexadécimal. `-0x10` est équivalent à `0xFFFFFFF0` (pour un type de donnée sur 32-bit).

Encore une chose: la valeur de retour est mise à 0 en utilisant un `MOV` usuel, pas un `XOR`. `MOV` charge seulement la valeur dans le registre. Le nom est mal choisi (la donnée n'est pas déplacée, mais plutôt copiée). Dans d'autres architectures, cette instruction est nommée « LOAD » ou « STORE » ou quelque chose de similaire.

Modification de chaînes (Win32)

Nous pouvons facilement trouver la chaîne "hello, world" dans l'exécutable en utilisant Hiew:

[22] Cette option de GCC peut être utilisée pour éliminer les macros « non nécessaires » : *-fno-asynchronous-unwind-tables*

[23] À propos, dans certaine fonction C standard (e.g., memcpy(), strcpy()) les arguments sont listés de la même manière que dans la syntaxe Intel: en premier se trouve le pointeur du bloc mémoire de destination, et ensuite le pointeur sur le bloc mémoire source.

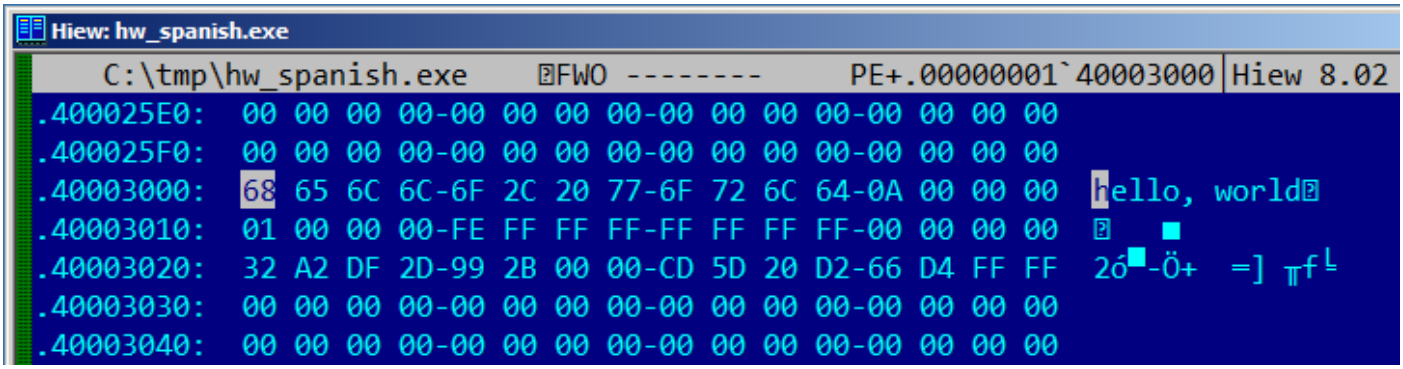


Fig. 1.1: Hiew

Et nous pouvons essayer de traduire notre message en espagnol:

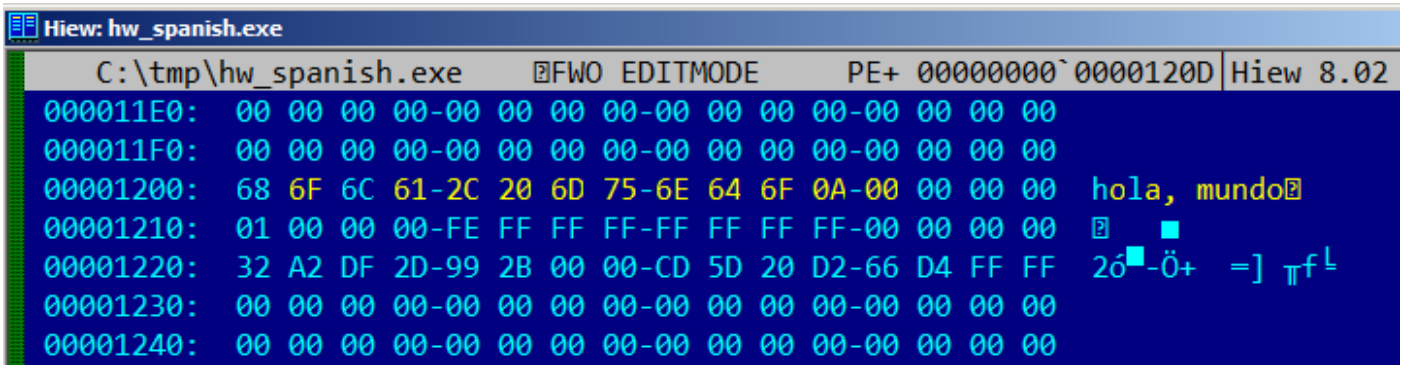


Fig. 1.2: Hiew

Le texte en espagnol est un octet plus court que celui en anglais, nous ajoutons l'octet 0x0A à la fin (\n) ainsi qu'un octet à zéro.

Ça fonctionne.

Comment faire si nous voulons insérer un message plus long ? Il y a quelques octets à zéro après le texte original en anglais. Il est difficile de dire s'ils peuvent être écrasés: ils peuvent être utilisés quelque part dans du code CRT, ou pas. De toutes façons, écrasez-les seulement si vous savez vraiment ce que vous faîtes.

Modification de chaînes (Linux x64)

Essayons de modifier un exécutable Linux x64 en utilisant rada.re :

Listing 1.19: rada.re session

```
dennis@bigbox ~/tmp % gcc hw.c

dennis@bigbox ~/tmp % radare2 a.out
 -- SHALL WE PLAY A GAME?
[0x00400430]> / hello
Searching 5 bytes from 0x00400000 to 0x00601040 : 68 65 6c 6c 6f
Searching 5 bytes in [0x400000-0x601040]
hits : 1
0x004005c4 hit0_0 .HHhello, world;0.

[0x00400430]> s 0x004005c4

[0x004005c4]> px
- offset -   0 1  2 3  4 5  6 7  8 9  A B  C D  E F  0123456789ABCDEF
0x004005c4  6865 6c6c 6f2c 2077 6f72 6c64 0000 0000  hello, world....
0x004005d4  011b 033b 3000 0000 0500 0000 1cfe ffff  ...;0...........
```

```
0x004005e4  7c00 0000 5cfe ffff 4c00 0000 52ff ffff  |...\...L...R...
0x004005f4  a400 0000 6cff ffff c400 0000 dcff ffff  ....l...........
0x00400604  0c01 0000 1400 0000 0000 0000 017a 5200  .............zR.
0x00400614  0178 1001 1b0c 0708 9001 0710 1400 0000  .x..............
0x00400624  1c00 0000 08fe ffff 2a00 0000 0000 0000  ........*.......
0x00400634  0000 0000 1400 0000 0000 0000 017a 5200  .............zR.
0x00400644  0178 1001 1b0c 0708 9001 0000 2400 0000  .x..........$...
0x00400654  1c00 0000 98fd ffff 3000 0000 000e 1046  ........0......F
0x00400664  0e18 4a0f 0b77 0880 003f 1a3b 2a33 2422  ..J..w...?.;*3$"
0x00400674  0000 0000 1c00 0000 4400 0000 a6fe ffff  ........D.......
0x00400684  1500 0000 0041 0e10 8602 430d 0650 0c07  .....A....C..P..
0x00400694  0800 0000 4400 0000 6400 0000 a0fe ffff  ....D...d.......
0x004006a4  6500 0000 0042 0e10 8f02 420e 188e 0345  e....B....B....E
0x004006b4  0e20 8d04 420e 288c 0548 0e30 8606 480e  . ..B.(..H.0..H.

[0x004005c4]> oo+
File a.out reopened in read-write mode

[0x004005c4]> w hola, mundo\x00

[0x004005c4]> q

dennis@bigbox ~/tmp % ./a.out
hola, mundo
```

Ce que je fais ici: je cherche la chaîne « hello » en utilisant la commande `/`, ensuite je déplace le *curseur* (ou *seek* selon la terminologie de rada.re) à cette adresse. Je veux être certain d'être à la bonne adresse: `px` affiche les octets ici. `oo+` passe rada.re en mode *read-write*. `w` écrit une chaîne ASCII à la *seek* (*position*) courante. Notez le `\00` à la fin–c'est l'octet à zéro. `q` quitte.

Traduction de logiciel à l'ère MS-DOS

La méthode que je viens de décrire était couramment employée pour traduire des logiciels sous MS-DOS en russe dans les années 1980 et 1990. Les mots et les phrases russes sont en général un peu plus longs qu'en anglais, c'est pourquoi les logiciels *traduits* sont pleins d'acronymes sibyllins et d'abréviations difficilement lisibles.

Peut-être que cela s'est produit pour d'autres langages durant cette période.

1.5.2 x86-64

MSVC: x86-64

Essayons MSVC 64-bit:

Listing 1.20: MSVC 2012 x64

```
$SG2989 DB      'hello, world', 0AH, 00H

main    PROC
        sub     rsp, 40
        lea     rcx, OFFSET FLAT :$SG2989
        call    printf
        xor     eax, eax
        add     rsp, 40
        ret     0
main    ENDP
```

En x86-64, tous les registres ont été étendus à 64-bit et leurs noms ont maintenant le préfixe `R-`. Afin d'utiliser la pile moins souvent (en d'autres termes, pour accéder moins souvent à la mémoire externe/au cache), il existe un moyen répandu de passer les arguments aux fonctions par les registres (*fastcall*) 4.1.3 on page 545. I.e., une partie des arguments de la fonction est passée par les registres, le reste—par la pile. En Win64, 4 arguments de fonction sont passés dans les registres RCX, RDX, R8, R9. C'est ce que l'on voit ci-dessus: un pointeur sur la chaîne pour `printf()` est passé non pas par la pile, mais par le registre RCX. Les pointeurs font maintenant 64-bit, ils sont donc passés dans les registres 64-bit (qui ont le préfixe

R-). Toutefois, pour la rétrocompatibilité, il est toujours possible d'accéder à la partie 32-bits des registres, en utilisant le préfixe E-. Voici à quoi ressemblent les registres RAX/EAX/AX/AL en x86-64:

Octet d'indice							
7	6	5	4	3	2	1	0
RAX^{x64}							
				EAX			
						AX	
						AH	AL

La fonction `main()` renvoie un type *int*, qui est, en C/C++, pour une meilleure rétrocompatibilité et portabilité, toujours 32-bit, c'est pourquoi le registre EAX est mis à zéro à la fin de la fonction (i.e., la partie 32-bit du registre) au lieu de RAX. Il y aussi 40 octets alloués sur la pile locale. Cela est appelé le « shadow space », dont nous parlerons plus tard: 1.10.2 on page 101.

GCC: x86-64

Essayons GCC sur un Linux 64-bit:

Listing 1.21: GCC 4.4.6 x64

```
.string "hello, world\n"
main :
        sub     rsp, 8
        mov     edi, OFFSET FLAT :.LC0 ; "hello, world\n"
        xor     eax, eax  ; nombre de registres vectoriels
        call    printf
        xor     eax, eax
        add     rsp, 8
        ret
```

Une méthode de passage des arguments à la fonction dans des registres est aussi utilisée sur Linux, *BSD et Mac OS X est [Michael Matz, Jan Hubicka, Andreas Jaeger, Mark Mitchell, *System V Application Binary Interface. AMD64 Architecture Processor Supplement*, (2013)] [24]. Linux, *BSD et Mac OS X utilisent aussi une méthode pour passer les arguments d'une fonction par les registres: [Michael Matz, Jan Hubicka, Andreas Jaeger, Mark Mitchell, *System V Application Binary Interface. AMD64 Architecture Processor Supplement*, (2013)] [25].

Les 6 premiers arguments sont passés dans les registres RDI, RSI, RDX, RCX, R8, R9 et les autres—par la pile.

Donc le pointeur sur la chaîne est passé dans EDI (la partie 32-bit du registre). Mais pourquoi ne pas utiliser la partie 64-bit, RDI ?

Il est important de garder à l'esprit que toutes les instructions MOV en mode 64-bit qui écrivent quelque chose dans la partie 32-bit inférieuaer du registre efface également les 32-bit supérieurs (comme indiqué dans les manuels Intel: 8.1.4 on page 649).
I.e., l'instruction `MOV EAX, 011223344h` écrit correctement une valeur dans RAX, puisque que les bits supérieurs sont mis à zéro.

Si nous ouvrons le fichier objet compilé (.o), nous pouvons voir tous les opcodes des instructions [26] :

Listing 1.22: GCC 4.4.6 x64

```
.text :00000000004004D0                 main  proc near
.text :00000000004004D0 48 83 EC 08     sub     rsp, 8
.text :00000000004004D4 BF E8 05 40 00  mov     edi, offset format ; "hello, world\n"
.text :00000000004004D9 31 C0           xor     eax, eax
.text :00000000004004DB E8 D8 FE FF FF  call    _printf
.text :00000000004004E0 31 C0           xor     eax, eax
.text :00000000004004E2 48 83 C4 08     add     rsp, 8
.text :00000000004004E6 C3              retn
.text :00000000004004E6                 main  endp
```

[24] Aussi disponible en https://software.intel.com/sites/default/files/article/402129/mpx-linux64-abi.pdf
[25] Aussi disponible en https://software.intel.com/sites/default/files/article/402129/mpx-linux64-abi.pdf
[26] Ceci doit être activé dans **Options** → **Disassembly** → **Number of opcode bytes**

Comme on le voit, l'instruction qui écrit dans EDI en 0x4004D4 occupe 5 octets. La même instruction qui écrit une valeur sur 64-bit dans RDI occupe 7 octets. Il semble que GCC essaye d'économiser un peu d'espace. En outre, cela permet d'être sûr que le segment de données contenant la chaîne ne sera pas alloué à une adresse supérieure à 4 GiB.

Nous voyons aussi que le registre EAX est mis à zéro avant l'appel à la fonction `printf()`. Ceci, car conformément à l'ABI[27] standard mentionnée plus haut, le nombre de registres vectoriel utilisés est passé dans EAX sur les systèmes *NIX en x86-64.

Modification d'adresse (Win64)

Lorsque notre exemple est compilé sous MSVC 2013 avec l'option \MD (générant un exécutable plus petit du fait du lien avec MSVCR*.DLL), la fonction `main()` vient en premier et est trouvée facilement:

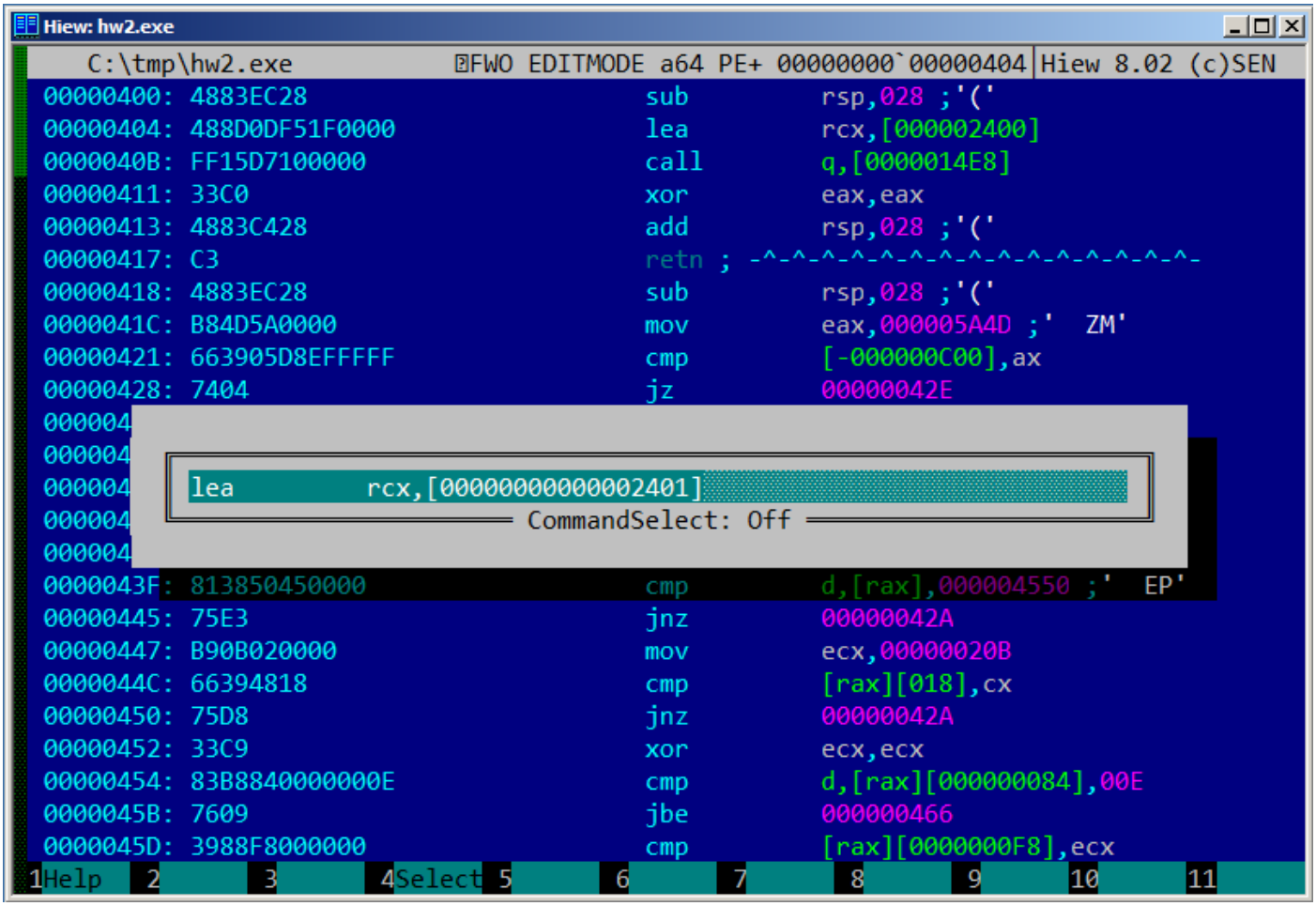


Fig. 1.3: Hiew

A titre expérimental, nous pouvons incrémenter l'adresse du pointeur de 1:

[27] Application Binary Interface

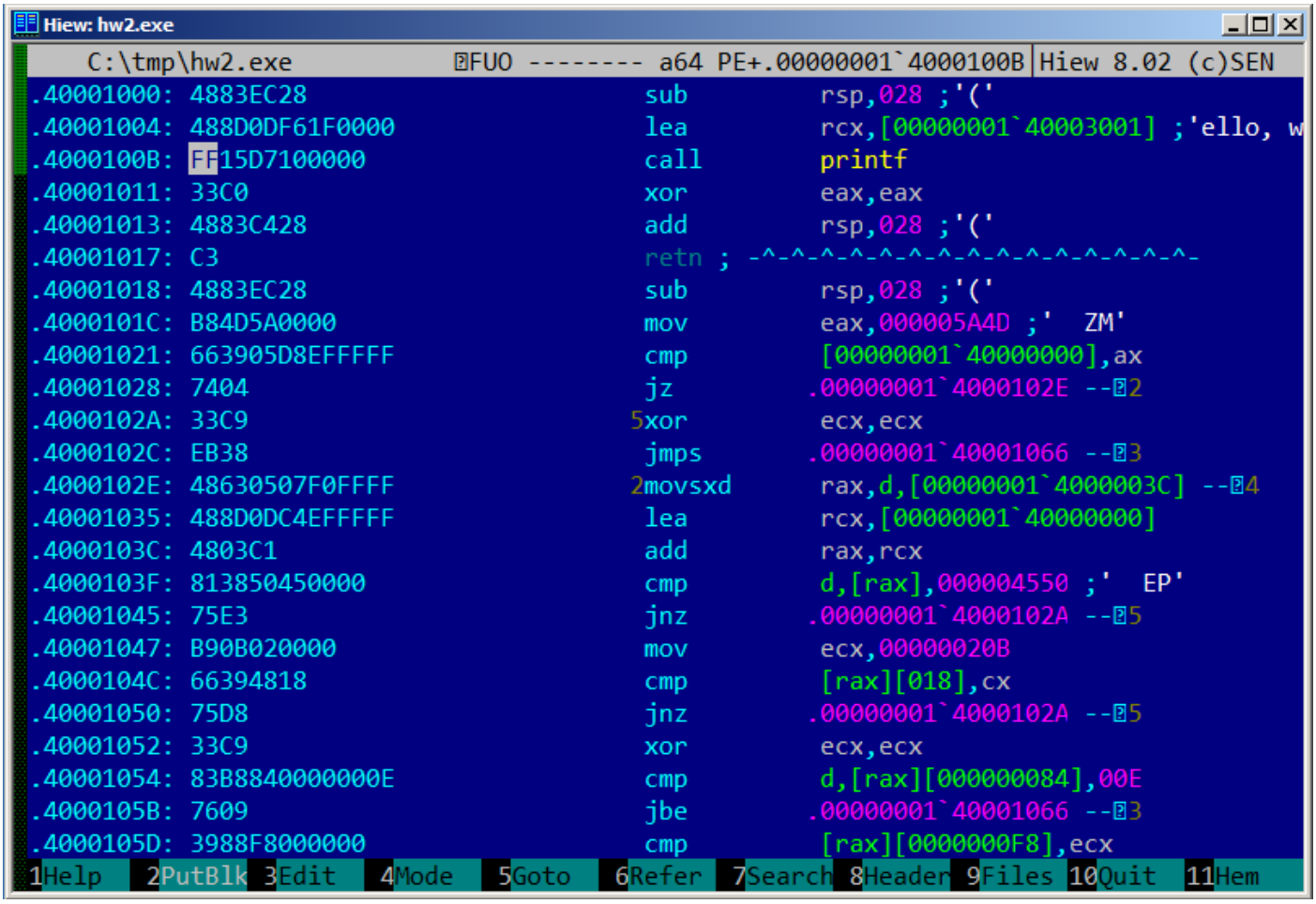


Fig. 1.4: Hiew

Hiew montre la chaîne « ello, world ». Et lorsque nous lançons l'exécutable modifié, la chaîne raccourcie est affichée.

Utiliser une autre chaîne d'un binaire (Linux x64)

Le fichier binaire que j'obtiens en compilant notre exemple avec GCC 5.4.0 sur un système Linux x64 contient de nombreuses autres chaînes: la plupart sont des noms de fonction et de bibliothèque importées.

Je lance *objdump* pour voir le contenu de toutes les sections du fichier compilé:

```
$ objdump -s a.out

a.out :     file format elf64-x86-64

Contents of section .interp :
 400238 2f6c6962 36342f6c 642d6c69 6e75782d  /lib64/ld-linux-
 400248 7838362d 36342e73 6f2e3200           x86-64.so.2.
Contents of section .note.ABI-tag :
 400254 04000000 10000000 01000000 474e5500  ............GNU.
 400264 00000000 02000000 06000000 20000000  ............ ...
Contents of section .note.gnu.build-id :
 400274 04000000 14000000 03000000 474e5500  ............GNU.
 400284 fe461178 5bb710b4 bbf2aca8 5ec1ec10  .F.x[.......^...
 400294 cf3f7ae4                             .?z.

...
```

Ce n'est pas un problème de passer l'adresse de la chaîne « /lib64/ld-linux-x86-64.so.2 » à l'appel de `printf()` :

```
#include <stdio.h>

int main()
{
    printf(0x400238);
    return 0;
}
```

Difficile à croire, ce code affiche la chaîne mentionnée.

Changez l'adresse en 0x400260, et la chaîne « GNU » sera affichée. L'adresse est valable pour cette version spécifique de GCC, outils GNU, etc. Sur votre système, l'exécutable peut être légèrement différent, et toutes les adresses seront différentes. Ainsi, ajouter/supprimer du code à/de ce code source va probablement décaler les adresses en arrière et avant.

1.5.3 GCC—encore une chose

Le fait est qu'une chaîne C *anonyme* a un type *const* (1.5.1 on page 9), et que les chaînes C allouées dans le segment des constantes sont garanties d'être immuables, a cette conséquence intéressante: Le compilateur peut utiliser une partie spécifique de la chaîne.

Voyons cela avec un exemple:

```
#include <stdio.h>

int f1()
{
        printf ("world\n");
}

int f2()
{
        printf ("hello world\n");
}

int main()
{
        f1();
        f2();
}
```

La plupart des compilateurs C/C++ (MSVC inclus) allouent deux chaînes, mais voyons ce que fait GCC 4.8.1:

Listing 1.23: GCC 4.8.1 + IDA listing

```
f1              proc near

s               = dword ptr -1Ch

                sub     esp, 1Ch
                mov     [esp+1Ch+s], offset s ; "world\n"
                call    _puts
                add     esp, 1Ch
                retn
f1              endp

f2              proc near

s               = dword ptr -1Ch

                sub     esp, 1Ch
                mov     [esp+1Ch+s], offset aHello ; "hello "
                call    _puts
                add     esp, 1Ch
                retn
f2              endp
```

```
aHello          db 'hello '
s               db 'world',0xa,0
```

Effectivement: lorsque nous affichons la chaîne « hello world » ses deux mots sont positionnés consécutivement en mémoire et l'appel à `puts()` depuis la fonction `f2()` n'est pas au courant que la chaîne est divisée. En fait, elle n'est pas divisée; elle l'est « virtuellement », dans ce listing.

Lorsque `puts()` est appelé depuis `f1()`, il utilise la chaîne « world » ainsi qu'un octet à zéro. `puts()` ne sait pas qu'il y a quelque chose avant cette chaîne!

Cette astuce est souvent utilisée, au moins par GCC, et permet d'économiser de la mémoire. C'est proche du *string interning*.

Un autre exemple concernant ceci se trouve là: 3.2 on page 473.

1.5.4 ARM

Pour mes expérimentations avec les processeurs ARM, différents compilateurs ont été utilisés:

- Très courant dans le monde de l'embarqué: Keil Release 6/2013.
- Apple Xcode 4.6.3 IDE avec le compilateur LLVM-GCC 4.2 [28]
- GCC 4.9 (Linaro) (pour ARM64), disponible comme exécutable win32 ici http://go.yurichev.com/17325.

C'est du code ARM 32-bit qui est utilisé (également pour les modes Thumb et Thumb-2) dans tous les cas dans ce livre, sauf mention contraire.

sans optimisation Keil 6/2013 (Mode ARM)

Commençons par compiler notre exemple avec Keil:

```
armcc.exe --arm --c90 -O0 1.c
```

Le compilateur *armcc* produit un listing assembleur en syntaxe Intel, mais il dispose de macros de haut niveau liées au processeur ARM[29]. Comme il est plus important pour nous de voir les instructions « telles quelles », nous regardons le résultat compilé dans IDA.

Listing 1.24: sans optimisation Keil 6/2013 (Mode ARM) IDA

```
.text :00000000                 main
.text :00000000 10 40 2D E9     STMFD   SP!, {R4,LR}
.text :00000004 1E 0E 8F E2     ADR     R0, aHelloWorld ; "hello, world"
.text :00000008 15 19 00 EB     BL      __2printf
.text :0000000C 00 00 A0 E3     MOV     R0, #0
.text :00000010 10 80 BD E8     LDMFD   SP!, {R4,PC}

.text :000001EC 68 65 6C 6C+aHelloWorld  DCB "hello, world",0    ; DATA XREF : main+4
```

Dans l'exemple, nous voyons facilement que chaque instruction a une taille de 4 octets. En effet, nous avons compilé notre code en mode ARM, pas pour Thumb.

La toute première instruction, `STMFD SP!, {R4,LR}`[30], fonctionne comme une instruction PUSH en x86, écrivant la valeur de deux registres (R4 et LR) sur la pile.

En effet, dans le listing de la sortie du compilateur *armcc*, dans un souci de simplification, il montre l'instruction `PUSH {r4,lr}`. Mais ce n'est pas très précis. L'instruction PUSH est seulement disponible dans le mode Thumb. Donc, pour rendre les choses moins confuses, nous faisons cela dans IDA.

Cette instruction décrémente d'abord le pointeur de pile SP[32] pour qu'il pointe sur de l'espace libre pour de nouvelles entrées, ensuite elle sauve les valeurs des registres R4 et LR à cette adresse.

Cette instruction (comme l'instruction PUSH en mode Thumb) est capable de sauvegarder plusieurs valeurs de registre à la fois, ce qui peut être très utile. À propos, elle n'a pas d'équivalent en x86. On peut noter que

[28] C'est ainsi: Apple Xcode 4.6.3 utilise les composants open-source GCC comme front-end et LLVM comme générateur de code
[29] e.g. les instructions PUSH/POP manquent en mode ARM
[30] STMFD[31]
[32] stack pointer. SP/ESP/RSP dans x86/x64. SP dans ARM.

l'instruction STMFD est une généralisation de l'instruction PUSH (étendant ses fonctionnalités), puisqu'elle peut travailler avec n'importe quel registre, pas seulement avec SP. En d'autres mots, l'instruction STMFD peut être utilisée pour stocker un ensemble de registres à une adresse donnée.

L'instruction `ADR R0, aHelloWorld` ajoute ou soustrait la valeur dans le registre PC[33] à l'offset où la chaîne `hello, world` se trouve. On peut se demander comment le registre PC est utilisé ici ? C'est appelé du « code indépendant de la position »[34].

Un tel code peut être exécuté à n'importe quelle adresse en mémoire. En d'autres mots, c'est un adressage PC-relatif. L'instruction ADR prend en compte la différence entre l'adresse de cette instruction et l'adresse où est située la chaîne. Cette différence (offset) est toujours la même, peu importe à quelle adresse notre code est chargé par l'OS. C'est pourquoi tout ce dont nous avons besoin est d'ajouter l'adresse de l'instruction courante (du PC) pour obtenir l'adresse absolue en mémoire de notre chaîne C.

L'instruction `BL __2printf`[35] appelle la fonction `printf()`. Voici comment fonctionne cette instruction:

- sauve l'adresse suivant l'instruction BL (0xC) dans LR ;
- puis passe le contrôle à `printf()` en écrivant son adresse dans le registre PC.

Lorsque la fonction `printf()` termine son exécution elle doit avoir savoir où elle doit redonner le contrôle. C'est pourquoi chaque fonction passe le contrôle à l'adresse se trouvant dans le registre LR.

C'est une différence entre un processeur RISC « pur » comme ARM et un processeur CISC[36] comme x86, où l'adresse de retour est en général sauvée sur la pile. Pour aller plus loin, lire la section (1.7 on page 30) suivante.

À propos, une adresse absolue ou un offset de 32-bit ne peuvent être encodés dans l'instruction 32-bit BL car il n'y a qu'un espace de 24 bits. Comme nous devons nous en souvenir, toutes les instructions ont une taille de 4 octets (32 bits). Par conséquent, elles ne peuvent se trouver qu'à des adresses alignées dur des limites de 4 octets. Cela implique que les 2 derniers bits de l'adresse d'une instruction (qui sont toujours des bits à zéro) peuvent être omis. En résumé, nous avons 26 bits pour encoder l'offset. C'est assez pour encoder $current\_PC \pm \approx 32M$.

Ensuite, l'instruction `MOV R0, #0`[37] écrit juste 0 dans le registre R0. C'est parce que notre fonction C renvoie 0 et la valeur de retour doit être mise dans le registre R0.

La dernière instruction est `LDMFD SP!, R4,PC`[38]. Elle prend des valeurs sur la pile (ou de toute autre endroit en mémoire) afin de les sauver dans R4 et PC, et incrémente le pointeur de pile SP. Cela fonctionne ici comme POP.
N.B. La toute première instruction STMFD a sauvé la paire de registres R4 et LR sur la pile, mais R4 et PC sont *restaurés* pendant l'exécution de LDMFD.

Comme nous le savons déjà, l'adresse où chaque fonction doit redonner le contrôle est usuellement sauvée dans le registre LR. La toute première instruction sauve sa valeur sur la pile car le même registre va être utilisé par notre fonction `main()` lors de l'appel à `printf()`. A la fin de la fonction, cette valeur peut être écrite directement dans le registre PC, passant ainsi le contrôle là où notre fonction a été appelée. Comme `main()` est en général la première fonction en C/C++, le contrôle sera redonné au chargeur de l'OS ou a un point dans un CRT, ou quelque chose comme ça.

Tout cela permet d'omettre l'instruction `BX LR` à la fin de la fonction.

`DCB` est une directive du langage d'assemblage définissant un tableau d'octets ou des chaînes ASCII, proche de la directive DB dans le langage d'assemblage x86.

sans optimisation Keil 6/2013 (Mode Thumb)

Compilons le même exemple en utilisant keil en mode Thumb:

```
armcc.exe --thumb --c90 -O0 1.c
```

Nous obtenons (dans IDA):

[33] Program Counter. IP/EIP/RIP dans x86/64. PC dans ARM.
[34] Lire à ce propos la section(4.4.1 on page 559)
[35] Branch with Link
[36] Complex Instruction Set Computing
[37] Signifiant MOVe
[38] LDMFD[39] est l'instruction inverse de STMFD

Listing 1.25: sans optimisation Keil 6/2013 (Mode Thumb) + IDA

```
.text :00000000             main
.text :00000000 10 B5          PUSH    {R4,LR}
.text :00000002 C0 A0          ADR     R0, aHelloWorld ; "hello, world"
.text :00000004 06 F0 2E F9    BL      __2printf
.text :00000008 00 20          MOVS    R0, #0
.text :0000000A 10 BD          POP     {R4,PC}

.text :00000304 68 65 6C 6C+aHelloWorld  DCB "hello, world",0    ; DATA XREF : main+2
```

Nous pouvons repérer facilement les opcodes sur 2 octets (16-bit). C'est, comme déjà noté, Thumb. L'instruction BL, toutefois, consiste en deux instructions 16-bit. C'est parce qu'il est impossible de charger un offset pour la fonction `printf()` en utilisant seulement le petit espace dans un opcode 16-bit. Donc, la première instruction 16-bit charge les 10 bits supérieurs de l'offset et la seconde instruction les 11 bits inférieurs de l'offset.

Comme il a été écrit, toutes les instructions en mode Thumb ont une taille de 2 octets (ou 16 bits). Cela implique qu'il impossible pour une instruction Thumb d'être à une adresse impaire, quelle qu'elle soit. En tenant compte de cela, le dernier bit de l'adresse peut être omis lors de l'encodage des instructions.

En résumé, l'instruction Thumb BL peut encoder une adresse en $current\_PC \pm \approx 2M$.

Comme pour les autres instructions dans la fonction: PUSH et POP fonctionnent ici comme les instructions décrites STMFD/LDMFD seul le registre SP n'est pas mentionné explicitement ici. ADR fonctionne comme dans l'exemple précédent. MOVS écrit 0 dans le registre R0 afin de renvoyer zéro.

avec optimisation Xcode 4.6.3 (LLVM) (Mode ARM)

Xcode 4.6.3 sans l'option d'optimisation produit beaucoup de code redondant c'est pourquoi nous allons étudier le code généré avec optimisation, où le nombre d'instruction est aussi petit que possible, en mettant l'option `-O3` du compilateur.

Listing 1.26: avec optimisation Xcode 4.6.3 (LLVM) (Mode ARM)

```
__text :000028C4             _hello_world
__text :000028C4 80 40 2D E9   STMFD           SP!, {R7,LR}
__text :000028C8 86 06 01 E3   MOV             R0, #0x1686
__text :000028CC 0D 70 A0 E1   MOV             R7, SP
__text :000028D0 00 00 40 E3   MOVT            R0, #0
__text :000028D4 00 00 8F E0   ADD             R0, PC, R0
__text :000028D8 C3 05 00 EB   BL              _puts
__text :000028DC 00 00 A0 E3   MOV             R0, #0
__text :000028E0 80 80 BD E8   LDMFD           SP!, {R7,PC}

__cstring :00003F62 48 65 6C 6C+aHelloWorld_0  DCB "Hello world!",0
```

Les instructions STMFD et LDMFD nous sont déjà familières.

L'instruction MOV écrit simplement le nombre 0x1686 dans le registre R0. C'est l'offset pointant sur la chaîne « Hello world! ».

Le registre R7 (tel qu'il est standardisé dans [*iOS ABI Function Call Guide*, (2010)][40]) est un pointeur de frame. Voir plus loin.

L'instruction `MOVT R0, #0` (MOVe Top) écrit 0 dans les 16 bits de poids fort du registre. Le problème ici est que l'instruction générique MOV en mode ARM peut n'écrire que dans les 16 bits de poids faible du registre.

Il faut garder à l'esprit que tout les opcodes d'instruction en mode ARM sont limités en taille à 32 bits. Bien sûr, cette limitation n'est pas relative au déplacement de données entre registres. C'est pourquoi une instruction supplémentaire existe MOVT pour écrire dans les bits de la partie haute (de 16 à 31 inclus). Son usage ici, toutefois, est redondant car l'instruction `MOV R0, #0x1686` ci dessus a effacé la partie haute du registre. C'est soi-disant un défaut du compilateur.

L'instruction `ADD R0, PC, R0` ajoute la valeur dans PC à celle de R0, pour calculer l'adresse absolue de la chaîne « Hello world! ». Comme nous l'avons déjà vu, il s'agit de « code indépendant de la position » donc la correction est essentielle ici.

[40] Aussi disponible en http://go.yurichev.com/17276

L'instruction BL appelle la fonction puts() au lieu de printf().

GCC a remplacé le premier appel à printf() par un à puts(). Effectivement: printf() avec un unique argument est presque analogue à puts().

Presque, car les deux fonctions produisent le même résultat uniquement dans le cas où la chaîne ne contient pas d'identifiants de format débutant par %. Dans le cas où elle en contient, l'effet de ces deux fonctions est différent[41].

Pourquoi est-ce que le compilateur a remplacé printf() par puts() ? Probablement car puts() est plus rapide[42].

Car il envoie seulement les caractères dans sortie standard sans comparer chacun d'entre eux avec le symbole %.

Ensuite, nous voyons l'instruction familière MOV R0, #0 pour mettre le registre R0 à 0.

avec optimisation Xcode 4.6.3 (LLVM) (Mode Thumb-2)

Par défaut Xcode 4.6.3 génère du code pour Thumb-2 de cette manière:

Listing 1.27: avec optimisation Xcode 4.6.3 (LLVM) (Mode Thumb-2)

```
__text :00002B6C                    _hello_world
__text :00002B6C 80 B5          PUSH            {R7,LR}
__text :00002B6E 41 F2 D8 30    MOVW            R0, #0x13D8
__text :00002B72 6F 46          MOV             R7, SP
__text :00002B74 C0 F2 00 00    MOVT.W          R0, #0
__text :00002B78 78 44          ADD             R0, PC
__text :00002B7A 01 F0 38 EA    BLX             _puts
__text :00002B7E 00 20          MOVS            R0, #0
__text :00002B80 80 BD          POP             {R7,PC}

...

__cstring :00003E70 48 65 6C 6C 6F 20+aHelloWorld  DCB "Hello world!",0xA,0
```

Les instructions BL et BLX en mode Thumb, comme on s'en souvient, sont encodées comme une paire d'instructions 16 bits. En Thumb-2 ces opcodes *substituts* sont étendus de telle sorte que les nouvelles instructions puissent être encodées comme des instructions 32-bit.

C'est évident en considérant que les opcodes des instructions Thumb-2 commencent toujours avec 0xFx ou 0xEx.

Mais dans le listing d'IDA les octets d'opcodes sont échangés car pour le processeur ARM les instructions sont encodées comme ceci: dernier octet en premier et ensuite le premier (pour les modes Thumb et Thumb-2) ou pour les instructions en mode ARM le quatrième octet vient en premier, ensuite le troisième, puis le second et enfin le premier (à cause des différents endianness).

C'est ainsi que les octets se trouvent dans le listing d'IDA:

- pour les modes ARM et ARM64: 4-3-2-1;
- pour le mode Thumb: 2-1;
- pour les paires d'instructions 16-bit en mode Thumb-2: 2-1-4-3.

Donc, comme on peut le voir, les instructions MOVW, MOVT.W et BLX commencent par 0xFx.

Une des instructions Thumb-2 est MOVW R0, #0x13D8 —elle stocke une valeur 16-bit dans la partie inférieure du registre R0, effaçant les bits supérieurs.

Aussi, MOVT.W R0, #0 fonctionne comme MOVT de l'exemple précédent mais il fonctionne en Thumb-2.

Parmi les autres différences, l'instruction BLX est utilisée dans ce cas à à la place de BL.

La différence est que, en plus de sauver RA[43] dans le registre LR et de passer le contrôle à la fonction puts(), le processeur change du mode Thumb/Thumb-2 au mode ARM (ou inversement).

[41] Il est à noter que puts() ne nécessite pas un '\n' symbole de retour à la ligne à la fin de la chaîne, donc nous ne le voyons pas ici.

[42] ciselant.de/projects/gcc_printf/gcc_printf.html

[43] Adresse de retour

Cette instruction est placée ici, car l'instruction à laquelle est passée le contrôle ressemble à (c'est encodé en mode ARM):

```
__symbolstub1 :00003FEC _puts           ; CODE XREF : _hello_world+E
__symbolstub1 :00003FEC 44 F0 9F E5     LDR  PC, =__imp__puts
```

Il s'agit principalement d'un saut à l'endroit où l'adresse de `puts()` est écrit dans la section import.

Mais alors, le lecteur attentif pourrait demander: pourquoi ne pas appeler `puts()` depuis l'endroit dans le code où on en a besoin ?

Parce que ce n'est pas très efficace en terme d'espace.

Presque tous les programmes utilisent des bibliothèques dynamiques externes (comme les DLL sous Windows, les .so sous *NIX ou les .dylib sous Mac OS X). Les bibliothèques dynamiques contiennent les bibliothèques fréquemment utilisées, incluant la fonction C standard `puts()`.

Dans un fichier binaire exécutable (Windows PE .exe, ELF ou Mach-O) se trouve une section d'import. Il s'agit d'une liste des symboles (fonctions ou variables globales) importées depuis des modules externes avec le nom des modules eux-même.

Le chargeur de l'OS charge tous les modules dont il a besoin, tout en énumérant les symboles d'import dans le module primaire, il détermine l'adresse correcte de chaque symbole.

Dans notre cas, *__imp__puts* est une variable 32-bit utilisée par le chargeur de l'OS pour sauver l'adresse correcte d'une fonction dans une bibliothèque externe. Ensuite l'instruction LDR lit la valeur 32-bit depuis cette variable et l'écrit dans le registre PC, lui passant le contrôle.

Donc, pour réduire le temps dont le chargeur de l'OS à besoin pour réaliser cette procédure, c'est une bonne idée d'écrire l'adresse de chaque symbole une seule fois, à une place dédiée.

À côté de ça, comme nous l'avons déjà compris, il est impossible de charger une valeur 32-bit dans un registre en utilisant seulement une instruction sans un accès mémoire.

Donc, la solution optimale est d'allouer une fonction séparée fonctionnant en mode ARM avec le seul but de passer le contrôle à la bibliothèque dynamique et ensuite de sauter à cette petite fonction d'une instruction (ainsi appelée fonction thunk) depuis le code Thumb.

À propos, dans l'exemple précédent (compilé en mode ARM), le contrôle est passé par BL à la même fonction thunk. Le mode du processeur, toutefois, n'est pas échangé (d'où l'absence d'un « X » dans le mnémonique de l'instruction).

Plus à propos des fonctions thunk

Les fonctions thunk sont difficile à comprendre, apparemment, à cause d'un mauvais nom. La manière la plus simple est de les voir comme des adaptateurs ou des convertisseurs d'un type jack à un autre. Par exemple, un adaptateur permettant l'insertion d'un cordon électrique britannique sur une prise murale américaine, ou vice-versa. Les fonctions thunk sont parfois appelées *wrappers*.

Voici quelques autres descriptions de ces fonctions:

> "Un morceau de code qui fournit une adresse:", d'après P. Z. Ingerman, qui inventa thunk en 1961 comme un moyen de lier les paramètres réels à leur définition formelle dans les appels de procédures en Algol-60. Si une procédure est appelée avec une expression à la place d'un paramètre formel, le compilateur génère un thunk qui calcule l'expression et laisse l'adresse du résultat dans une place standard.
>
> ...
>
> Microsoft et IBM ont tous les deux défini, dans systèmes basés sur Intel, un "environnement 16-bit" (avec leurs horribles registres de segment et la limite des adresses à 64K) et un "environnement 32-bit" (avec un adressage linéaire et une gestion semi-réelle de la mémoire). Les deux environnements peuvent fonctionner sur le même ordinateur et OS (grâce à ce qui est appelé, dans le monde Microsoft, WOW qui signifie Windows dans Windows). MS et IBM ont tous deux décidé que le procédé de passer de 16-bit à 32-bit et vice-versa est appelé un "thunk"; pour Window 95, il y a même un outil, THUNK.EXE, appelé un "compilateur thunk".

(The Jargon File)

Nous pouvons trouver un autre exemple dans la bibliothèque LAPCAK—un “Linear Algebra PACKage” écrit en FORTRAN. Les développeurs C/C++ veulent aussi utiliser LAPACK, mais c’est un non-sens de la récrire en C/C++ et de maintenir plusieurs versions. Donc, il y a des petites fonctions que l’on peut invoquer depuis un environnement C/C++, qui font, à leur tour, des appels aux fonctions FORTRAN, et qui font presque tout le reste:

```
double Blas_Dot_Prod(const LaVectorDouble &dx, const LaVectorDouble &dy)
{
    assert(dx.size()==dy.size());
    integer n = dx.size();
    integer incx = dx.inc(), incy = dy.inc();

    return F77NAME(ddot)(&n, &dx(0), &incx, &dy(0), &incy);
}
```

Donc, ce genre de fonctions est appelé “wrappers”.

ARM64

GCC

Compilons l’exemple en utilisant GCC 4.8.1 en ARM64:

Listing 1.28: GCC 4.8.1 sans optimisation + objdump

```
0000000000400590 <main> :
  400590:       a9bf7bfd        stp     x29, x30, [sp,#-16]!
  400594:       910003fd        mov     x29, sp
  400598:       90000000        adrp    x0, 400000 <_init-0x3b8>
  40059c :       91192000        add     x0, x0, #0x648
  4005a0 :       97ffffa0        bl      400420 <puts@plt>
  4005a4 :       52800000        mov     w0, #0x0                        // #0
  4005a8 :       a8c17bfd        ldp     x29, x30, [sp],#16
  4005ac :       d65f03c0        ret

...

Contents of section .rodata :
 400640 01000200 00000000 48656c6c 6f210a00  ........Hello!..
```

Il n’y a pas de mode Thumb ou Thumb-2 en ARM64, seulement en ARM, donc il n’y a que des instructions 32-bit. Le nombre de registres a doublé: **??** on page ??. Les registres 64-bit ont le préfixe `X-`, tandis que leurs partie 32-bit basse—`W-`.

L’instruction STP (*Store Pair* stocke une paire) sauve deux registres sur la pile simultanément: `X29` et `X30`.

Bien sûr, cette instruction peut sauvegarder cette paire à n’importe quelle endroit en mémoire, mais le registre SP est spécifié ici, donc la paire est sauvé sur le pile.

Les registres ARM64 font 64-bit, chacun a une taille de 8 octets, donc il faut 16 octets pour sauver deux registres.

Le point d’exclamation (“!”) après l’opérande signifie que 16 octets doivent d’abord être soustrait de SP, et ensuite les valeurs de la paire de registres peuvent être écrites sur la pile. Ceci est appelé le *pre-index*. À propos de la différence entre *post-index* et *pre-index* lisez ceci: 1.32.2 on page 442.

Dans la gamme plus connue du x86, la première instruction est analogue à la paire `PUSH X29` et `PUSH X30`. En ARM64, `X29` est utilisé comme FP[44] et `X30` comme LR, c’est pourquoi ils sont sauvegardés dans le prologue de la fonction et remis dans l’épilogue.

La seconde instruction copie SP dans `X29` (ou FP). Cela sert à préparer la pile de la fonction.

Les instructions `ADRP` et `ADD` sont utilisées pour remplir l’adresse de la chaîne « Hello! » dans le registre `X0`, car le premier argument de la fonction est passé dans ce registre. Il n’y a pas d’instruction, quelqu’elle soit, en ARM qui puisse stocker un nombre large dans un registre (car la longueur des instructions est limitée à 4 octets, cf: 1.32.3 on page 443). Plusieurs instructions doivent donc être utilisées. La première

[44] Frame Pointer

instruction (ADRP) écrit l'adresse de la page de 4KiB, où se trouve la chaîne, dans X0, et la seconde (ADD) ajoute simplement le reste de l'adresse. Plus d'information ici: 1.32.4 on page 445.

0x400000 + 0x648 = 0x400648, et nous voyons notre chaîne C « Hello! » dans le .rodata segment des données à cette adresse.

puts() est appelée après en utilisant l'instruction BL. Cela a déjà été discuté: 1.5.4 on page 22.

MOV écrit 0 dans W0. W0 est la partie basse 32 bits du registre 64-bit X0 :

Partie 32 bits haute	Partie 32 bits basse
X0	
	W0

Le résultat de la fonction est retourné via X0 et main renvoie 0, donc c'est ainsi que la valeur de retour est préparée. Mais pourquoi utiliser la partie 32-bit?

Parce que le type de donnée *int* en ARM64, tout comme en x86-64, est toujours 32-bit, pour une meilleure compatibilité.

Donc si la fonction renvoie un *int* 32-bit, seul les 32 premiers bits du registre X0 doivent être remplis.

Pour vérifier ceci, changeons un peu cet exemple et recompilons-le. Maintenant, main() renvoie une valeur sur 64-bit:

Listing 1.29: main() renvoie une valeur de type uint64_t type

```
#include <stdio.h>
#include <stdint.h>

uint64_t main()
{
        printf ("Hello!\n");
        return 0;
}
```

Le résultat est le même, mais c'est à quoi ressemble MOV à cette ligne maintenant:

Listing 1.30: GCC 4.8.1 sans optimisation + objdump

```
  4005a4 :       d2800000        mov     x0, #0x0        // #0
```

LDP (*Load Pair*) remet les registres X29 et X30.

Il n'y a pas de point d'exclamation après l'instruction: celui signifie que les valeurs sont d'abord chargées depuis la pile, et ensuite SP est incrémenté de 16. Cela est appelé *post-index*.

Une nouvelle instruction est apparue en ARM64: RET. Elle fonctionne comme BX LR, un *hint* bit particulier est ajouté, qui informe le CPU qu'il s'agit d'un retour de fonction, et pas d'une autre instruction de saut, et il peut l'exécuter de manière plus optimale.

À cause de la simplicité de la fonction, GCC avec l'option d'optimisation génère le même code.

1.5.5 MIPS

Un mot à propos du « pointeur global »

Un concept MIPS important est le « pointeur global ». Comme nous le savons déjà, chaque instruction MIPS a une taille de 32-bit, donc il est impossible d'avoir une adresse 32-bit dans une instruction: il faut pour cela utiliser une paire. (comme le fait GCC dans notre exemple pour le chargement de l'adresse de la chaîne de texte). Il est possible, toutefois, de charger des données depuis une adresse dans l'interval $register - 32768 ... register + 32767$ en utilisant une seule instruction (car un offset signé de 16 bits peut être encodé dans une seule instruction). Nous pouvons alors allouer un registre dans ce but et dédier un bloc de 64KiB pour les données les plus utilisées. Ce registre dédié est appelé un « pointeur global » et il pointe au milieu du bloc de 64 KiB. Ce bloc contient en général les variables globales et les adresses des fonctions importées, comme printf(), car les développeurs de GCC ont décidé qu'obtenir l'adresse d'une fonction devait se faire en une instruction au lieu de deux. Dans un fichier ELF ce bloc de 64KiB se trouve en partie dans une section .sbss (« small BSS[45] ») pour les données non initialisées et .sdata (« small data ») pour celles initialisées. Cela implique que le programmeur peut choisir quelle donnée il/elle souhaite rendre

[45] Block Started by Symbol

accessible rapidement et doit les stocker dans .sdata/.sbss. Certains programmeurs old-school peuvent se souvenir du modèle de mémoire MS-DOS 7.6 on page 644 ou des gestionnaires de mémoire MS-DOS comme XMS/EMS où toute la mémoire était divisée en bloc de 64KiB.

Ce concept n'est pas restreint à MIPS. Au moins les PowerPC utilisent aussi cette technique.

GCC avec optimisation

Considérons l'exemple suivant, qui illustre le concept de « pointeur global ».

Listing 1.31: GCC 4.4.5 avec optimisation (résultat en sortie de l'assembleur)

```
1  $LC0 :
2  ; \000 est l'octet à zéro en base octale:
3          .ascii  "Hello, world!\012\000"
4  main :
5  ; prologue de la fonction.
6  ; définir GP:
7          lui     $28,%hi(__gnu_local_gp)
8          addiu   $sp,$sp,-32
9          addiu   $28,$28,%lo(__gnu_local_gp)
10 ; sauver RA sur la pile locale :
11         sw      $31,28($sp)
12 ; charger l'adresse de la fonction puts() dans $25 depuis GP :
13         lw      $25,%call16(puts)($28)
14 ; charger l'adresse de la chaîne de texte dans $4 ($a0):
15         lui     $4,%hi($LC0)
16 ; sauter à puts(), en sauvant l'adresse de retour dans le register link:
17         jalr    $25
18         addiu   $4,$4,%lo($LC0) ; slot de retard de branchement
19 ; restaurer RA :
20         lw      $31,28($sp)
21 ; copier 0 depuis $zero dans $v0 :
22         move    $2,$0
23 ; retourner en sautant à la valeur dans RA:
24         j       $31
25 ; épilogue de la fonction:
26         addiu   $sp,$sp,32 ; slot de retard de branchement + libérer la pile locale
```

Comme on le voit, le registre $GP est défini dans le prologue de la fonction pour pointer au milieu de ce bloc. Le registre RA est sauvé sur la pile locale. `puts()` est utilisé ici au lieu de `printf()`. L'adresse de la fonction `puts()` est chargée dans $25 en utilisant l'instruction LW (« Load Word »). Ensuite l'adresse de la chaîne de texte est chargée dans $4 avec la paire d'instructions LUI ((« Load Upper Immediate ») et ADDIU (« Add Immediate Unsigned Word »). LUI défini les 16 bits de poids fort du registre (d'où le mot « upper » dans le nom de l'instruction) et ADDIU ajoute les 16 bits de poids faible de l'adresse.

ADDIU suit JALR (vous n'avez pas déjà oublié le *slot de délai de branchement* ?). Le registre $4 est aussi appelé $A0, qui est utilisé pour passer le premier argument d'une fonction [46].

JALR (« Jump and Link Register ») saute à l'adresse stockée dans le registre $25 (adresse de `puts()`) en sauvant l'adresse de la prochaine instruction (LW) dans RA. C'est très similaire à ARM. Oh, encore une chose importante, l'adresse sauvée dans RA n'est pas l'adresse de l'instruction suivante (car c'est celle du *slot de délai* et elle est exécutée avant l'instruction de saut), mais l'adresse de l'instruction après la suivante (après le *slot de délai*). Par conséquent, $PC + 8$ est écrit dans RA pendant l'exécution de JALR, dans notre cas, c'est l'adresse de l'instruction LW après ADDIU.

LW (« Load Word ») à la ligne 20 restaure RA depuis la pile locale (cette instruction fait partie de l'épilogue de la fonction).

MOVE à la ligne 22 copie la valeur du registre $0 ($ZERO) dans $2 ($V0).

MIPS a un registre *constant*, qui contient toujours zéro. Apparemment, les développeurs de MIPS avaient à l'esprit que zéro est la constante la plus utilisée en programmation, utilisons donc le registre $0 à chaque fois que zéro est requis.

Un autre fait intéressant est qu'il manque en MIPS une instruction qui transfère des données entre des registres. En fait, `MOVE DST, SRC` est `ADD DST, SRC, $ZERO` ($DST = SRC + 0$), qui fait la même chose. Manifestement, les développeurs de MIPS voulaient une table des opcodes compacte. Cela ne signifie

[46] La table des registres MIPS est disponible en appendice **??** on page ??

pas qu'il y a une addition à chaque instruction MOVE. Très probablement, le CPU optimise ces pseudo-instructions et l'UAL[47] n'est jamais utilisé.

J à la ligne 24 saute à l'adresse dans RA, qui effectue effectivement un retour de la fonction. ADDIU après J est en fait exécutée avant J (vous vous rappeler du *slot de délai de branchement* ?) et fait partie de l'épilogue de la fonction. Voici un listing généré par IDA. Chaque registre a son propre pseudo nom:

Listing 1.32: GCC 4.4.5 avec optimisation (IDA)

```
1  .text :00000000 main :
2  .text :00000000
3  .text :00000000 var_10          = -0x10
4  .text :00000000 var_4           = -4
5  .text :00000000
6  ; prologue de la fonction.
7  ; définir GP:
8  .text :00000000                 lui     $gp, (__gnu_local_gp >> 16)
9  .text :00000004                 addiu   $sp, -0x20
10 .text :00000008                 la      $gp, (__gnu_local_gp & 0xFFFF)
11 ; sauver RA sur la pile locale :
12 .text :0000000C                 sw      $ra, 0x20+var_4($sp)
13 ; sauver GP sur la pile locale :
14 ; pour une raison, cette instruction manque dans la sortie en assembleur de GCC :
15 .text :00000010                 sw      $gp, 0x20+var_10($sp)
16 ; charger l'adresse de la fonction puts() dans $9 depuis GP :
17 .text :00000014                 lw      $t9, (puts & 0xFFFF)($gp)
18 ; générer l'adresse de la chaîne de texte dans $a0:
19 .text :00000018                 lui     $a0, ($LC0 >> 16)  # "Hello, world!"
20 ; sauter à puts(), en sauvant l'adresse de retour dans le register link:
21 .text :0000001C                 jalr    $t9
22 .text :00000020                 la      $a0, ($LC0 & 0xFFFF)  # "Hello, world!"
23 ; restaurer RA :
24 .text :00000024                 lw      $ra, 0x20+var_4($sp)
25 ; copier 0 depuis $zero dans $v0 :
26 .text :00000028                 move    $v0, $zero
27 ; retourner en sautant à la valeur dans RA:
28 .text :0000002C                 jr      $ra
29 ; épilogue de la fonction:
30 .text :00000030                 addiu   $sp, 0x20
```

L'instruction à la ligne 15 sauve la valeur de GP sur la pile locale, et cette instruction manque mystérieusement dans le listing de sortie de GCC, peut-être une erreur de GCC [48]. La valeur de GP doit effectivement être sauvée, car chaque fonction utilise sa propre fenêtre de 64KiB. Le registre contenant l'adresse de `puts()` est appelé $T9, car les registres préfixés avec T- sont appelés « temporaires » et leur contenu ne doit pas être préservé.

GCC sans optimisation

GCC sans optimisation est plus verbeux.

Listing 1.33: GCC 4.4.5 sans optimisation (résultat en sortie de l'assembleur)

```
1  $LC0 :
2          .ascii  "Hello, world!\012\000"
3  main :
4  ; prologue de la fonction.
5  ; sauver RA ($31) et FP sur la pile :
6          addiu   $sp,$sp,-32
7          sw      $31,28($sp)
8          sw      $fp,24($sp)
9  ; définir le pointeur de pile FP (stack frame pointer) :
10         move    $fp,$sp
11 ; définir GP:
12         lui     $28,%hi(__gnu_local_gp)
13         addiu   $28,$28,%lo(__gnu_local_gp)
14 ; charger l'adresse de la chaîne de texte:
```

[47] Unité arithmétique et logique

[48] Apparemment, les fonctions générant les listings ne sont pas si critique pour les utilisateurs de GCC, donc des erreurs peuvent toujours subsister.

```
        lui     $2,%hi($LC0)
        addiu   $4,$2,%lo($LC0)
; charger l'adresse de puts() en utilisant GP :
        lw      $2,%call16(puts)($28)
        nop
; appeler puts() :
        move    $25,$2
        jalr    $25
        nop  ; slot de retard de branchement

; restaurer GP depuis la pile locale :
        lw      $28,16($fp)
; mettre le registre $2 ($V0) à zéro:
        move    $2,$0
; épilogue de la fonction.
; restaurer SP :
        move    $sp,$fp
; restaurer RA :
        lw      $31,28($sp)
; restaurer FP :
        lw      $fp,24($sp)
        addiu   $sp,$sp,32
; sauter en RA :
        j       $31
        nop  ; slot de délai de branchement
```

Nous voyons ici que le registre FP est utilisé comme un pointeur sur la pile. Nous voyons aussi 3 NOPs. Le second et le troisième suivent une instruction de branchement. Peut-être que le compilateur GCC ajoute toujours des NOPs (à cause du *slot de retard de branchement*) après les instructions de branchement, et, si l'optimisation est demandée, il essaye alors de les éliminer. Donc, dans ce cas, ils sont laissés en place.

Voici le listing IDA :

Listing 1.34: GCC 4.4.5 sans optimisation (IDA)

```
.text :00000000 main :
.text :00000000
.text :00000000 var_10          = -0x10
.text :00000000 var_8           = -8
.text :00000000 var_4           = -4
.text :00000000
; prologue de la fonction.
; sauver RA et FP sur la pile :
.text :00000000                 addiu   $sp, -0x20
.text :00000004                 sw      $ra, 0x20+var_4($sp)
.text :00000008                 sw      $fp, 0x20+var_8($sp)
; définir FP (stack frame pointer) :
.text :0000000C                 move    $fp, $sp
; définir GP:
.text :00000010                 la      $gp, __gnu_local_gp
.text :00000018                 sw      $gp, 0x20+var_10($sp)
; charger l'adresse de la chaîne de texte:
.text :0000001C                 lui     $v0, (aHelloWorld >> 16)  # "Hello, world!"
.text :00000020                 addiu   $a0, $v0, (aHelloWorld & 0xFFFF)  # "Hello, world!"
; charger l'adresse de puts() en utilisant GP :
.text :00000024                 lw      $v0, (puts & 0xFFFF)($gp)
.text :00000028                 or      $at, $zero ; NOP
; appeler puts() :
.text :0000002C                 move    $t9, $v0
.text :00000030                 jalr    $t9
.text :00000034                 or      $at, $zero ; NOP
; restaurer GP depuis la pile locale :
.text :00000038                 lw      $gp, 0x20+var_10($fp)
; mettre le registre $2 ($V0) à zéro:
.text :0000003C                 move    $v0, $zero
; épilogue de la fonction.
; restaurer SP :
.text :00000040                 move    $sp, $fp
; restaurer RA :
.text :00000044                 lw      $ra, 0x20+var_4($sp)
; restaurer FP :
```

```
37 .text :00000048                lw      $fp, 0x20+var_8($sp)
38 .text :0000004C                addiu   $sp, 0x20
39 ; sauter en RA :
40 .text :00000050                jr      $ra
41 .text :00000054                or      $at, $zero ; NOP
```

Intéressant, IDA a reconnu les instructions LUI/ADDIU et les a agrégées en une pseudo instruction LA (« Load Address ») à la ligne 15. Nous pouvons voir que cette pseudo instruction a une taille de 8 octets! C'est une pseudo instruction (ou *macro*) car ce n'est pas une instruction MIPS réelle, mais plutôt un nom pratique pour une paire d'instructions.

Une autre chose est qu'IDA ne reconnaît pas les instructions NOP, donc ici elles se trouvent aux lignes 22, 26 et 41. C'est OR $AT, $ZERO. Essentiellement, cette instruction applique l'opération OR au contenu du registre $AT avec zéro, ce qui, bien sûr, est une instruction sans effet. MIPS, comme beaucoup d'autres ISAs, n'a pas une instruction NOP.

Rôle de la pile dans cet exemple

L'adresse de la chaîne de texte est passée dans le registre. Pourquoi définir une pile locale quand même? La raison de cela est que la valeur des registres RA et GP doit être sauvée quelque part (car `printf()` est appelée), et que la pile locale est utilisée pour cela. Si cela avait été une fonction leaf, il aurait été possible de se passer du prologue et de l'épilogue de la fonction, par exemple: 1.4.3 on page 8.

GCC avec optimisation : chargeons-le dans GDB

Listing 1.35: extrait d'une session GDB

```
root@debian-mips :~# gcc hw.c -O3 -o hw

root@debian-mips :~# gdb hw
GNU gdb (GDB) 7.0.1-debian
...
Reading symbols from /root/hw...(no debugging symbols found)...done.
(gdb) b main
Breakpoint 1 at 0x400654
(gdb) run
Starting program : /root/hw

Breakpoint 1, 0x00400654 in main ()
(gdb) set step-mode on
(gdb) disas
Dump of assembler code for function main :
0x00400640 <main+0> :    lui     gp,0x42
0x00400644 <main+4> :    addiu   sp,sp,-32
0x00400648 <main+8> :    addiu   gp,gp,-30624
0x0040064c <main+12> :   sw      ra,28(sp)
0x00400650 <main+16> :   sw      gp,16(sp)
0x00400654 <main+20> :   lw      t9,-32716(gp)
0x00400658 <main+24> :   lui     a0,0x40
0x0040065c <main+28> :   jalr    t9
0x00400660 <main+32> :   addiu   a0,a0,2080
0x00400664 <main+36> :   lw      ra,28(sp)
0x00400668 <main+40> :   move    v0,zero
0x0040066c <main+44> :   jr      ra
0x00400670 <main+48> :   addiu   sp,sp,32
End of assembler dump.
(gdb) s
0x00400658 in main ()
(gdb) s
0x0040065c in main ()
(gdb) s
0x2ab2de60 in printf () from /lib/libc.so.6
(gdb) x/s $a0
0x400820 :        "hello, world"
(gdb)
```

1.5.6 Conclusion

La différence principale entre le code x86/ARM et x64/ARM64 est que le pointeur sur la chaîne a une taille de 64 bits. Le fait est que les CPUs modernes sont maintenant 64-bit à cause le la baisse du coût de la mémoire et du grand besoin de cette dernière par les applications modernes. Nous pouvons ajouter bien plus de mémoire à nos ordinateurs que les pointeurs 32-bit ne peuvent en adresser. Ainsi, tous les pointeurs sont maintenant 64-bit.

1.5.7 Exercices

- http://challenges.re/48
- http://challenges.re/49

1.6 Fonction prologue et épilogue

Un prologue de fonction est une séquence particulière d'instructions située au début d'une fonction. Il ressemble souvent à ce morceau de code:

```
    push    ebp
    mov     ebp, esp
    sub     esp, X
```

Ce que ces instructions font: sauvent la valeur du registre EBP dans la pile (push ebp), sauvent la valeur actuelle du registre ESP dans le registre EBP (mov ebp, esp) et enfin allouent de la mémoire dans la pile pour les variables locales de la fonction (sub esp, X).

La valeur du registre EBP reste la même durant la période où la fonction s'exécute et est utilisée pour accéder aux variables locales et aux arguments de la fonction.

Le registre ESP peut aussi être utilisé pour accéder aux variables locales et aux arguments de la fonction, cependant cette approche n'est pas pratique car sa valeur est susceptible de changer au cours de l'exécution de cette fonction.

L'épilogue de fonction libère la mémoire allouée dans la pile (mov esp, ebp), restaure l'ancienne valeur de EBP précédemment sauvegardée dans la pile (pop ebp) puis rend l'exécution à l'appelant (ret 0).

```
    mov     esp, ebp
    pop     ebp
    ret     0
```

Les prologues et épilogues de fonction sont généralement détectés par les désassembleurs pour déterminer où une fonction commence et où elle se termine.

1.6.1 Récursivité

Les prologues et épilogues de fonction peuvent affecter négativement les performances de la récursion.

Plus d'information sur la récursivité dans ce livre: 3.4.3 on page 485.

1.7 Pile

La pile est une des structures de données les plus fondamentales en informatique [49]. AKA[50] LIFO[51].

Techniquement, il s'agit d'un bloc de mémoire situé dans l'espace d'adressage d'un processus et qui est utilisé par le registre ESP en x86, RSP en x64 ou par le registre SP en ARM comme un pointeur dans ce bloc mémoire.

[49] wikipedia.org/wiki/Call_stack
[50] Aussi connu sous le nom de
[51] Dernier entré, premier sorti

Les instructions d'accès à la pile sont PUSH et POP (en x86 ainsi qu'en ARM Thumb-mode). PUSH soustrait à ESP/RSP/SP 4 en mode 32-bit (ou 8 en mode 64-bit) et écrit ensuite le contenu de l'opérande associé à l'adresse mémoire pointée par ESP/RSP/SP.

POP est l'opération inverse: elle récupère la donnée depuis l'adresse mémoire pointée par SP, l'écrit dans l'opérande associé (souvent un registre) puis ajoute 4 (ou 8) au pointeur de pile.

Après une allocation sur la pile, le pointeur de pile pointe sur le bas de la pile. PUSH décrémente le pointeur de pile et POP l'incrémente.

Le bas de la pile représente en réalité le début de la mémoire allouée pour le bloc de pile. Cela semble étrange, mais c'est comme ça.

ARM supporte à la fois les piles ascendantes et descendantes.

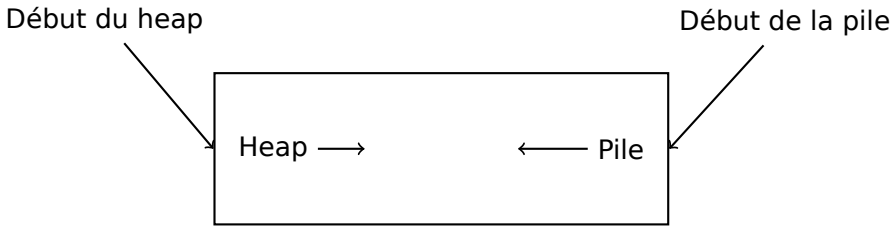
Par exemple les instructions STMFD/LDMFD, STMED[52]/LDMED[53] sont utilisées pour gérer les piles descendantes (qui grandissent vers le bas en commençant avec une adresse haute et évoluent vers une plus basse).

Les instructions STMFA[54]/LDMFA[55], STMEA[56]/LDMEA[57] sont utilisées pour gérer les piles montantes (qui grandissent vers les adresses hautes de l'espace d'adressage, en commençant avec une adresse située en bas de l'espace d'adressage).

1.7.1 Pourquoi la pile grandit en descendant ?

Intuitivement, on pourrait penser que la pile grandit vers le haut, i.e. vers des adresses plus élevées, comme n'importe qu'elle autre structure de données.

La raison pour laquelle la pile grandit vers le bas est probablement historique. Dans le passé, les ordinateurs étaient énormes et occupaient des pièces entières, il était facile de diviser la mémoire en deux parties, une pour le heap et une pour la pile. Évidemment, on ignorait quelle serait la taille du heap et de la pile durant l'exécution du programme, donc cette solution était la plus simple possible.



Dans [D. M. Ritchie and K. Thompson, *The UNIX Time Sharing System*, (1974)][58]on peut lire:

> The user-core part of an image is divided into three logical segments. The program text segment begins at location 0 in the virtual address space. During execution, this segment is write-protected and a single copy of it is shared among all processes executing the same program. At the first 8K byte boundary above the program text segment in the virtual address space begins a nonshared, writable data segment, the size of which may be extended by a system call. Starting at the highest address in the virtual address space is a pile segment, which automatically grows downward as the hardware's pile pointer fluctuates.

Cela nous rappelle comment certains étudiants prennent des notes pour deux cours différents dans un seul et même cahier en prenant un cours d'un côté du cahier, et l'autre cours de l'autre côté. Les notes de cours finissent par se rencontrer à un moment dans le cahier quand il n'y a plus de place.

[52] Store Multiple Empty Descending (instruction ARM)
[53] Load Multiple Empty Descending (instruction ARM)
[54] Store Multiple Full Ascending (instruction ARM)
[55] Load Multiple Full Ascending (instruction ARM)
[56] Store Multiple Empty Ascending (instruction ARM)
[57] Load Multiple Empty Ascending (instruction ARM)
[58] Aussi disponible en http://go.yurichev.com/17270

1.7.2 Quel est le rôle de la pile ?

Sauvegarder l'adresse de retour de la fonction

x86

Lorsque l'on appelle une fonction avec une instruction CALL, l'adresse du point exactement après cette dernière est sauvegardée sur la pile et un saut inconditionnel à l'adresse de l'opérande CALL est exécuté.

L'instruction CALL est équivalente à la paire d'instructions
`PUSH address_after_call / JMP operand`.

RET va chercher une valeur sur la pile et y saute —ce qui est équivalent à la paire d'instructions `POP tmp / JMP tmp`.

Déborder de la pile est très facile. Il suffit de lancer une récursion éternelle:

```
void f()
{
        f();
};
```

MSVC 2008 signale le problème:

```
c :\tmp6>cl ss.cpp /Fass.asm
Microsoft (R) 32-bit C/C++ Optimizing Compiler Version 15.00.21022.08 for 80x86
Copyright (C) Microsoft Corporation.  All rights reserved.

ss.cpp
c :\tmp6\ss.cpp(4)  : warning C4717 : 'f'  : recursive on all control paths, function will ⤶
    ↳ cause runtime stack overflow
```

...mais génère tout de même le code correspondant:

```
?f@@YAXXZ PROC                  ; f
; File c :\tmp6\ss.cpp
; Line 2
        push    ebp
        mov     ebp, esp
; Line 3
        call    ?f@@YAXXZ       ; f
; Line 4
        pop     ebp
        ret     0
?f@@YAXXZ ENDP                  ; f
```

...Si nous utilisons l'option d'optimisation du compilateur (option /0x) le code optimisé ne va pas déborder de la pile et au lieu de cela va fonctionner *correctemment*[59] :

```
?f@@YAXXZ PROC                  ; f
; File c :\tmp6\ss.cpp
; Line 2
$LL3@f :
; Line 3
        jmp     SHORT $LL3@f
?f@@YAXXZ ENDP                  ; f
```

GCC 4.4.1 génère un code similaire dans les deux cas, sans, toutefois émettre d'avertissement à propos de ce problème.

ARM

Les programmes ARM utilisent également la pile pour sauver les adresses de retour, mais différemment. Comme mentionné dans « Hello, world! » (1.5.4 on page 19), RA est sauvegardé dans LR (link register).

[59] ironique ici

Si l'on a toutefois besoin d'appeler une autre fonction et d'utiliser le registre LR une fois de plus, sa valeur doit être sauvegardée. Usuellement, cela se fait dans le prologue de la fonction.

Souvent, nous voyons des instructions comme PUSH R4-R7,LR en même temps que cette instruction dans l'épilogue POP R4-R7,PC—ces registres qui sont utilisés dans la fonction sont sauvegardés sur la pile, LR inclus.

Néanmoins, si une fonction n'appelle jamais d'autre fonction, dans la terminologie RISC elle est appelée *fonction leaf*[60]. Ceci a comme conséquence que les fonctions leaf ne sauvegardent pas le registre LR (car elles ne le modifient pas). Si une telle fonction est petite et utilise un petit nombre de registres, elle peut ne pas utiliser du tout la pile. Ainsi, il est possible d'appeler des fonctions leaf sans utiliser la pile. Ce qui peut être plus rapide sur des vieilles machines x86 car la mémoire externe n'est pas utilisée pour la pile [61]. Cela peut être utile pour des situations où la mémoire pour la pile n'est pas encore allouée ou disponible.

Quelques exemples de fonctions leaf: 1.10.3 on page 104, 1.10.3 on page 104, 1.276 on page 317, 1.292 on page 335, 1.22.5 on page 335, 1.184 on page 210, 1.182 on page 208, 1.201 on page 227.

Passage des arguments d'une fonction

Le moyen le plus utilisé pour passer des arguments en x86 est appelé « cdecl » :

```
push arg3
push arg2
push arg1
call f
add esp, 12 ; 4*3=12
```

La fonction appelée reçoit ses arguments par la pile.

Voici donc comment sont stockés les arguments sur la pile avant l'exécution de la première instruction de la fonction `f()` :

ESP	return address
ESP+4	argument#1, marqué dans IDA comme `arg_0`
ESP+8	argument#2, marqué dans IDA comme `arg_4`
ESP+0xC	argument#3, marqué dans IDA comme `arg_8`
...	...

Pour plus d'information sur les conventions d'appel, voir cette section (4.1 on page 544).

À propos, la fonction appelée n'a aucune d'information sur le nombre d'arguments qui ont été passés. Les fonctions C avec un nombre variable d'arguments (comme `printf()`) déterminent leur nombre en utilisant les spécificateurs de la chaîne de format (qui commencent pas le symbole %).

Si nous écrivons quelque comme:

```
printf("%d %d %d", 1234);
```

`printf()` va afficher 1234, et deux autres nombres aléatoires[62], qui sont situés à côté dans la pile.

C'est pourquoi la façon dont la fonction `main()` est déclarée n'est pas très importante: comme `main()`, `main(int argc, char *argv[])` ou `main(int argc, char *argv[], char *envp[])`.

En fait, le code-CRT appelle `main()`, schématiquement, de cette façon:

```
push envp
push argv
push argc
call main
...
```

Si vous déclarez `main()` comme `main()` sans argument, ils sont néanmoins toujours présents sur la pile, mais ne sont pas utilisés. Si vous déclarez `main()` as comme `main(int argc, char *argv[])`, vous pourrez utiliser les deux premiers arguments, et le troisième restera « invisible » pour votre fonction. Il est même possible de déclarer `main()` comme `main(int argc)`, cela fonctionnera.

[60] infocenter.arm.com/help/index.jsp?topic=/com.arm.doc.faqs/ka13785.html

[61] Il y a quelque temps, sur PDP-11 et VAX, l'instruction CALL (appel d'autres fonctions) était coûteuse; jusqu'à 50% du temps d'exécution pouvait être passé à ça, il était donc considéré qu'avoir un grand nombre de petites fonctions était un anti-pattern [Eric S. Raymond, *The Art of UNIX Programming*, (2003)Chapter 4, Part II].

[62] Pas aléatoire dans le sens strict du terme, mais plutôt imprévisibles: **??** on page ??

Autres façons de passer les arguments

Il est à noter que rien n'oblige les programmeurs à passer les arguments à travers la pile. Ce n'est pas une exigence. On peut implémenter n'importe quelle autre méthode sans utiliser du tout la pile.

Une méthode répandue chez les débutants en assembleur est de passer les arguments par des variables globales, comme:

Listing 1.36: Code assembleur

```
        ...

        mov     X, 123
        mov     Y, 456
        call    do_something

        ...

X       dd      ?
Y       dd      ?

do_something proc near
        ; take X
        ; take Y
        ; do something
        retn
do_something endp
```

Mais cette méthode a un inconvénient évident: la fonction *do_something()* ne peut pas s'appeler elle-même récursivement (ou par une autre fonction), car il faudrait écraser ses propres arguments. La même histoire avec les variables locales: si vous les stockez dans des variables globales, la fonction ne peut pas s'appeler elle-même. Et ce n'est pas thread-safe [63]. Une méthode qui stocke ces informations sur la pile rend cela plus facile—elle peut contenir autant d'arguments de fonctions et/ou de valeurs, que la pile a d'espace.

[Donald E. Knuth, *The Art of Computer Programming*, Volume 1, 3rd ed., (1997), 189] mentionne un schéma encore plus étrange, particulièrement pratique sur les IBM System/360.

MS-DOS a une manière de passer tous les arguments de fonctions via des registres, par exemple, c'est un morceau de code pour un ancien MS-DOS 16-bit qui affiche "Hello, world!":

```
mov  dx, msg      ; address of message
mov  ah, 9        ; 9 means "print string" function
int  21h          ; DOS "syscall"

mov  ah, 4ch      ; "terminate program" function
int  21h          ; DOS "syscall"

msg  db 'Hello, World!\$'
```

C'est presque similaire à la méthode 4.1.3 on page 545. Et c'est aussi très similaire aux appels systèmes sous Linux (4.3.1 on page 558) et Windows.

Si une fonction MS-DOS devait renvoyer une valeur booléenne (i.e., un simple bit, souvent pour indiquer un état d'erreur), le flag CF était souvent utilisé.

Par exemple:

```
mov ah, 3ch       ; create file
lea dx, filename
mov cl, 1
int 21h
jc  error
mov file_handle, ax
...
error :
...
```

[63] Correctement implémenté, chaque thread aurait sa propre pile avec ses propres arguments/variables.

En cas d'erreur, le flag CF est mis. Sinon, le handle du fichier nouvellement créé est retourné via AX.

Cette méthode est encore utilisée par les programmeurs en langage d'assemblage. Dans le code source de Windows Research Kernel (qui est très similaire à Windows 2003) nous pouvons trouver quelque chose comme ça (file *base/ntos/ke/i386/cpu.asm*):

```
        public  Get386Stepping
Get386Stepping  proc

        call    MultiplyTest            ; Perform multiplication test
        jnc     short G3s00             ; if nc, muttest is ok
        mov     ax, 0
        ret
G3s00 :
        call    Check386B0              ; Check for B0 stepping
        jnc     short G3s05             ; if nc, it's B1/later
        mov     ax, 100h                ; It is B0/earlier stepping
        ret

G3s05 :
        call    Check386D1              ; Check for D1 stepping
        jc      short G3s10             ; if c, it is NOT D1
        mov     ax, 301h                ; It is D1/later stepping
        ret

G3s10 :
        mov     ax, 101h                ; assume it is B1 stepping
        ret

        ...

MultiplyTest    proc

        xor     cx,cx                   ; 64K times is a nice round number
mlt00 :  push    cx
        call    Multiply                ; does this chip's multiply work?
        pop     cx
        jc      short mltx              ; if c, No, exit
        loop    mlt00                   ; if nc, YEs, loop to try again
        clc
mltx :
        ret

MultiplyTest    endp
```

Stockage des variables locales

Une fonction peut allouer de l'espace sur la pile pour ses variables locales simplement en décrémentant le pointeur de pile vers le bas de la pile.

Donc, c'est très rapide, peu importe combien de variables locales sont définies. Ce n'est pas une nécessité de stocker les variables locales sur la pile. Vous pouvez les stocker où bon vous semble, mais c'est traditionnellement fait comme cela.

x86: alloca() function

Intéressons-nous à la fonction `alloca()` [64]

Cette fonction fonctionne comme `malloc()`, mais alloue de la mémoire directement sur la pile. L'espace de mémoire ne doit pas être libéré via un appel à la fonction `free()`, puisque l'épilogue de fonction (1.6 on page 30) remet ESP à son état initial ce qui va automatiquement libérer cet espace mémoire.

Intéressons-nous à l'implémentation d'`alloca()`. Cette fonction décale simplement ESP du nombre d'octets demandé vers le bas de la pile et définit ESP comme un pointeur vers la mémoire *allouée*.

[64] Avec MSVC, l'implémentation de cette fonction peut être trouvée dans les fichiers `alloca16.asm` et `chkstk.asm` dans `C:\Program Files (x86)\Microsoft Visual Studio 10.0\VC\crt\src\intel`

Essayons :

```
#ifdef __GNUC__
#include <alloca.h> // GCC
#else
#include <malloc.h> // MSVC
#endif
#include <stdio.h>

void f()
{
    char *buf=(char*)alloca (600);
#ifdef __GNUC__
    snprintf (buf, 600, "hi! %d, %d, %d\n", 1, 2, 3); // GCC
#else
    _snprintf (buf, 600, "hi! %d, %d, %d\n", 1, 2, 3); // MSVC
#endif

    puts (buf);
};
```

La fonction `_snprintf()` fonctionne comme `printf()`, mais au lieu d'afficher le résultat sur la sortie standard (ex., dans un terminal ou une console), il l'écrit dans le buffer `buf`. La fonction `puts()` copie le contenu de `buf` dans la sortie standard. Évidemment, ces deux appels de fonctions peuvent être remplacés par un seul appel à la fonction `printf()`, mais nous devons illustrer l'utilisation de petit buffer.

MSVC

Compilons (MSVC 2010) :

Listing 1.37: MSVC 2010

```
...

    mov    eax, 600  ; 00000258H
    call   __alloca_probe_16
    mov    esi, esp

    push   3
    push   2
    push   1
    push   OFFSET $SG2672
    push   600       ; 00000258H
    push   esi
    call   __snprintf

    push   esi
    call   _puts
    add    esp, 28

...
```

Le seul argument d'`alloca()` est passé via EAX (au lieu de le mettre sur la pile) [65].

GCC + Syntaxe Intel

GCC 4.4.1 fait la même chose sans effectuer d'appel à des fonctions externes :

Listing 1.38: GCC 4.7.3

```
.LC0 :
```

[65] C'est parce que alloca() est plutôt une fonctionnalité intrinsèque du compilateur (7.3 on page 640) qu'une fonction normale. Une des raisons pour laquelle nous avons besoin d'une fonction séparée au lieu de quelques instructions dans le code, est parce que l'implémentation d'alloca() par MSVC[66] a également du code qui lit depuis la mémoire récemment allouée pour laisser l'OS mapper la mémoire physique vers la VM[67]. Aprés l'appel à la fonction `alloca()`, ESP pointe sur un bloc de 600 octets que nous pouvons utiliser pour le tableau `buf`.

```
        .string "hi! %d, %d, %d\n"
f :
        push    ebp
        mov     ebp, esp
        push    ebx
        sub     esp, 660
        lea     ebx, [esp+39]
        and     ebx, -16                ; align pointer by 16-bit border
        mov     DWORD PTR [esp], ebx    ; s
        mov     DWORD PTR [esp+20], 3
        mov     DWORD PTR [esp+16], 2
        mov     DWORD PTR [esp+12], 1
        mov     DWORD PTR [esp+8], OFFSET FLAT :.LC0 ; "hi! %d, %d, %d\n"
        mov     DWORD PTR [esp+4], 600  ; maxlen
        call    _snprintf
        mov     DWORD PTR [esp], ebx    ; s
        call    puts
        mov     ebx, DWORD PTR [ebp-4]
        leave
        ret
```

GCC + Syntaxe AT&T

Voyons le même code mais avec la syntaxe AT&T :

Listing 1.39: GCC 4.7.3

```
.LC0 :
        .string "hi! %d, %d, %d\n"
f :
        pushl   %ebp
        movl    %esp, %ebp
        pushl   %ebx
        subl    $660, %esp
        leal    39(%esp), %ebx
        andl    $-16, %ebx
        movl    %ebx, (%esp)
        movl    $3, 20(%esp)
        movl    $2, 16(%esp)
        movl    $1, 12(%esp)
        movl    $.LC0, 8(%esp)
        movl    $600, 4(%esp)
        call    _snprintf
        movl    %ebx, (%esp)
        call    puts
        movl    -4(%ebp), %ebx
        leave
        ret
```

Le code est le même que le précédent.

Au fait, `movl $3, 20(%esp)` correspond à `mov DWORD PTR [esp+20], 3` avec la syntaxe intel. Dans la syntaxe AT&T, le format registre+offset pour l'adressage mémoire ressemble à `offset(%register)`.

(Windows) SEH

Les enregistrements SEH[68] sont aussi stockés dans la pile (s'ils sont présents). Lire à ce propos: (4.5.3 on page 575).

Protection contre les débordements de tampon

Lire à ce propos (1.20.2 on page 276).

[68] Structured Exception Handling

Dé-allocation automatique de données dans la pile

Peut-être que la raison pour laquelle les variables locales et les enregistrements SEH sont stockés dans la pile est qu'ils sont automatiquement libérés quand la fonction se termine en utilisant simplement une instruction pour corriger la position du pointeur de pile (souvent ADD). Les arguments de fonction sont aussi désalloués automatiquement à la fin de la fonction. À l'inverse, toutes les données allouées sur le *heap* doivent être désallouées de façon explicite.

1.7.3 Une disposition typique de la pile

Une disposition typique de la pile dans un environnement 32-bit au début d'une fonction, avant l'exécution de sa première instruction ressemble à ceci:

...	...
ESP-0xC	variable locale#2, marqué dans IDA comme var_8
ESP-8	variable locale#1, marqué dans IDA comme var_4
ESP-4	valeur enregistrée deEBP
ESP	Adresse de retour
ESP+4	argument#1, marqué dans IDA comme arg_0
ESP+8	argument#2, marqué dans IDA comme arg_4
ESP+0xC	argument#3, marqué dans IDA comme arg_8
...	...

1.7.4 Bruit dans la pile

> Quand quelqu'un dit que quelques chose est aléatoire, ce que cela signifie en pratique c'est qu'il n'est pas capable de voir les régularités de cette chose
>
> Stephen Wolfram, A New Kind of Science.

Dans ce livre les valeurs dites « bruitée » ou « poubelle » présente dans la pile ou dans la mémoire sont souvent mentionnées.

D'où viennent-elles ? Ces valeurs ont été laissées sur la pile après l'exécution de fonctions précédentes. Par exemple:

```
#include <stdio.h>

void f1()
{
        int a=1, b=2, c=3;
};

void f2()
{
        int a, b, c;
        printf ("%d, %d, %d\n", a, b, c);
};

int main()
{
        f1();
        f2();
};
```

Compilons ...

Listing 1.40: sans optimisation MSVC 2010

```
$SG2752 DB      '%d, %d, %d', 0aH, 00H

_c$ = -12       ; size = 4
_b$ = -8        ; size = 4
_a$ = -4        ; size = 4
```

```
_f1     PROC
        push    ebp
        mov     ebp, esp
        sub     esp, 12
        mov     DWORD PTR _a$[ebp], 1
        mov     DWORD PTR _b$[ebp], 2
        mov     DWORD PTR _c$[ebp], 3
        mov     esp, ebp
        pop     ebp
        ret     0
_f1     ENDP

_c$ = -12       ; size = 4
_b$ = -8        ; size = 4
_a$ = -4        ; size = 4
_f2     PROC
        push    ebp
        mov     ebp, esp
        sub     esp, 12
        mov     eax, DWORD PTR _c$[ebp]
        push    eax
        mov     ecx, DWORD PTR _b$[ebp]
        push    ecx
        mov     edx, DWORD PTR _a$[ebp]
        push    edx
        push    OFFSET $SG2752 ; '%d, %d, %d'
        call    DWORD PTR __imp__printf
        add     esp, 16
        mov     esp, ebp
        pop     ebp
        ret     0
_f2     ENDP

_main   PROC
        push    ebp
        mov     ebp, esp
        call    _f1
        call    _f2
        xor     eax, eax
        pop     ebp
        ret     0
_main   ENDP
```

Le compilateur va rouspéter un peu...

```
c :\Polygon\c>cl st.c /Fast.asm /MD
Microsoft (R) 32-bit C/C++ Optimizing Compiler Version 16.00.40219.01 for 80x86
Copyright (C) Microsoft Corporation.  All rights reserved.

st.c
c :\polygon\c\st.c(11)  : warning C4700 : uninitialized local variable 'c' used
c :\polygon\c\st.c(11)  : warning C4700 : uninitialized local variable 'b' used
c :\polygon\c\st.c(11)  : warning C4700 : uninitialized local variable 'a' used
Microsoft (R) Incremental Linker Version 10.00.40219.01
Copyright (C) Microsoft Corporation.  All rights reserved.

/out :st.exe
st.obj
```

Mais quand nous lançons le programme compilé ...

```
c :\Polygon\c>st
1, 2, 3
```

Quel résultat étrange ! Aucune variables n'a été initialisées dans `f2()`. Ce sont des valeurs « fantômes » qui sont toujours dans la pile.

Chargeons cet exemple dans OllyDbg :

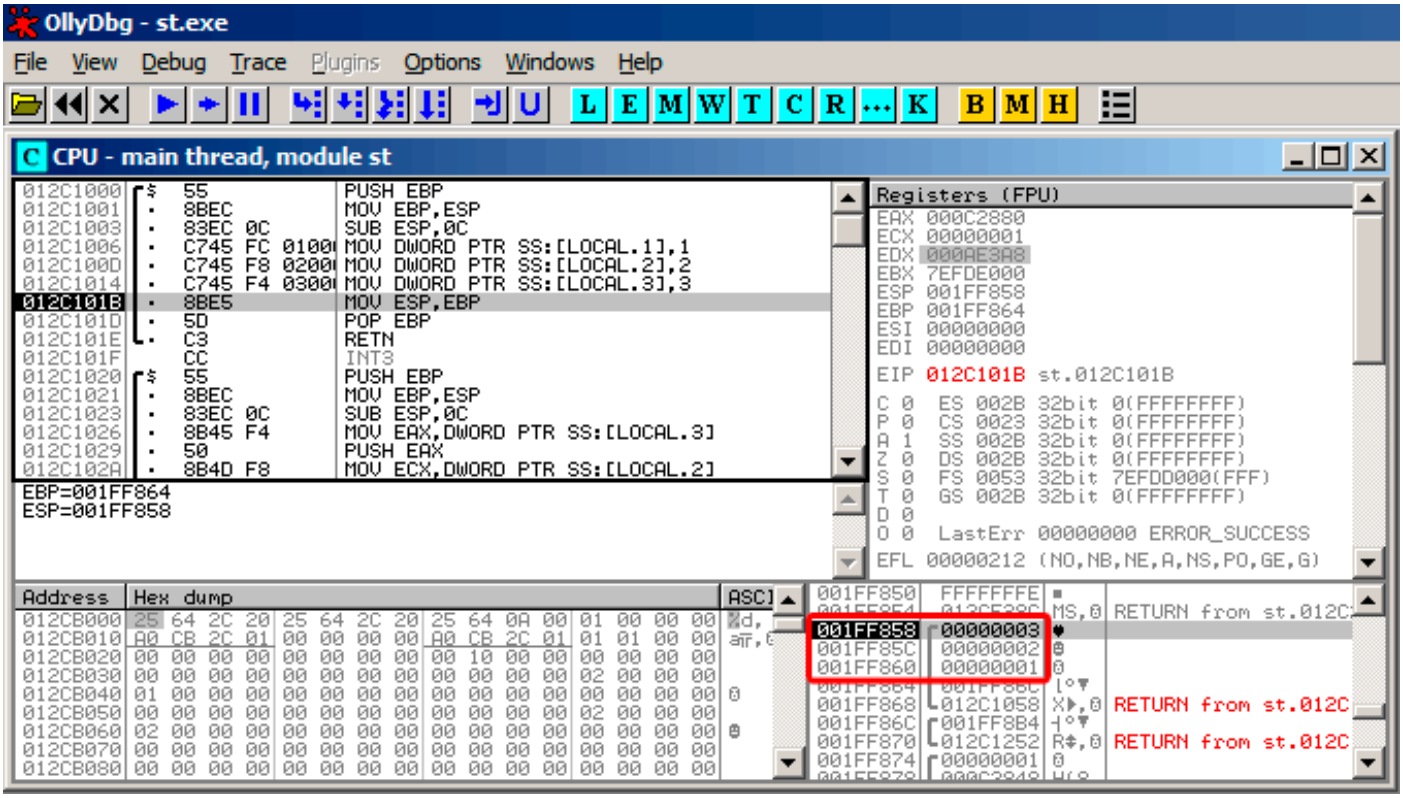


Fig. 1.5: OllyDbg : `f1()`

Quand `f1()` assigne les variable a, b et c, leurs valeurs sont stockées à l'adresse `0x1FF860` et ainsi de suite.

Et quand f2() s'exécute:

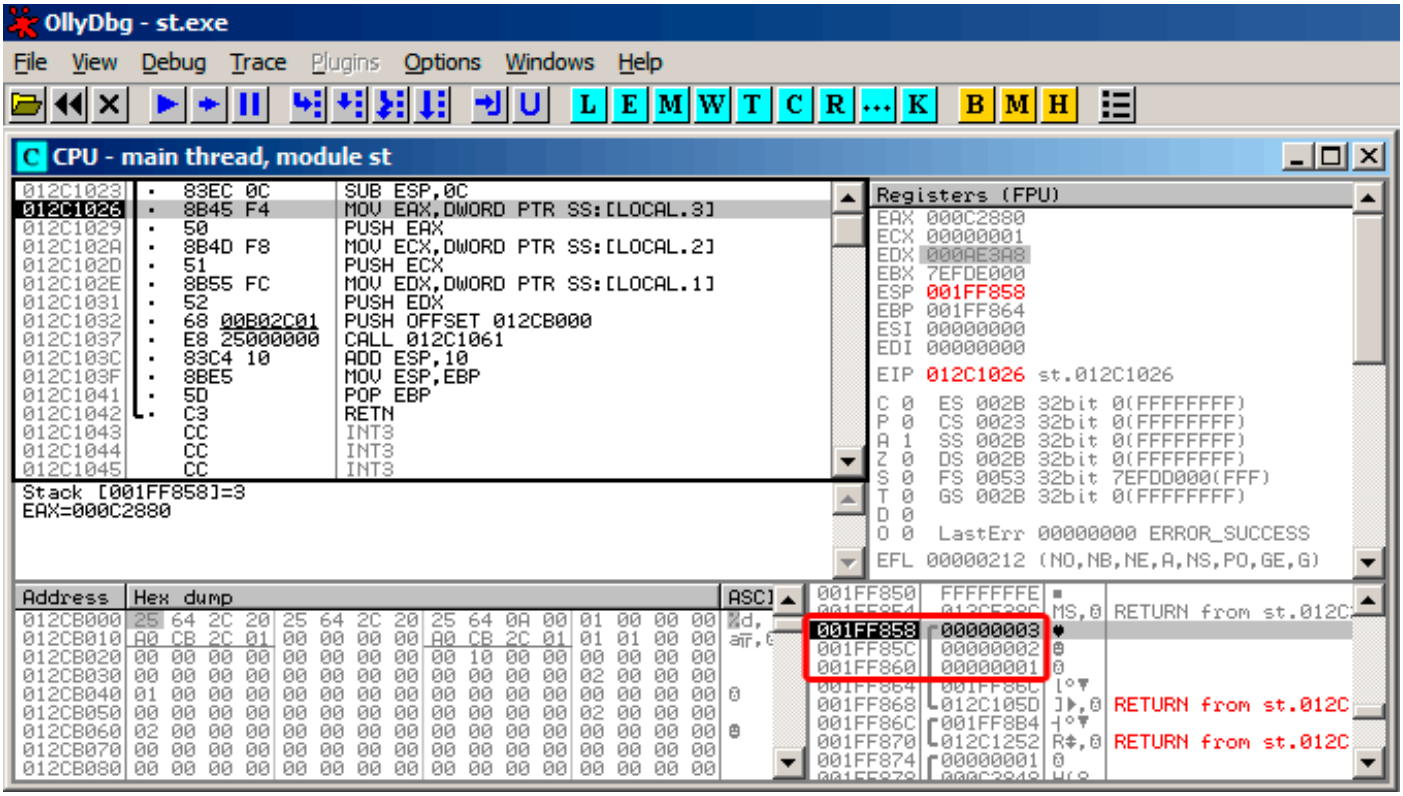


Fig. 1.6: OllyDbg : f2()

... a, b et c de la fonction f2() sont situées à la même adresse ! Aucunes autre fonction n'a encore écrasées ces valeurs, elles sont donc encore inchangées. Pour que cette situation arrive, il faut que plusieurs fonctions soit appelées les unes après les autres et que SP soit le même à chaque début de fonction (i.e., les fonctions doivent avoir le même nombre d'arguments). Les variables locales seront donc positionnées au même endroit dans la pile. Pour résumer, toutes les valeurs sur la pile sont des valeurs laissées par des appels de fonction précédents. Ces valeurs laissées sur la pile ne sont pas réellement aléatoires dans le sens strict du terme, mais elles sont imprévisibles. Y a t'il une autre option ? Il serait probablement possible de nettoyer des parties de la pile avant chaque nouvelle exécution de fonction, mais cela engendrerait du travail et du temps d'exécution (non nécessaire) en plus.

MSVC 2013

Cet exemple a été compilé avec MSVC 2010. Si vous essayez de compiler cet exemple avec MSVC 2013 et de l'exécuter, ces 3 nombres seront inversés:

```
c :\Polygon\c>st
3, 2, 1
```

Pourquoi ? J'ai aussi compilé cet exemple avec MSVC 2013 et constaté ceci:

Listing 1.41: MSVC 2013

```
_a$ = -12       ; size = 4
_b$ = -8        ; size = 4
_c$ = -4        ; size = 4
_f2     PROC

...

_f2     ENDP

_c$ = -12       ; size = 4
_b$ = -8        ; size = 4
_a$ = -4        ; size = 4
```

```
_f1     PROC

...

_f1     ENDP
```

Contrairement à MSVC 2010, MSVC 2013 alloue les variables a/b/c dans la fonction `f2()` dans l'ordre inverse puisqu'il se comporte différemment en raison d'un changement supposé dans son fonctionnement interne.Ceci est correct, car le standard du C/C++ n'a aucune règle sur l'ordre d'allocation des variables locales sur la pile.

1.7.5 Exercices

- http://challenges.re/51
- http://challenges.re/52

1.8 printf() avec plusieurs arguments

Maintenant, améliorons l'exemple *Hello, world!* (1.5 on page 9) en remplaçant `printf()` dans la fonction `main()` par ceci:

```
#include <stdio.h>

int main()
{
        printf("a=%d; b=%d; c=%d", 1, 2, 3);
        return 0;
};
```

1.8.1 x86

x86: 3 arguments

MSVC

En le compilant avec MSVC 2010 Express nous obtenons:

```
$SG3830 DB      'a=%d; b=%d; c=%d', 00H

...

        push    3
        push    2
        push    1
        push    OFFSET $SG3830
        call    _printf
        add     esp, 16                                 ; 00000010H
```

Presque la même chose, mais maintenant nous voyons que les arguments de `printf()` sont poussés sur la pile en ordre inverse. Le premier argument est poussé en dernier.

À propos, dans un environnement 32-bit les variables de type *int* ont une taille de 32-bit. ce qui fait 4 octets.

Donc, nous avons 4 arguments ici. $4 * 4 = 16$ —ils occupent exactement 16 octets dans la pile: un pointeur 32-bit sur une chaîne et 3 nombres de type *int*.

Lorsque le pointeur de pile (registre ESP) est re-modifié par l'instruction `ADD ESP, X` après un appel de fonction, souvent, le nombre d'arguments de la fonction peut-être déduit en divisant simplement X par 4.

Bien sûr, cela est spécifique à la convention d'appel *cdecl*, et seulement pour un environnement 32-bit.

Voir aussi la section sur les conventions d'appel (4.1 on page 544).

Dans certains cas, plusieurs fonctions se terminent les une après les autres, le compilateur peut concaténer plusieurs instructions « ADD ESP, X » en une seule, après le dernier appel:

```
push a1
push a2
call ...
...
push a1
call ...
...
push a1
push a2
push a3
call ...
add esp, 24
```

Voici un exemple réel:

Listing 1.42: x86

```
.text :100113E7    push    3
.text :100113E9    call    sub_100018B0 ; prendre un argument (3)
.text :100113EE    call    sub_100019D0 ; ne prendre aucun argument
.text :100113F3    call    sub_10006A90 ; ne prendre aucun argument
.text :100113F8    push    1
.text :100113FA    call    sub_100018B0 ; prendre un argument (1)
.text :100113FF    add     esp, 8       ; supprimer deux arguments de la pile à la fois
```

MSVC et OllyDbg

Maintenant, essayons de charger cet exemple dans OllyDbg. C'est l'un des débuggers en espace utilisateur win32 les plus populaire. Nous pouvons compiler notre exemple avec l'option `/MD` de MSVC 2012, qui signifie lier avec `MSVCR*.DLL`, ainsi nous verrons clairement les fonctions importées dans le debugger.

Ensuite chargeons l'exécutable dans OllyDbg. Le tout premier point d'arrêt est dans `ntdll.dll`, appuyez sur F9 (run). Le second point d'arrêt est dans le code CRT. Nous devons maintenant trouver la fonction `main()`.

Trouvez ce code en vous déplaçant au tout début du code (MSVC alloue la fonction `main()` au tout début de la section de code):

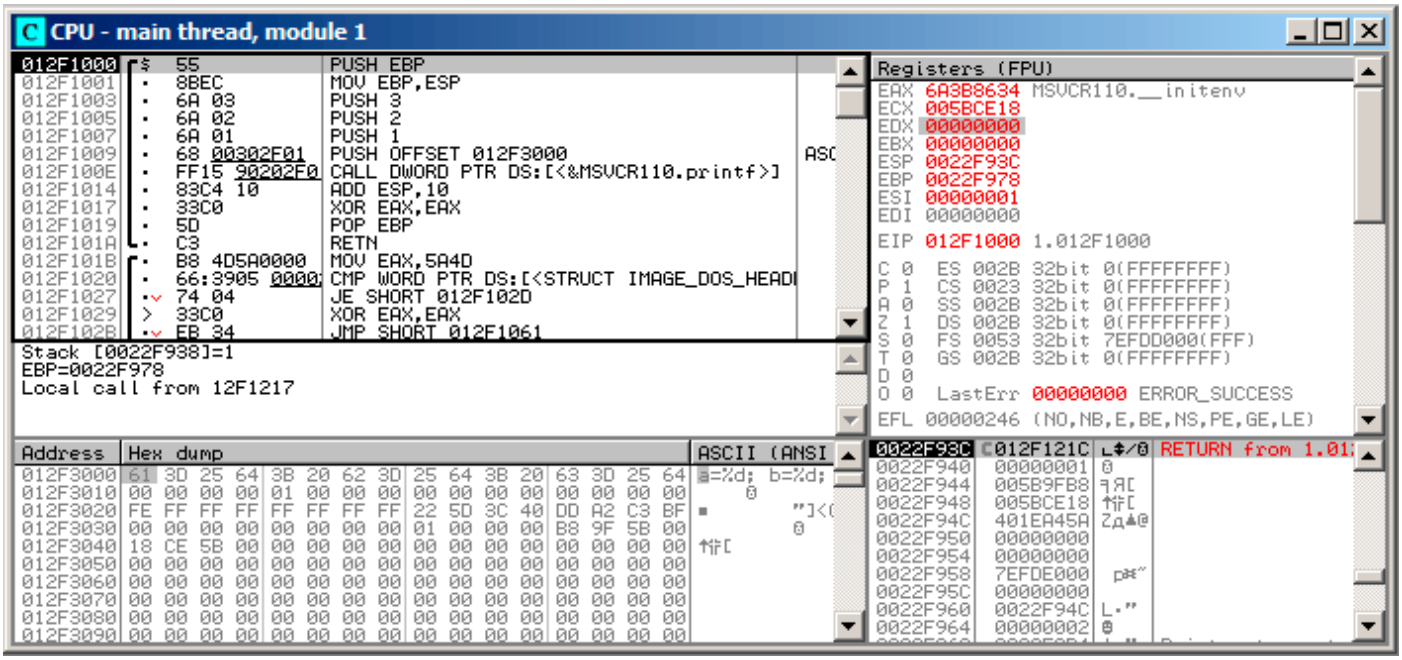


Fig. 1.7: OllyDbg : le tout début de la fonction `main()`

Clickez sur l'instruction PUSH EBP, pressez F2 (mettre un point d'arrêt) et pressez F9 (lancer le programme). Nous devons effectuer ces actions pour éviter le code CRT, car il ne nous intéresse pas pour le moment.

Presser F8 (enjamber) 6 fois, i.e. sauter 6 instructions:

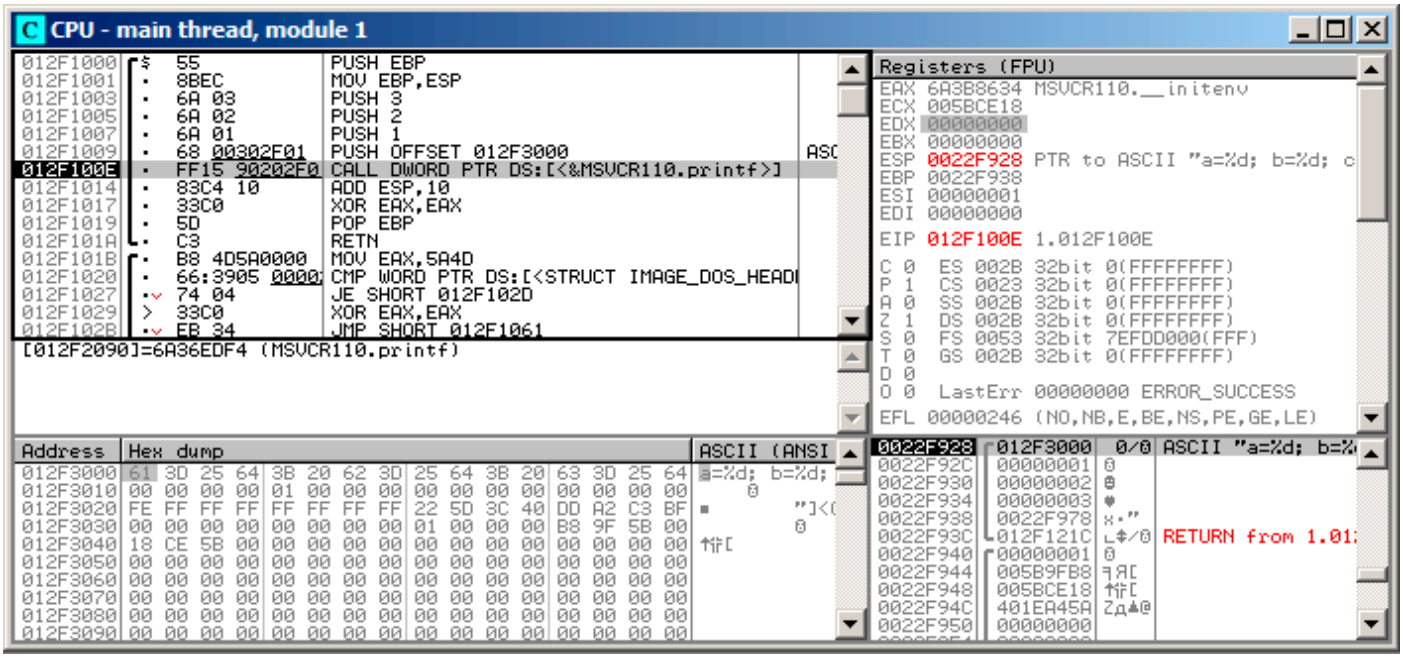


Fig. 1.8: OllyDbg : avant l'exécution de `printf()`

Maintenant le PC pointe vers l'instruction `CALL printf`. OllyDbg, comme d'autres debuggers, souligne la valeur des registres qui ont changé. Donc, chaque fois que vous appuyez sur F8, EIP change et sa valeur est affichée en rouge. ESP change aussi, car les valeurs des arguments sont poussées sur la pile.

Où sont les valeurs dans la pile? Regardez en bas à droite de la fenêtre du debugger:

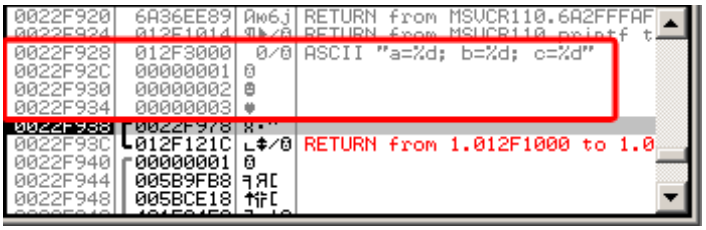


Fig. 1.9: OllyDbg : pile après que les valeurs des arguments aient été poussées (Le rectangle rouge a été ajouté par l'auteur dans un éditeur graphique)

Nous pouvons voir 3 colonnes ici: adresse dans la pile, valeur dans la pile et quelques commentaires additionnels d'OllyDbg. OllyDbg reconnaît les chaînes de type `printf()`, donc il signale ici la chaîne et les 3 valeurs *attachées* à elle.

Il est possible de faire un clic-droit sur la chaîne de format, cliquer sur « Follow in dump », et la chaîne de format va apparaître dans la fenêtre en bas à gauche du debugger. qui affiche toujours des parties de la mémoire. Ces valeurs en mémoire peuvent être modifiées. Il est possible de changer la chaîne de format, auquel cas le résultat de notre exemple sera différent. Cela n'est pas très utile dans le cas présent, mais ce peut-être un bon exercice pour commencer à comprendre comment tout fonctionne ici.

Appuyer sur F8 (enjamber).

Nous voyons la sortie suivante dans la console:

```
a=1; b=2; c=3
```

Regardons comment les registres et la pile ont changés:

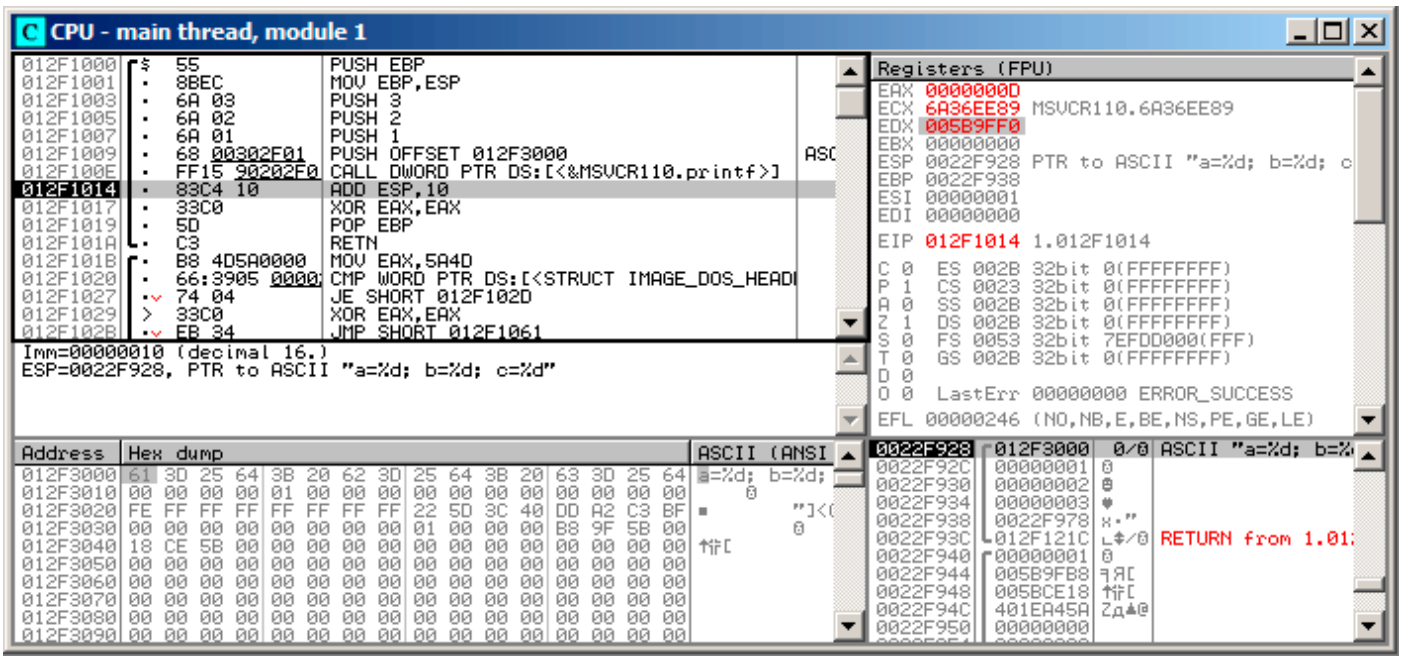


Fig. 1.10: OllyDbg après l'exécution de `printf()`

Le registre EAX contient maintenant `0xD` (13). C'est correct, puisque `printf()` renvoie le nombre de caractères écrits. La valeur de EIP a changé: en effet, il contient maintenant l'adresse de l'instruction venant après `CALL printf`. Les valeurs de ECX et EDX ont également changé. Apparemment, le mécanisme interne de la fonction `printf()` les a utilisés pour dans ses propres besoins.

Un fait très important est que ni la valeur de ESP, ni l'état de la pile n'ont été changés! Nous voyons clairement que la chaîne de format et les trois valeurs correspondantes sont toujours là. C'est en effet le comportement de la convention d'appel *cdecl* : l'appelée ne doit pas remettre ESP à sa valeur précédente. L'appelant est responsable de le faire.

Appuyer sur F8 à nouveau pour exécuter l'instruction `ADD ESP, 10` :

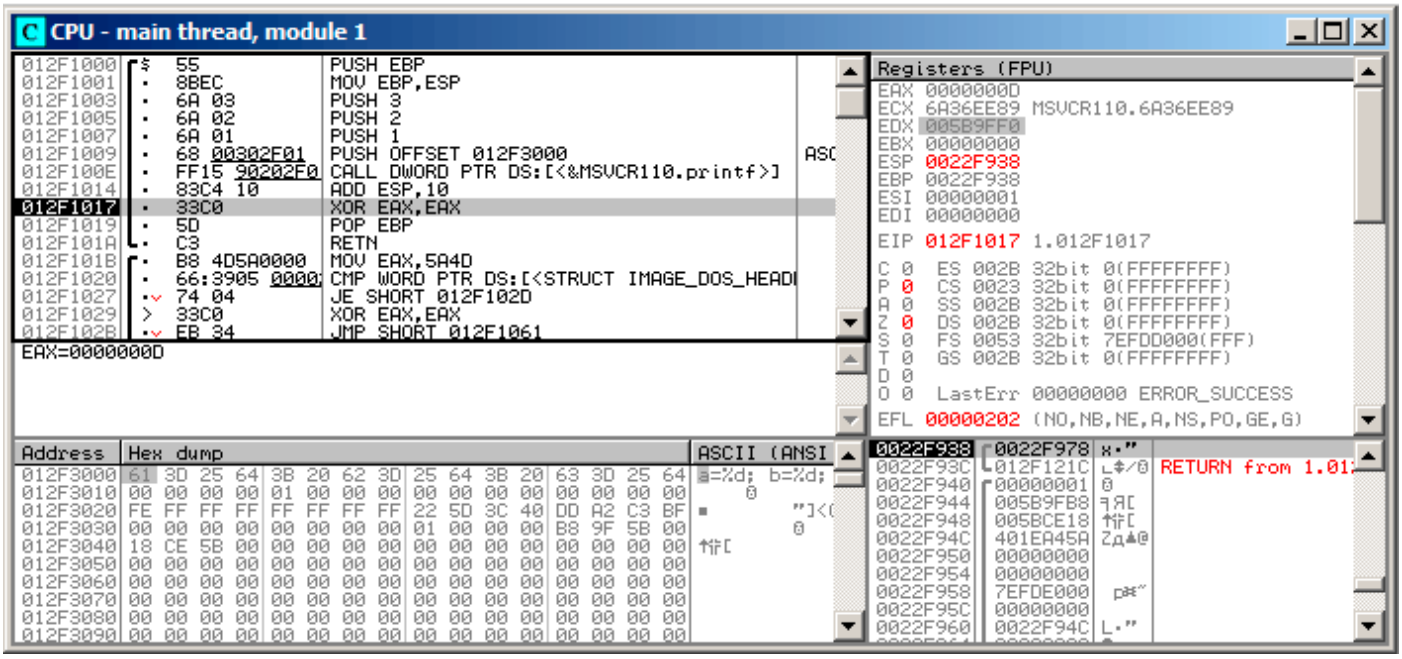


Fig. 1.11: OllyDbg : après l'exécution de l'instruction `ADD ESP, 10`

ESP a changé, mais les valeurs sont toujours dans la pile! Oui, bien sûr; il n'y a pas besoin de mettre ces valeurs à zéro ou quelque chose comme ça. Tout ce qui se trouve au dessus du pointeur de pile (SP) est du *bruit* ou des *déchets* et n'a pas du tout de signification. Ça prendrait beaucoup de temps de mettre à zéro les entrées inutilisées de la pile, et personne n'a vraiment besoin de le faire.

GCC

Maintenant, compilons la même programme sous Linux en utilisant GCC 4.4.1 et regardons ce que nous obtenons dans IDA :

```
main            proc near

var_10          = dword ptr -10h
var_C           = dword ptr -0Ch
var_8           = dword ptr -8
var_4           = dword ptr -4

                push    ebp
                mov     ebp, esp
                and     esp, 0FFFFFFF0h
                sub     esp, 10h
                mov     eax, offset aADBDCD ; "a=%d; b=%d; c=%d"
                mov     [esp+10h+var_4], 3
                mov     [esp+10h+var_8], 2
                mov     [esp+10h+var_C], 1
                mov     [esp+10h+var_10], eax
                call    _printf
                mov     eax, 0
                leave
                retn
main            endp
```

Il est visible que la différence entre le code MSVC et celui de GCC est seulement dans la manière dont les arguments sont stockés sur la pile. Ici GCC manipule directement la pile sans utiliser PUSH/POP.

GCC and GDB

Essayons cet exemple dans GDB[69] sous Linux.

L'option -g indique au compilateur d'inclure les informations de debug dans le fichier exécutable.

```
$ gcc 1.c -g -o 1
```

```
$ gdb 1
GNU gdb (GDB) 7.6.1-ubuntu
...
Reading symbols from /home/dennis/polygon/1...done.
```

Listing 1.43: let's set breakpoint on printf()

```
(gdb) b printf
Breakpoint 1 at 0x80482f0
```

Lançons le programme. Nous n'avons pas la code source de la fonction printf() ici, donc GDB ne peut pas le montrer, mais pourrait.

```
(gdb) run
Starting program : /home/dennis/polygon/1

Breakpoint 1, __printf (format=0x80484f0 "a=%d; b=%d; c=%d") at printf.c :29
29      printf.c : No such file or directory.
```

Afficher 10 éléments de la pile. La colonne la plus à gauche contient les adresses de la pile.

```
(gdb) x/10w $esp
0xbffff11c :     0x0804844a      0x080484f0      0x00000001      0x00000002
0xbffff12c :     0x00000003      0x08048460      0x00000000      0x00000000
0xbffff13c :     0xb7e29905      0x00000001
```

Le tout premier élément est la RA (0x0804844a). Nous pouvons le vérifier en désassemblant la mémoire à cette adresse:

```
(gdb) x/5i 0x0804844a
   0x804844a <main+45> : mov    $0x0,%eax
   0x804844f <main+50> : leave
   0x8048450 <main+51> : ret
   0x8048451 :   xchg   %ax,%ax
   0x8048453 :   xchg   %ax,%ax
```

Les deux instructions XCHG sont des instructions sans effet, analogues à NOPs.

Le second élément (0x080484f0) est l'adresse de la chaîne de format:

```
(gdb) x/s 0x080484f0
0x80484f0 :      "a=%d; b=%d; c=%d"
```

Les 3 éléments suivants (1, 2, 3) sont les arguments de printf(). Le reste des éléments sont juste des « restes » sur la pile, mais peuvent aussi être des valeurs d'autres fonctions, leurs variables locales, etc. Nous pouvons les ignorer pour le moment.

Lancer la commande « finish ». Cette commande ordonne à GDB d'« exécuter toutes les instructions jusqu'à la fin de la fonction ». Dans ce cas: exécuter jusqu'à la fin de printf().

```
(gdb) finish
Run till exit from #0  __printf (format=0x80484f0 "a=%d; b=%d; c=%d") at printf.c :29
main () at 1.c :6
6               return 0;
Value returned is $2 = 13
```

GDB montre ce que printf() a renvoyé dans EAX (13). C'est le nombre de caractères écrits, exactement comme dans l'exemple avec OllyDbg.

Nous voyons également « return 0; » et l'information que cette expression se trouve à la ligne 6 du fichier 1.c. En effet, le fichier 1.c se trouve dans le répertoire courant, et GDB y a trouvé la chaîne. Comment est-ce que GDB sait quelle ligne est exécutée à un instant donné? Cela est du au fait que lorsque le

[69] GNU Debugger

compilateur génère les informations de debug, il sauve également une table contenant la relation entre le numéro des lignes du code source et les adresses des instructions. GDB est un debugger niveau source, après tout.

Examinons les registres. 13 in EAX :

```
(gdb) info registers
eax            0xd      13
ecx            0x0      0
edx            0x0      0
ebx            0xb7fc0000       -1208221696
esp            0xbffff120       0xbffff120
ebp            0xbffff138       0xbffff138
esi            0x0      0
edi            0x0      0
eip            0x804844a        0x804844a <main+45>
...
```

Désassemblons les instructions courantes. La flèche pointe sur la prochaine instruction qui sera exécutée.

```
(gdb) disas
Dump of assembler code for function main :
   0x0804841d <+0> :     push   %ebp
   0x0804841e <+1> :     mov    %esp,%ebp
   0x08048420 <+3> :     and    $0xfffffff0,%esp
   0x08048423 <+6> :     sub    $0x10,%esp
   0x08048426 <+9> :     movl   $0x3,0xc(%esp)
   0x0804842e <+17> :    movl   $0x2,0x8(%esp)
   0x08048436 <+25> :    movl   $0x1,0x4(%esp)
   0x0804843e <+33> :    movl   $0x80484f0,(%esp)
   0x08048445 <+40> :    call   0x80482f0 <printf@plt>
=> 0x0804844a <+45> :    mov    $0x0,%eax
   0x0804844f <+50> :    leave
   0x08048450 <+51> :    ret
End of assembler dump.
```

GDB utilise la syntaxe AT&T par défaut. Mais il est possible de choisir la syntaxe Intel:

```
(gdb) set disassembly-flavor intel
(gdb) disas
Dump of assembler code for function main :
   0x0804841d <+0> :     push   ebp
   0x0804841e <+1> :     mov    ebp,esp
   0x08048420 <+3> :     and    esp,0xfffffff0
   0x08048423 <+6> :     sub    esp,0x10
   0x08048426 <+9> :     mov    DWORD PTR [esp+0xc],0x3
   0x0804842e <+17> :    mov    DWORD PTR [esp+0x8],0x2
   0x08048436 <+25> :    mov    DWORD PTR [esp+0x4],0x1
   0x0804843e <+33> :    mov    DWORD PTR [esp],0x80484f0
   0x08048445 <+40> :    call   0x80482f0 <printf@plt>
=> 0x0804844a <+45> :    mov    eax,0x0
   0x0804844f <+50> :    leave
   0x08048450 <+51> :    ret
End of assembler dump.
```

Exécuter l'instruction suivante. GDB montre une parenthèse fermante, signifiant la fin du bloc.

```
(gdb) step
7       };
```

Examinons les registres après l'exécution de l'instruction `MOV EAX, 0`. En effet, EAX est à zéro à ce stade.

```
(gdb) info registers
eax            0x0      0
ecx            0x0      0
edx            0x0      0
ebx            0xb7fc0000       -1208221696
esp            0xbffff120       0xbffff120
ebp            0xbffff138       0xbffff138
esi            0x0      0
```

```
edi            0x0      0
eip            0x804844f        0x804844f <main+50>
...
```

x64: 8 arguments

Pour voir comment les autres arguments sont passés par la pile, changeons encore notre exemple en augmentant le nombre d'arguments à 9 (chaîne de format de `printf()` + 8 variables *int*):

```
#include <stdio.h>

int main()
{
        printf("a=%d; b=%d; c=%d; d=%d; e=%d; f=%d; g=%d; h=%d\n", 1, 2, 3, 4, 5, 6, 7, 8);
        return 0;
};
```

MSVC

Comme il a déjà été mentionné, les 4 premiers arguments sont passés par les registres RCX, RDX, R8, R9 sous Win64, tandis les autres le sont—par la pile. C'est exactement de que l'on voit ici. Toutefois, l'instruction MOV est utilisée ici à la place de PUSH, donc les valeurs sont stockées sur la pile d'une manière simple.

Listing 1.44: MSVC 2012 x64

```
$SG2923 DB      'a=%d; b=%d; c=%d; d=%d; e=%d; f=%d; g=%d; h=%d', 0aH, 00H

main    PROC
        sub     rsp, 88

        mov     DWORD PTR [rsp+64], 8
        mov     DWORD PTR [rsp+56], 7
        mov     DWORD PTR [rsp+48], 6
        mov     DWORD PTR [rsp+40], 5
        mov     DWORD PTR [rsp+32], 4
        mov     r9d, 3
        mov     r8d, 2
        mov     edx, 1
        lea     rcx, OFFSET FLAT :$SG2923
        call    printf

        ; renvoyer 0
        xor     eax, eax

        add     rsp, 88
        ret     0
main    ENDP
_TEXT   ENDS
END
```

Le lecteur observateur pourrait demander pourquoi 8 octets sont alloués sur la pile pour les valeurs *int*, alors que 4 suffisent? Oui, il faut se rappeler: 8 octets sont alloués pour tout type de données plus petit que 64 bits. Ceci est instauré pour des raisons de commodités: cela rend facile le calcul de l'adresse de n'importe quel argument. En outre, ils sont tous situés à des adresses mémoires alignées. Il en est de même dans les environnements 32-bit: 4 octets sont réservés pour tout types de données.

GCC

Le tableau est similaire pour les OS x86-64 *NIX, excepté que les 6 premiers arguments sont passés par les registres RDI, RSI, RDX, RCX, R8, R9. Tout les autres—par la pile. GCC génère du code stockant le pointeur de chaîne dans EDI au lieu de RDI—nous l'avons noté précédemment: 1.5.2 on page 16.

Nous avions également noté que le registre EAX a été vidé avant l'appel à `printf()` : 1.5.2 on page 16.

Listing 1.45: GCC 4.4.6 x64 avec optimisation

```
.LC0 :
        .string "a=%d ; b=%d ; c=%d ; d=%d ; e=%d ; f=%d ; g=%d ; h=%d\n"

main :
        sub     rsp, 40

        mov     r9d, 5
        mov     r8d, 4
        mov     ecx, 3
        mov     edx, 2
        mov     esi, 1
        mov     edi, OFFSET FLAT :.LC0
        xor     eax, eax ; nombre de registres vectoriels
        mov     DWORD PTR [rsp+16], 8
        mov     DWORD PTR [rsp+8], 7
        mov     DWORD PTR [rsp], 6
        call    printf

        ; renvoyer 0

        xor     eax, eax
        add     rsp, 40
        ret
```

GCC + GDB

Essayons cet exemple dans GDB.

```
$ gcc -g 2.c -o 2
```

```
$ gdb 2
GNU gdb (GDB) 7.6.1-ubuntu
...
Reading symbols from /home/dennis/polygon/2...done.
```

Listing 1.46: mettons le point d'arrêt à `printf()`, et lançons

```
(gdb) b printf
Breakpoint 1 at 0x400410
(gdb) run
Starting program : /home/dennis/polygon/2

Breakpoint 1, __printf (format=0x400628 "a=%d ; b=%d ; c=%d ; d=%d ; e=%d ; f=%d ; g=%d ; h=%d\n") ↙
    ↳ at printf.c :29
29      printf.c : No such file or directory.
```

Les registres RSI/RDX/RCX/R8/R9 ont les valeurs attendues. RIP contient l'adresse de la toute première instruction de la fonction `printf()`.

```
(gdb) info registers
rax            0x0      0
rbx            0x0      0
rcx            0x3      3
rdx            0x2      2
rsi            0x1      1
rdi            0x400628 4195880
rbp            0x7fffffffdf60   0x7fffffffdf60
rsp            0x7fffffffdf38   0x7fffffffdf38
r8             0x4      4
r9             0x5      5
r10            0x7fffffffdce0   140737488346336
r11            0x7ffff7a65f60   140737348263776
r12            0x400440 4195392
r13            0x7fffffffe040   140737488347200
r14            0x0      0
r15            0x0      0
```

```
rip            0x7ffff7a65f60   0x7ffff7a65f60 <__printf>
...
```

Listing 1.47: inspectons la chaîne de format

```
(gdb) x/s $rdi
0x400628 :       "a=%d ; b=%d ; c=%d ; d=%d ; e=%d ; f=%d ; g=%d ; h=%d\n"
```

Affichons la pile avec la commande x/g cette fois—*g* est l'unité pour *giant words*, i.e., mots de 64-bit.

```
(gdb) x/10g $rsp
0x7fffffffdf38 : 0x0000000000400576      0x0000000000000006
0x7fffffffdf48 : 0x0000000000000007      0x00007fff00000008
0x7fffffffdf58 : 0x0000000000000000      0x0000000000000000
0x7fffffffdf68 : 0x00007ffff7a33de5      0x0000000000000000
0x7fffffffdf78 : 0x00007fffffffe048      0x0000000100000000
```

Le tout premier élément de la pile, comme dans le cas précédent, est la RA. 3 valeurs sont aussi passées par la pile: 6, 7, 8. Nous voyons également que 8 est passé avec les 32-bits de poids fort non effacés: 0x00007fff00000008. C'est en ordre, car les valeurs sont d'un type *int*, qui est 32-bit. Donc, la partie haute du registre ou l'élément de la pile peuvent contenir des « restes de données aléatoires ».

Si vous regardez où le contrôle reviendra après l'exécution de `printf()`, GDB affiche la fonction `main()` en entier:

```
(gdb) set disassembly-flavor intel
(gdb) disas 0x0000000000400576
Dump of assembler code for function main :
   0x000000000040052d <+0> :     push   rbp
   0x000000000040052e <+1> :     mov    rbp,rsp
   0x0000000000400531 <+4> :     sub    rsp,0x20
   0x0000000000400535 <+8> :     mov    DWORD PTR [rsp+0x10],0x8
   0x000000000040053d <+16> :    mov    DWORD PTR [rsp+0x8],0x7
   0x0000000000400545 <+24> :    mov    DWORD PTR [rsp],0x6
   0x000000000040054c <+31> :    mov    r9d,0x5
   0x0000000000400552 <+37> :    mov    r8d,0x4
   0x0000000000400558 <+43> :    mov    ecx,0x3
   0x000000000040055d <+48> :    mov    edx,0x2
   0x0000000000400562 <+53> :    mov    esi,0x1
   0x0000000000400567 <+58> :    mov    edi,0x400628
   0x000000000040056c <+63> :    mov    eax,0x0
   0x0000000000400571 <+68> :    call   0x400410 <printf@plt>
   0x0000000000400576 <+73> :    mov    eax,0x0
   0x000000000040057b <+78> :    leave
   0x000000000040057c <+79> :    ret
End of assembler dump.
```

Laissons se terminer l'exécution de `printf()`, exécutez l'instruction mettant EAX à zéro, et notez que le registre EAX à une valeur d'exactement zéro. RIP pointe maintenant sur l'instruction LEAVE, i.e, la pénultième de la fonction `main()`.

```
(gdb) finish
Run till exit from #0  __printf (format=0x400628 "a=%d ; b=%d ; c=%d ; d=%d ; e=%d ; f=%d ; g=%d ; h↙
    ↳ =%d\n") at printf.c :29
a=1; b=2; c=3; d=4; e=5; f=6; g=7; h=8
main () at 2.c :6
6               return 0;
Value returned is $1 = 39
(gdb) next
7       };
(gdb) info registers
rax            0x0      0
rbx            0x0      0
rcx            0x26     38
rdx            0x7ffff7dd59f0   140737351866864
rsi            0x7fffffd9       2147483609
rdi            0x0      0
rbp            0x7fffffffdf60   0x7fffffffdf60
rsp            0x7fffffffdf40   0x7fffffffdf40
r8             0x7ffff7dd26a0   140737351853728
```

```
r9             0x7ffff7a60134   140737348239668
r10            0x7fffffffd5b0   140737488344496
r11            0x7ffff7a95900   140737348458752
r12            0x400440 4195392
r13            0x7fffffffe040   140737488347200
r14            0x0      0
r15            0x0      0
rip            0x40057b 0x40057b <main+78>
...
```

1.8.2 ARM

ARM: 3 arguments

Le schéma ARM traditionnel pour passer des arguments (convention d'appel) se comporte de cette façon: les 4 premiers arguments sont passés par les registres R0-R3; les autres par la pile. Cela ressemble au schéma de passage des arguments dans fastcall (4.1.3 on page 545) ou win64 (4.1.5 on page 547).

ARM 32-bit

sans optimisation Keil 6/2013 (Mode ARM)

Listing 1.48: sans optimisation Keil 6/2013 (Mode ARM)

```
.text :00000000 main
.text :00000000 10 40 2D E9   STMFD   SP!, {R4,LR}
.text :00000004 03 30 A0 E3   MOV     R3, #3
.text :00000008 02 20 A0 E3   MOV     R2, #2
.text :0000000C 01 10 A0 E3   MOV     R1, #1
.text :00000010 08 00 8F E2   ADR     R0, aADBDCD     ; "a=%d; b=%d; c=%d"
.text :00000014 06 00 00 EB   BL      __2printf
.text :00000018 00 00 A0 E3   MOV     R0, #0          ; renvoyer 0
.text :0000001C 10 80 BD E8   LDMFD   SP!, {R4,PC}
```

Donc, les 4 premiers arguments sont passés par les registres R0-R3 dans cet ordre: un pointeur sur la chaîne de format de `printf()` dans R0, puis 1 dans R1, 2 dans R2 et 3 dans R3. L'instruction en 0x18 écrit 0 dans R0—c'est la déclaration C de *return 0*.

avec optimisation Keil 6/2013 génère le même code.

avec optimisation Keil 6/2013 (Mode Thumb)

Listing 1.49: avec optimisation Keil 6/2013 (Mode Thumb)

```
.text :00000000 main
.text :00000000 10 B5         PUSH    {R4,LR}
.text :00000002 03 23         MOVS    R3, #3
.text :00000004 02 22         MOVS    R2, #2
.text :00000006 01 21         MOVS    R1, #1
.text :00000008 02 A0         ADR     R0, aADBDCD     ; "a=%d; b=%d; c=%d"
.text :0000000A 00 F0 0D F8   BL      __2printf
.text :0000000E 00 20         MOVS    R0, #0
.text :00000010 10 BD         POP     {R4,PC}
```

Il n'y a pas de différence significative avec le code non optimisé pour le mode ARM.

avec optimisation Keil 6/2013 (Mode ARM) + supprimons le retour

Retravaillons légèrement l'exemple en supprimant *return 0* :

```
#include <stdio.h>

void main()
{
        printf("a=%d; b=%d; c=%d", 1, 2, 3);
};
```

Le résultat est quelque peu inhabituel:

Listing 1.50: avec optimisation Keil 6/2013 (Mode ARM)

```
.text :00000014 main
.text :00000014 03 30 A0 E3   MOV     R3, #3
.text :00000018 02 20 A0 E3   MOV     R2, #2
.text :0000001C 01 10 A0 E3   MOV     R1, #1
.text :00000020 1E 0E 8F E2   ADR     R0, aADBDCD     ; "a=%d; b=%d; c=%d\n"
.text :00000024 CB 18 00 EA   B       __2printf
```

C'est la version optimisée (`-03`) pour le mode ARM et cette fois nous voyons `B` comme dernière instruction au lieu du `BL` habituel. Une autre différence entre cette version optimisée et la précédente (compilée sans optimisation) est l'absence de fonctions prologue et épilogue (les instructions qui préservent les valeurs des registres `R0` et LR). L'instruction `B` saute simplement à une autre adresse, sans manipuler le registre LR, de façon similaire au `JMP` en x86. Pourquoi est-ce que fonctionne? Parce ce code est en fait bien équivalent au précédent. Il y a deux raisons principales: 1) Ni la pile ni SP (pointeur de pile) ne sont modifiés; 2) l'appel à `printf()` est la dernière instruction, donc il ne se passe rien après. A la fin, la fonction `printf()` rend simplement le contrôle à l'adresse stockée dans LR. Puisque LR contient actuellement l'adresse du point depuis lequel notre fonction a été appelée alors le contrôle après `printf()` sera redonné à ce point. Par conséquent, nous n'avons pas besoin de sauver LR car il ne nous est pas nécessaire de le modifier. Et il ne nous est non plus pas nécessaire de modifier LR car il n'y a pas d'autre appel de fonction excepté `printf()`. Par ailleurs, après cet appel nous ne faisons rien d'autre! C'est la raison pour laquelle une telle optimisation est possible.

Cette optimisation est souvent utilisée dans les fonctions où la dernière déclaration est un appel à une autre fonction. Un exemple similaire est présenté ici: 1.15.1 on page 155.

ARM64

GCC (Linaro) 4.9 sans optimisation

Listing 1.51: GCC (Linaro) 4.9 sans optimisation

```
.LC1 :
        .string "a=%d; b=%d; c=%d"
f2 :
; sauver FP et LR sur la pile :
        stp     x29, x30, [sp, -16]!
; définir la pile (FP=SP):
        add     x29, sp, 0
        adrp    x0, .LC1
        add     x0, x0,  :lo12 :.LC1
        mov     w1, 1
        mov     w2, 2
        mov     w3, 3
        bl      printf
        mov     w0, 0
; restaurer FP et LR
        ldp     x29, x30, [sp], 16
        ret
```

La première instruction STP (*Store Pair*) sauve FP (X29) et LR (X30) sur la pile.

La seconde instruction, `ADD X29, SP, 0` crée la pile. Elle écrit simplement la valeur de SP dans X29.

Ensuite nous voyons la paire d'instructions habituelle ADRP/ADD, qui crée le pointeur sur la chaîne. *lo12* signifie les 12 bits de poids faible, i.e., le linker va écrire les 12 bits de poids faible de l'adresse LC1 dans l'opcode de l'instruction ADD. `%d` dans la chaîne de format de `printf()` est un *int* 32-bit, les 1, 2 et 3 sont chargés dans les parties 32-bit des registres.

GCC (Linaro) 4.9 avec optimisation génère le même code.

ARM: 8 arguments

Utilisons de nouveau l'exemple avec 9 arguments de la section précédente: 1.8.1 on page 50.

```
#include <stdio.h>

int main()
{
        printf("a=%d; b=%d; c=%d; d=%d; e=%d; f=%d; g=%d; h=%d\n", 1, 2, 3, 4, 5, 6, 7, 8);
        return 0;
};
```

avec optimisation Keil 6/2013 : Mode ARM

```
.text :00000028             main
.text :00000028
.text :00000028             var_18 = -0x18
.text :00000028             var_14 = -0x14
.text :00000028             var_4  = -4
.text :00000028
.text :00000028 04 E0 2D E5  STR    LR, [SP,#var_4]!
.text :0000002C 14 D0 4D E2  SUB    SP, SP, #0x14
.text :00000030 08 30 A0 E3  MOV    R3, #8
.text :00000034 07 20 A0 E3  MOV    R2, #7
.text :00000038 06 10 A0 E3  MOV    R1, #6
.text :0000003C 05 00 A0 E3  MOV    R0, #5
.text :00000040 04 C0 8D E2  ADD    R12, SP, #0x18+var_14
.text :00000044 0F 00 8C E8  STMIA  R12, {R0-R3}
.text :00000048 04 00 A0 E3  MOV    R0, #4
.text :0000004C 00 00 8D E5  STR    R0, [SP,#0x18+var_18]
.text :00000050 03 30 A0 E3  MOV    R3, #3
.text :00000054 02 20 A0 E3  MOV    R2, #2
.text :00000058 01 10 A0 E3  MOV    R1, #1
.text :0000005C 6E 0F 8F E2  ADR    R0, aADBDCDDDEDFDGD ; "a=%d; b=%d; c=%d; d=%d; e=%d; f=%d↙
    ↳ ; g=%"...
.text :00000060 BC 18 00 EB  BL     __2printf
.text :00000064 14 D0 8D E2  ADD    SP, SP, #0x14
.text :00000068 04 F0 9D E4  LDR    PC, [SP+4+var_4],#4
```

Ce code peut être divisé en plusieurs parties:

- Prologue de la fonction:

  La toute première instruction `STR LR, [SP,#var_4]!` sauve LR sur la pile, car nous allons utiliser ce registre pour l'appel à `printf()`. Le point d'exclamation à la fin indique un *pré-index*.

  Cela signifie que SP est d'abord décrémenté de 4, et qu'ensuite LR va être sauvé à l'adresse stockée dans SP. C'est similaire à PUSH en x86. Lire aussi à ce propos: 1.32.2 on page 442.

  La seconde instruction `SUB SP, SP, #0x14` décrémente SP (le pointeur de pile) afin d'allouer `0x14` (20) octets sur la pile. En effet, nous devons passer 5 valeurs de 32-bit par la pile à la fonction `printf()`, et chacune occupe 4 octets, ce qui fait exactement $5 * 4 = 20$. Les 4 autres valeurs de 32-bit sont passées par les registres.

- Passer 5, 6, 7 et 8 par la pile: ils sont stockés dans les registres R0, R1, R2 et R3 respectivement. Ensuite, l'instruction `ADD R12, SP, #0x18+var_14` écrit l'adresse de la pile où ces 4 variables doivent

être stockées dans le registre R12. *var_14* est une macro d'assemblage, égal à -0x14, créée par IDA pour afficher commodément le code accédant à la pile. Les macros *var_?* générée par IDA reflètent les variables locales dans la pile.

Donc, SP+4 doit être stocké dans le registre R12.
L'instruction suivante `STMIA R12, R0-R3` écrit le contenu des registres R0-R3 dans la mémoire pointée par R12. STMIA est l'abréviation de *Store Multiple Increment After* (stocker plusieurs incrémenter après). « *Increment After* » signifie que R12 doit être incrémenté de 4 après l'écriture de chaque valeur d'un registre.

- Passer 4 par la pile: 4 est stocké dans R0 et ensuite, cette valeur, avec l'aide de l'instruction `STR R0, [SP,#0x18+var_18]` est sauvée dans la pile. *var_18* est -0x18, donc l'offset est 0, donc la valeur du registre R0 (4) est écrite à l'adresse écrite dans SP.
- Passer 1, 2 et 3 par des registres: Les valeurs des 3 premiers nombres (a,b,c) (respectivement 1, 2, 3) sont passées par les registres R1, R2 et R3 juste avant l'appel de `printf()`, et les 5 autres valeurs sont passées par la pile:
- appel de `printf()`
- Épilogue de fonction:

  L'instruction `ADD SP, SP, #0x14` restaure le pointeur SP à sa valeur précédente, nettoyant ainsi la pile. Bien sûr, ce qui a été stocké sur la pile y reste, mais sera récrit lors de l'exécution ultérieure de fonctions.

  L'instruction `LDR PC, [SP+4+var_4],#4` charge la valeur sauvée de LR depuis la pile dans le registre PC, provoquant ainsi la sortie de la fonction. Il n'y a pas de point d'exclamation—effectivement, PC est d'abord chargé depuis l'adresse stockées dans SP ($4 + var\_4 = 4 + (-4) = 0$, donc cette instruction est analogue à INSLDR PC, [SP],#4), et ensuite SP est incrémenté de 4. Il s'agit de *post-index*[70]. Pourquoi est-ce qu'IDA affiche l'instruction comme ça? Parce qu'il veut illustrer la disposition de la pile et le fait que var_4 est alloué pour sauver la valeur de LR dans la pile locale. Cette instruction est quelque peu similaire à `POP PC` en x86[71].

avec optimisation Keil 6/2013 : Mode Thumb

```
.text :0000001C             printf_main2
.text :0000001C
.text :0000001C             var_18 = -0x18
.text :0000001C             var_14 = -0x14
.text :0000001C             var_8  = -8
.text :0000001C
.text :0000001C 00 B5        PUSH    {LR}
.text :0000001E 08 23        MOVS    R3, #8
.text :00000020 85 B0        SUB     SP, SP, #0x14
.text :00000022 04 93        STR     R3, [SP,#0x18+var_8]
.text :00000024 07 22        MOVS    R2, #7
.text :00000026 06 21        MOVS    R1, #6
.text :00000028 05 20        MOVS    R0, #5
.text :0000002A 01 AB        ADD     R3, SP, #0x18+var_14
.text :0000002C 07 C3        STMIA   R3!, {R0-R2}
.text :0000002E 04 20        MOVS    R0, #4
.text :00000030 00 90        STR     R0, [SP,#0x18+var_18]
.text :00000032 03 23        MOVS    R3, #3
.text :00000034 02 22        MOVS    R2, #2
.text :00000036 01 21        MOVS    R1, #1
.text :00000038 A0 A0        ADR     R0, aADBDCDDDEDFDGD ; "a=%d; b=%d; c=%d; d=%d; e=%d; f=%↙
    ↳ d; g=%"...
.text :0000003A 06 F0 D9 F8  BL      __2printf
.text :0000003E
.text :0000003E             loc_3E   ; CODE XREF : example13_f+16
.text :0000003E 05 B0        ADD     SP, SP, #0x14
.text :00000040 00 BD        POP     {PC}
```

[70] Lire à ce propos: 1.32.2 on page 442.
[71] Il est impossible de définir la valeur de IP/EIP/RIP en utilisant POP en x86, mais de toutes façons, vous avez le droit de faire l'analogie.

La sortie est presque comme dans les exemples précédents. Toutefois, c'est du code Thumb et les valeurs sont arrangées différemment dans la pile: 8 vient en premier, puis 5, 6, 7 et 4 vient en troisième.

avec optimisation Xcode 4.6.3 (LLVM) : Mode ARM

```
__text :0000290C             _printf_main2
__text :0000290C
__text :0000290C             var_1C = -0x1C
__text :0000290C             var_C  = -0xC
__text :0000290C
__text :0000290C 80 40 2D E9   STMFD  SP!, {R7,LR}
__text :00002910 0D 70 A0 E1   MOV    R7, SP
__text :00002914 14 D0 4D E2   SUB    SP, SP, #0x14
__text :00002918 70 05 01 E3   MOV    R0, #0x1570
__text :0000291C 07 C0 A0 E3   MOV    R12, #7
__text :00002920 00 00 40 E3   MOVT   R0, #0
__text :00002924 04 20 A0 E3   MOV    R2, #4
__text :00002928 00 00 8F E0   ADD    R0, PC, R0
__text :0000292C 06 30 A0 E3   MOV    R3, #6
__text :00002930 05 10 A0 E3   MOV    R1, #5
__text :00002934 00 20 8D E5   STR    R2, [SP,#0x1C+var_1C]
__text :00002938 0A 10 8D E9   STMFA  SP, {R1,R3,R12}
__text :0000293C 08 90 A0 E3   MOV    R9, #8
__text :00002940 01 10 A0 E3   MOV    R1, #1
__text :00002944 02 20 A0 E3   MOV    R2, #2
__text :00002948 03 30 A0 E3   MOV    R3, #3
__text :0000294C 10 90 8D E5   STR    R9, [SP,#0x1C+var_C]
__text :00002950 A4 05 00 EB   BL     _printf
__text :00002954 07 D0 A0 E1   MOV    SP, R7
__text :00002958 80 80 BD E8   LDMFD  SP!, {R7,PC}
```

Presque la même chose que ce que nous avons déjà vu, avec l'exception de l'instruction STMFA (Store Multiple Full Ascending), qui est un synonyme de l'instruction STMIB (Store Multiple Increment Before). Cette instruction incrémente la valeur du registre SP et écrit seulement après la valeur registre suivant dans la mémoire, plutôt que d'effectuer ces deux actions dans l'ordre inverse.

Une autre chose qui accroche le regard est que les instructions semblent être arrangées de manière aléatoire. Par exemple, la valeur dans le registre R0 est manipulée en trois endroits, aux adresses 0x2918, 0x2920 et 0x2928, alors qu'il serait possible de le faire en un seul endroit.

Toutefois, le compilateur qui optimise doit avoir ses propres raisons d'ordonner les instructions pour avoir une plus grande efficacité à l'exécution.

D'habitude, le processeur essaye d'exécuter simultanément des instructions situées côte à côte.
Par exemple, des instructions comme `MOVT R0, #0` et `ADD R0, PC, R0` ne peuvent pas être exécutées simultanément puisqu'elles modifient toutes deux le registre R0. D'un autre côté, les instructions `MOVT R0, #0` et `MOV R2, #4` peuvent être exécutées simultanément puisque leurs effets n'interfèrent pas l'un avec l'autre lors de leurs exécution. Probablement que le compilateur essaye de générer du code arrangé de cette façon (lorsque c'est possible).

avec optimisation Xcode 4.6.3 (LLVM) : Mode Thumb-2

```
__text :00002BA0              _printf_main2
__text :00002BA0
__text :00002BA0              var_1C = -0x1C
__text :00002BA0              var_18 = -0x18
__text :00002BA0              var_C  = -0xC
__text :00002BA0
__text :00002BA0 80 B5          PUSH     {R7,LR}
__text :00002BA2 6F 46          MOV      R7, SP
__text :00002BA4 85 B0          SUB      SP, SP, #0x14
__text :00002BA6 41 F2 D8 20    MOVW     R0, #0x12D8
__text :00002BAA 4F F0 07 0C    MOV.W    R12, #7
__text :00002BAE C0 F2 00 00    MOVT.W   R0, #0
__text :00002BB2 04 22          MOVS     R2, #4
__text :00002BB4 78 44          ADD      R0, PC   ; char *
```

```
__text :00002BB6 06 23          MOVS     R3, #6
__text :00002BB8 05 21          MOVS     R1, #5
__text :00002BBA 0D F1 04 0E    ADD.W    LR, SP, #0x1C+var_18
__text :00002BBE 00 92          STR      R2, [SP,#0x1C+var_1C]
__text :00002BC0 4F F0 08 09    MOV.W    R9, #8
__text :00002BC4 8E E8 0A 10    STMIA.W  LR, {R1,R3,R12}
__text :00002BC8 01 21          MOVS     R1, #1
__text :00002BCA 02 22          MOVS     R2, #2
__text :00002BCC 03 23          MOVS     R3, #3
__text :00002BCE CD F8 10 90    STR.W    R9, [SP,#0x1C+var_C]
__text :00002BD2 01 F0 0A EA    BLX      _printf
__text :00002BD6 05 B0          ADD      SP, SP, #0x14
__text :00002BD8 80 BD          POP      {R7,PC}
```

La sortie est presque la même que dans l'exemple précédent, avec l'exception que des instructions Thumb sont utilisées à la place.

ARM64

GCC (Linaro) 4.9 sans optimisation

Listing 1.52: GCC (Linaro) 4.9 sans optimisation

```
.LC2 :
        .string "a=%d; b=%d; c=%d; d=%d; e=%d; f=%d; g=%d; h=%d\n"
f3 :
; Réserver plus d'espace dans la pile:
        sub     sp, sp, #32
; sauver FP et LR sur la pile :
        stp     x29, x30, [sp,16]
; définir la pile (FP=SP):
        add     x29, sp, 16
        adrp    x0, .LC2  ; "a=%d; b=%d; c=%d; d=%d; e=%d; f=%d; g=%d; h=%d\n"
        add     x0, x0,  :lo12 :.LC2
        mov     w1, 8             ; 9ème argument
        str     w1, [sp]          ; stocker le 9ème argument dans la pile
        mov     w1, 1
        mov     w2, 2
        mov     w3, 3
        mov     w4, 4
        mov     w5, 5
        mov     w6, 6
        mov     w7, 7
        bl      printf
        sub     sp, x29, #16
; restaurer FP et LR
        ldp     x29, x30, [sp,16]
        add     sp, sp, 32
        ret
```

Les 8 premiers arguments sont passés dans des registres X- ou W-: [*Procedure Call Standard for the ARM 64-bit Architecture (AArch64)*, (2013)][72]. Un pointeur de chaîne nécessite un registre 64-bit, donc il est passé dans X0. Toutes les autres valeurs ont un type *int* 32-bit, donc elles sont stockées dans la partie 32-bit des registres (W-). Le 9ème argument (8) est passé par la pile. En effet: il n'est pas possible de passer un grand nombre d'arguments par les registres, car le nombre de registres est limité.

GCC (Linaro) 4.9 avec optimisation génère le même code.

[72] Aussi disponible en http://go.yurichev.com/17287

1.8.3 MIPS

3 arguments

GCC 4.4.5 avec optimisation

La différence principale avec l'exemple « Hello, world! » est que dans ce cas, `printf()` est appelée à la place de `puts()` et 3 arguments de plus sont passés à travers les registres $5...$7 (ou $A0...$A2). C'est pourquoi ces registres sont préfixés avec A-, ceci sous-entend qu'ils sont utilisés pour le passage des arguments aux fonctions.

Listing 1.53: GCC 4.4.5 avec optimisation (résultat en sortie de l'assembleur)

```
$LC0 :
        .ascii  "a=%d; b=%d; c=%d\000"
main :
; prologue de la fonction :
        lui     $28,%hi(__gnu_local_gp)
        addiu   $sp,$sp,-32
        addiu   $28,$28,%lo(__gnu_local_gp)
        sw      $31,28($sp)
; charger l'adresse de printf() :
        lw      $25,%call16(printf)($28)
; charger l'adresse de la chaîne de texte et mettre le 1er argument de printf():
        lui     $4,%hi($LC0)
        addiu   $4,$4,%lo($LC0)
; mettre le 2nd argument de printf() :
        li      $5,1                    # 0x1
; mettre le 3ème argument de printf():
        li      $6,2                    # 0x2
; appeler printf() :
        jalr    $25
; mettre le 4ème argument de printf() (slot de délai branchement):
        li      $7,3                    # 0x3

; épilogue de la fonction:
        lw      $31,28($sp)
; mettre la valeur de retour à 0:
        move    $2,$0
; retourner
        j       $31
        addiu   $sp,$sp,32 ; slot de délai de branchement
```

Listing 1.54: GCC 4.4.5 avec optimisation (IDA)

```
.text :00000000 main :
.text :00000000
.text :00000000 var_10          = -0x10
.text :00000000 var_4           = -4
.text :00000000
; prologue de la fonction :
.text :00000000                 lui     $gp, (__gnu_local_gp >> 16)
.text :00000004                 addiu   $sp, -0x20
.text :00000008                 la      $gp, (__gnu_local_gp & 0xFFFF)
.text :0000000C                 sw      $ra, 0x20+var_4($sp)
.text :00000010                 sw      $gp, 0x20+var_10($sp)
; charger l'adresse de printf() :
.text :00000014                 lw      $t9, (printf & 0xFFFF)($gp)
; charger l'adresse de la chaîne de texte et mettre le 1er argument de printf():
.text :00000018                 la      $a0, $LC0        # "a=%d; b=%d; c=%d"
; mettre le 2nd argument de printf() :
.text :00000020                 li      $a1, 1
; mettre le 3ème argument de printf():
.text :00000024                 li      $a2, 2
; appeler printf() :
.text :00000028                 jalr    $t9
; mettre le 4ème argument de printf(): (slot de délai de branchement)
.text :0000002C                 li      $a3, 3
; épilogue de la fonction:
```

```
.text :00000030                 lw      $ra, 0x20+var_4($sp)
; mettre la valeur de retour à 0:
.text :00000034                 move    $v0, $zero
; retourner
.text :00000038                 jr      $ra
.text :0000003C                 addiu   $sp, 0x20 ; slot de délai de branchement
```

IDA a agrégé la paire d'instructions LUI et ADDIU en une pseudo instruction LA. C'est pourquoi il n'y a pas d'instruction à l'adresse 0x1C: car LA *occupe* 8 octets.

GCC 4.4.5 sans optimisation

GCC sans optimisation est plus verbeux:

Listing 1.55: GCC 4.4.5 sans optimisation (résultat en sortie de l'assembleur)

```
$LC0 :
        .ascii  "a=%d; b=%d; c=%d\000"
main :
; prologue de la fonction :
        addiu   $sp,$sp,-32
        sw      $31,28($sp)
        sw      $fp,24($sp)
        move    $fp,$sp
        lui     $28,%hi(__gnu_local_gp)
        addiu   $28,$28,%lo(__gnu_local_gp)
; charger l'adresse de la chaîne de texte:
        lui     $2,%hi($LC0)
        addiu   $2,$2,%lo($LC0)
; mettre le 1er argument de printf() :
        move    $4,$2
; mettre le 2nd argument de printf() :
        li      $5,1                    # 0x1
; mettre le 3ème argument de printf():
        li      $6,2                    # 0x2
; mettre le 4ème argument de printf():
        li      $7,3                    # 0x3
; charger l'adresse de printf() :
        lw      $2,%call16(printf)($28)
        nop
; appeler printf() :
        move    $25,$2
        jalr    $25
        nop

; épilogue de la fonction:
        lw      $28,16($fp)
; mettre la valeur de retour à 0:
        move    $2,$0
        move    $sp,$fp
        lw      $31,28($sp)
        lw      $fp,24($sp)
        addiu   $sp,$sp,32
; retourner
        j       $31
        nop
```

Listing 1.56: GCC 4.4.5 sans optimisation (IDA)

```
.text :00000000 main :
.text :00000000
.text :00000000 var_10          = -0x10
.text :00000000 var_8           = -8
.text :00000000 var_4           = -4
.text :00000000
; prologue de la fonction :
.text :00000000                 addiu   $sp, -0x20
.text :00000004                 sw      $ra, 0x20+var_4($sp)
```

```
.text :00000008                 sw      $fp, 0x20+var_8($sp)
.text :0000000C                 move    $fp, $sp
.text :00000010                 la      $gp, __gnu_local_gp
.text :00000018                 sw      $gp, 0x20+var_10($sp)
; charge l'adresse de la chaîne de texte:
.text :0000001C                 la      $v0, aADBDCD     # "a=%d; b=%d; c=%d"
; mettre le 1er argument de printf() :
.text :00000024                 move    $a0, $v0
; mettre le 2nd argument de printf() :
.text :00000028                 li      $a1, 1
; mettre le 3ème argument de printf():
.text :0000002C                 li      $a2, 2
; mettre le 4ème argument de printf():
.text :00000030                 li      $a3, 3
; charger l'adresse de printf() :
.text :00000034                 lw      $v0, (printf & 0xFFFF)($gp)
.text :00000038                 or      $at, $zero
; appeler printf() :
.text :0000003C                 move    $t9, $v0
.text :00000040                 jalr    $t9
.text :00000044                 or      $at, $zero ; NOP
; épilogue de la fonction:
.text :00000048                 lw      $gp, 0x20+var_10($fp)
; mettre la valeur de retour à 0:
.text :0000004C                 move    $v0, $zero
.text :00000050                 move    $sp, $fp
.text :00000054                 lw      $ra, 0x20+var_4($sp)
.text :00000058                 lw      $fp, 0x20+var_8($sp)
.text :0000005C                 addiu   $sp, 0x20
; retourner
.text :00000060                 jr      $ra
.text :00000064                 or      $at, $zero ; NOP
```

8 arguments

Utilisons encore l'exemple de la section précédente avec 9 arguments: 1.8.1 on page 50.

```
#include <stdio.h>

int main()
{
        printf("a=%d; b=%d; c=%d; d=%d; e=%d; f=%d; g=%d; h=%d\n", 1, 2, 3, 4, 5, 6, 7, 8);
        return 0;
};
```

GCC 4.4.5 avec optimisation

Seul les 4 premiers arguments sont passés dans les registres $A0 ...$A3, les autres sont passés par la pile.

C'est la convention d'appel O32 (qui est la plus commune dans le monde MIPS). D'autres conventions d'appel (comme N32) peuvent utiliser les registres à d'autres fins.

SW est l'abbréviation de « Store Word » (depuis un registre vers la mémoire). En MIPS, il manque une instructions pour stocker une valeur dans la mémoire, donc une paire d'instruction doit être utilisée à la place (LI/SW).

Listing 1.57: GCC 4.4.5 avec optimisation (résultat en sortie de l'assembleur)

```
$LC0 :
        .ascii  "a=%d; b=%d; c=%d; d=%d; e=%d; f=%d; g=%d; h=%d\012\000"
main :
; prologue de la fonction :
        lui     $28,%hi(__gnu_local_gp)
        addiu   $sp,$sp,-56
        addiu   $28,$28,%lo(__gnu_local_gp)
```

```
        sw      $31,52($sp)
; passer le 5ème argument dans la pile:
        li      $2,4                    # 0x4
        sw      $2,16($sp)
; passer le 6ème argument dans la pile:
        li      $2,5                    # 0x5
        sw      $2,20($sp)
; passer le 7ème argument dans la pile:
        li      $2,6                    # 0x6
        sw      $2,24($sp)
; passer le 8ème argument dans la pile:
        li      $2,7                    # 0x7
        lw      $25,%call16(printf)($28)
        sw      $2,28($sp)
; passer le 1er argument dans $a0 :
        lui     $4,%hi($LC0)
; passer le 9ème argument dans la pile:
        li      $2,8                    # 0x8
        sw      $2,32($sp)
        addiu   $4,$4,%lo($LC0)
; passer le 2nd argument dans $a1 :
        li      $5,1                    # 0x1
; passer le 3ème argument dans $a2 :
        li      $6,2                    # 0x2
; appeler printf() :
        jalr    $25
; passer le 4ème argument dans $a3 (slot de délai de branchement):
        li      $7,3                    # 0x3

; épilogue de la fonction:
        lw      $31,52($sp)
; mettre la valeur de retour à 0:
        move    $2,$0
; retourner
        j       $31
        addiu   $sp,$sp,56 ; slot de délai de branchement
```

Listing 1.58: GCC 4.4.5 avec optimisation (IDA)

```
.text :00000000 main :
.text :00000000
.text :00000000 var_28          = -0x28
.text :00000000 var_24          = -0x24
.text :00000000 var_20          = -0x20
.text :00000000 var_1C          = -0x1C
.text :00000000 var_18          = -0x18
.text :00000000 var_10          = -0x10
.text :00000000 var_4           = -4
.text :00000000
; prologue de la fonction :
.text :00000000                 lui     $gp, (__gnu_local_gp >> 16)
.text :00000004                 addiu   $sp, -0x38
.text :00000008                 la      $gp, (__gnu_local_gp & 0xFFFF)
.text :0000000C                 sw      $ra, 0x38+var_4($sp)
.text :00000010                 sw      $gp, 0x38+var_10($sp)
; passer le 5ème argument dans la pile:
.text :00000014                 li      $v0, 4
.text :00000018                 sw      $v0, 0x38+var_28($sp)
; passer le 6ème argument dans la pile:
.text :0000001C                 li      $v0, 5
.text :00000020                 sw      $v0, 0x38+var_24($sp)
; passer le 7ème argument dans la pile:
.text :00000024                 li      $v0, 6
.text :00000028                 sw      $v0, 0x38+var_20($sp)
; passer le 8ème argument dans la pile:
.text :0000002C                 li      $v0, 7
.text :00000030                 lw      $t9, (printf & 0xFFFF)($gp)
.text :00000034                 sw      $v0, 0x38+var_1C($sp)
; préparer le 1er argument dans $a0:
.text :00000038                 lui     $a0, ($LC0 >> 16)  # "a=%d; b=%d; c=%d; d=%d; e=%d; f↙
```

```
    ↳ =%d ; g=%"...
; passer le 9ème argument dans la pile:
.text :0000003C                 li      $v0, 8
.text :00000040                 sw      $v0, 0x38+var_18($sp)
; passer le 1er argument in $a0 :
.text :00000044                 la      $a0, ($LC0 & 0xFFFF)  # "a=%d ; b=%d ; c=%d ; d=%d ; e=%d ;↙
    ↳  f=%d ; g=%"...
; passer le 2nd argument dans $a1 :
.text :00000048                 li      $a1, 1
; passer le 3ème argument dans $a2 :
.text :0000004C                 li      $a2, 2
; appeler printf() :
.text :00000050                 jalr    $t9
; passer le 4ème argument dans $a3 (slot de délai de branchement):
.text :00000054                 li      $a3, 3
; épilogue de la fonction:
.text :00000058                 lw      $ra, 0x38+var_4($sp)
; mettre la valeur de retour à 0:
.text :0000005C                 move    $v0, $zero
; retourner
.text :00000060                 jr      $ra
.text :00000064                 addiu   $sp, 0x38 ; slot de délai de branchement
```

GCC 4.4.5 sans optimisation

GCC sans optimisation est plus verbeux:

Listing 1.59: sans optimisation GCC 4.4.5 (résultat en sortie de l'assembleur)

```
$LC0 :
        .ascii  "a=%d ; b=%d ; c=%d ; d=%d ; e=%d ; f=%d ; g=%d ; h=%d\012\000"
main :
; prologue de la fonction :
        addiu   $sp,$sp,-56
        sw      $31,52($sp)
        sw      $fp,48($sp)
        move    $fp,$sp
        lui     $28,%hi(__gnu_local_gp)
        addiu   $28,$28,%lo(__gnu_local_gp)
        lui     $2,%hi($LC0)
        addiu   $2,$2,%lo($LC0)
; passer le 5ème argument dans la pile:
        li      $3,4                    # 0x4
        sw      $3,16($sp)
; passer le 6ème argument dans la pile:
        li      $3,5                    # 0x5
        sw      $3,20($sp)
; passer le 7ème argument dans la pile:
        li      $3,6                    # 0x6
        sw      $3,24($sp)
; passer le 8ème argument dans la pile:
        li      $3,7                    # 0x7
        sw      $3,28($sp)
; passer le 9ème argument dans la pile:
        li      $3,8                    # 0x8
        sw      $3,32($sp)
; passer le 1er argument dans $a0 :
        move    $4,$2
; passer le 2nd argument dans $a1 :
        li      $5,1                    # 0x1
; passer le 3ème argument dans $a2 :
        li      $6,2                    # 0x2
; passer le 4ème argument dans $a3 :
        li      $7,3                    # 0x3
; appeler printf() :
        lw      $2,%call16(printf)($28)
        nop
        move    $25,$2
```

```
        jalr    $25
        nop
; épilogue de la fonction:
        lw      $28,40($fp)
; mettre la valeur de retour à 0:
        move    $2,$0
        move    $sp,$fp
        lw      $31,52($sp)
        lw      $fp,48($sp)
        addiu   $sp,$sp,56
; retourner
        j       $31
        nop
```

Listing 1.60: sans optimisation GCC 4.4.5 (IDA)

```
.text :00000000 main :
.text :00000000
.text :00000000 var_28          = -0x28
.text :00000000 var_24          = -0x24
.text :00000000 var_20          = -0x20
.text :00000000 var_1C          = -0x1C
.text :00000000 var_18          = -0x18
.text :00000000 var_10          = -0x10
.text :00000000 var_8           = -8
.text :00000000 var_4           = -4
.text :00000000
; prologue de la fonction :
.text :00000000                 addiu   $sp, -0x38
.text :00000004                 sw      $ra, 0x38+var_4($sp)
.text :00000008                 sw      $fp, 0x38+var_8($sp)
.text :0000000C                 move    $fp, $sp
.text :00000010                 la      $gp, __gnu_local_gp
.text :00000018                 sw      $gp, 0x38+var_10($sp)
.text :0000001C                 la      $v0, aADBDCDDDEDFDGD  # "a=%d; b=%d; c=%d; d=%d; e=%d;↙
    ↳ f=%d; g=%"...
; passer le 5ème argument dans la pile:
.text :00000024                 li      $v1, 4
.text :00000028                 sw      $v1, 0x38+var_28($sp)
; passer le 6ème argument dans la pile:
.text :0000002C                 li      $v1, 5
.text :00000030                 sw      $v1, 0x38+var_24($sp)
; passer le 7ème argument dans la pile:
.text :00000034                 li      $v1, 6
.text :00000038                 sw      $v1, 0x38+var_20($sp)
; passer le 8ème argument dans la pile:
.text :0000003C                 li      $v1, 7
.text :00000040                 sw      $v1, 0x38+var_1C($sp)
; passer le 9ème argument dans la pile:
.text :00000044                 li      $v1, 8
.text :00000048                 sw      $v1, 0x38+var_18($sp)
; passer le 1er argument dans $a0 :
.text :0000004C                 move    $a0, $v0
; passer le 2nd argument dans $a1 :
.text :00000050                 li      $a1, 1
; passer le 3ème argument dans $a2 :
.text :00000054                 li      $a2, 2
; passer le 4ème argument dans $a3 :
.text :00000058                 li      $a3, 3
; appeler printf() :
.text :0000005C                 lw      $v0, (printf & 0xFFFF)($gp)
.text :00000060                 or      $at, $zero
.text :00000064                 move    $t9, $v0
.text :00000068                 jalr    $t9
.text :0000006C                 or      $at, $zero ; NOP
; épilogue de la fonction:
.text :00000070                 lw      $gp, 0x38+var_10($fp)
; mettre la valeur de retour à 0:
.text :00000074                 move    $v0, $zero
.text :00000078                 move    $sp, $fp
```

```
.text :0000007C                 lw      $ra, 0x38+var_4($sp)
.text :00000080                 lw      $fp, 0x38+var_8($sp)
.text :00000084                 addiu   $sp, 0x38
; retourner
.text :00000088                 jr      $ra
.text :0000008C                 or      $at, $zero ; NOP
```

1.8.4 Conclusion

Voici un schéma grossier de l'appel de la fonction:

Listing 1.61: x86

```
...
PUSH 3rd argument
PUSH 2nd argument
PUSH 1st argument
CALL fonction
; modifier le pointeur de pile (si besoin)
```

Listing 1.62: x64 (MSVC)

```
MOV RCX, 1st argument
MOV RDX, 2nd argument
MOV R8, 3rd argument
MOV R9, 4th argument
...
PUSH 5ème, 6ème argument, etc. (si besoin)
CALL fonction
; modifier le pointeur de pile (si besoin)
```

Listing 1.63: x64 (GCC)

```
MOV RDI, 1st argument
MOV RSI, 2nd argument
MOV RDX, 3rd argument
MOV RCX, 4th argument
MOV R8, 5th argument
MOV R9, 6th argument
...
PUSH 7ème, 8ème argument, etc. (si besoin)
CALL fonction
; modifier le pointeur de pile (si besoin)
```

Listing 1.64: ARM

```
MOV R0, 1st argument
MOV R1, 2nd argument
MOV R2, 3rd argument
MOV R3, 4th argument
; passer le 5ème, 6ème argument, etc., dans la pile (si besoin)
BL fonction
; modifier le pointeur de pile (si besoin)
```

Listing 1.65: ARM64

```
MOV X0, 1st argument
MOV X1, 2nd argument
MOV X2, 3rd argument
MOV X3, 4th argument
MOV X4, 5th argument
MOV X5, 6th argument
MOV X6, 7th argument
MOV X7, 8th argument
; passer le 9ème, 10ème argument, etc., dans la pile (si besoin)
BL fonction
; modifier le pointeur de pile (si besoin)
```

Listing 1.66: MIPS (O32 calling convention)

```
LI $4, 1st argument ; AKA $A0
LI $5, 2nd argument ; AKA $A1
LI $6, 3rd argument ; AKA $A2
LI $7, 4th argument ; AKA $A3
; passer le 5ème, 6ème argument, etc., dans la pile (si besoin)
LW temp_reg, adresse de la fonction
JALR temp_reg
```

1.8.5 À propos

À propos, cette différence dans le passage des arguments entre x86, x64, fastcall, ARM et MIPS est une bonne illustration du fait que le CPU est inconscient de comment les arguments sont passés aux fonctions. Il est aussi possible de créer un hypothétique compilateur capable de passer les arguments via une structure spéciale sans utiliser du tout la pile.

Les registres MIPS $A0 ...$A3 sont appelés comme ceci par commodité (c'est dans la convention d'appel O32). Les programmeurs peuvent utiliser n'importe quel autre registre, (bon, peut-être à l'exception de $ZERO) pour passer des données ou n'importe quelle autre convention d'appel.

Le CPU n'est pas au courant de quoi que ce soit des conventions d'appel.

Nous pouvons aussi nous rappeler comment les débutants en langage d'assemblage passent les arguments aux autres fonctions: généralement par les registres, sans ordre explicite, ou même par des variables globales. Bien sûr, cela fonctionne.

1.9 scanf()

Maintenant utilisons la fonction scanf().

1.9.1 Exemple simple

```
#include <stdio.h>

int main()
{
        int x;
        printf ("Enter X :\n");

        scanf ("%d", &x);

        printf ("You entered %d...\n", x);

        return 0;
};
```

Il n'est pas astucieux d'utiliser `scanf()` pour les interactions utilisateurs de nos jours. Mais nous pouvons, toutefois, illustrer le passage d'un pointeur sur une variable de type *int*.

À propos des pointeurs

Les pointeurs sont l'un des concepts fondamentaux de l'informatique. Souvent, passer un gros tableau, structure ou objet comme argument à une autre fonction est trop coûteux, tandis que passer leur adresse l'est très peu. Par exemple, si vous voulez afficher une chaîne de texte sur la console, il est plus facile de passer son adresse au noyau de l'OS.

En plus, si la fonction appelée doit modifier quelque chose dans un gros tableau ou structure reçu comme paramètre et renvoyer le tout, la situation est proche de l'absurde. Donc, la chose la plus simple est de passer l'adresse du tableau ou de la structure à la fonction appelée, et de la laisser changer ce qui doit l'être.

Un pointeur en C/C++—est simplement l'adresse d'un emplacement mémoire quelconque.

En x86, l'adresse est représentée par un nombre de 32-bit (i.e., il occupe 4 octets), tandis qu'en x86-64 c'est un nombre de 64-bit (occupant 8 octets). À propos, c'est la cause de l'indignation de certaines personnes concernant le changement vers x86-64—tous les pointeurs en architecture x64 ayant besoin de deux fois plus de place, incluant la mémoire cache, qui est de la mémoire "coûteuse".

Il est possible de travailler seulement avec des pointeurs non typés, moyennant quelques efforts; e.g. la fonction C standard `memcpy()`, qui copie un bloc de mémoire d'un endroit à un autre, prend 2 pointeurs de type `void*` comme arguments, puisqu'il est impossible de prévoir le type de données qu'il faudra copier. Les types de données ne sont pas importants, seule la taille du bloc compte.

Les pointeurs sont aussi couramment utilisés lorsqu'une fonction doit renvoyer plus d'une valeur (nous reviendrons là-dessus plus tard (1.12 on page 110)).

La fonction *scanf()*—en est une telle.

Hormis le fait que la fonction doit indiquer combien de valeurs ont été lues avec succès, elle doit aussi renvoyer toutes ces valeurs.

En C/C++ le type du pointeur est seulement nécessaire pour la vérification de type lors de la compilation.

Il n'y a aucune information du tout sur le type des pointeurs à l'intérieur du code compilé.

x86

MSVC

Voici ce que l'on obtient après avoir compilé avec MSVC 2010:

```
CONST    SEGMENT
$SG3831    DB    'Enter X :', 0aH, 00H
$SG3832    DB    '%d', 00H
$SG3833    DB    'You entered %d...', 0aH, 00H
CONST    ENDS
PUBLIC    _main
EXTRN    _scanf :PROC
EXTRN    _printf :PROC
; Options de compilation de la fonction : /Odtp
_TEXT    SEGMENT
_x$ = -4                        ; size = 4
_main    PROC
    push   ebp
    mov    ebp, esp
    push   ecx
    push   OFFSET $SG3831 ; 'Enter X :'
    call   _printf
    add    esp, 4
    lea    eax, DWORD PTR _x$[ebp]
    push   eax
    push   OFFSET $SG3832 ; '%d'
    call   _scanf
    add    esp, 8
    mov    ecx, DWORD PTR _x$[ebp]
    push   ecx
    push   OFFSET $SG3833 ; 'You entered %d...'
    call   _printf
    add    esp, 8

    ; retourner 0
    xor    eax, eax
    mov    esp, ebp
    pop    ebp
    ret    0
_main    ENDP
_TEXT    ENDS
```

x est une variable locale.

D'après le standard C/C++ elle ne doit être visible que dans cette fonction et dans aucune autre portée. Traditionnellement, les variables locales sont stockées sur la pile. Il y a probablement d'autres moyens de les allouer, mais en x86, c'est la façon de faire.

Le but de l'instruction suivant le prologue de la fonction, `PUSH ECX`, n'est pas de sauver l'état de ECX (noter l'absence d'un `POP ECX` à la fin de la fonction).

En fait, cela alloue 4 octets sur la pile pour stocker la variable x.

x est accédée à l'aide de la macro `_x$` (qui vaut -4) et du registre EBP qui pointe sur la structure de pile courante.

Pendant la durée de l'exécution de la fonction, EBP pointe sur la structure locale de pile courante, rendant possible l'accès aux variables locales et aux arguments de la fonction via `EBP+offset`.

Il est aussi possible d'utiliser ESP dans le même but, bien que ça ne soit pas très commode, car il change fréquemment. La valeur de EBP peut être perçue comme un *état figé* de la valeur de ESP au début de l'exécution de la fonction.

Voici une structure de pile typique dans un environnement 32-bit:

...	...
EBP-8	variable locale #2, marqué dans IDA comme `var_8`
EBP-4	variable locale #1, marqué dans IDA comme `var_4`
EBP	valeur sauvée de EBP
EBP+4	adresse de retour
EBP+8	argument#1, marqué dans IDA comme `arg_0`
EBP+0xC	argument#2, marqué dans IDA comme `arg_4`
EBP+0x10	argument#3, marqué dans IDA comme `arg_8`
...	...

La fonction `scanf()` de notre exemple a deux arguments.

Le premier est un pointeur sur la chaîne contenant `%d` et le second est l'adresse de la variable x.

Tout d'abord, l'adresse de la variable x est chargée dans le registre EAX par l'instruction `lea eax, DWORD PTR _x$[ebp]`.

LEA signifie *load effective address* (charger l'adresse effective) et est souvent utilisée pour composer une adresse (**??** on page ??).

Nous pouvons dire que dans ce cas, LEA stocke simplement la somme de la valeur du registre EBP et de la macro `_x$` dans le registre EAX.

C'est la même chose que `lea eax, [ebp-4]`.

Donc, 4 est soustrait de la valeur du registre EBP et le résultat est chargé dans le registre EAX. Ensuite, la valeur du registre EAX est poussée sur la pile et `scanf()` est appelée.

`printf()` est appelée ensuite avec son premier argument — un pointeur sur la chaîne: `You entered %d...\n`.

Le second argument est préparé avec: `mov ecx, [ebp-4]`. L'instruction stocke la valeur de la variable x et non son adresse, dans le registre ECX.

Puis, la valeur de ECX est stockée sur la pile et le dernier appel à `printf()` est effectué.

MSVC + OllyDbg

Essayons cet exemple dans OllyDbg. Chargeons-le et appuyons sur F8 (enjamber) jusqu'à ce que nous atteignons notre exécutable au lieu de `ntdll.dll`. Défiler vers le haut jusqu'à ce que `main()` apparaisse.

Cliquer sur la première instruction (PUSH EBP), appuyer sur F2 (*set a breakpoint*), puis F9 (*Run*). Le point d'arrêt sera déclenché lorsque `main()` commencera.

Continuons jusqu'au point où la variable x est calculée:

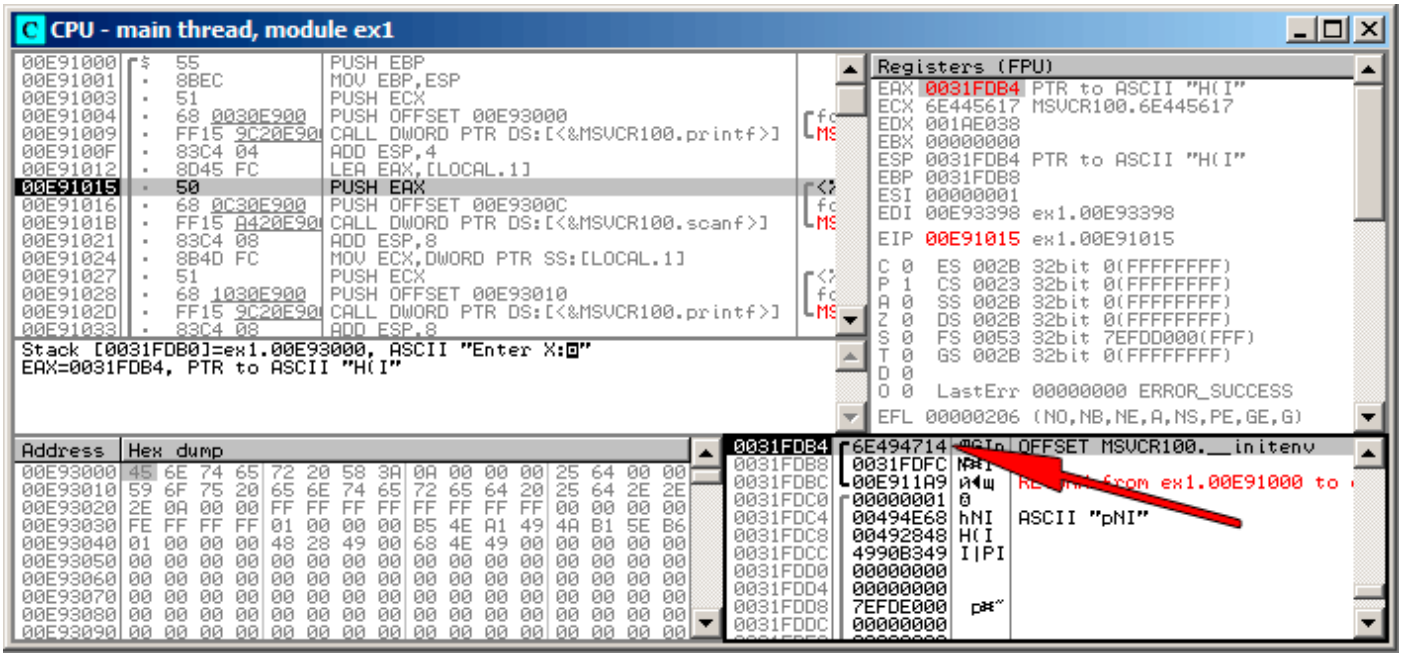


Fig. 1.12: OllyDbg : L'adresse de la variable locale est calculée

Cliquer droit sur EAX dans la fenêtre des registres et choisir « Follow in stack ».

Cette adresse va apparaître dans la fenêtre de la pile. La flèche rouge a été ajoutée, pointant la variable dans la pile locale. A ce point, cet espace contient des restes de données (0x6E494714). Maintenant. avec l'aide de l'instruction PUSH, l'adresse de cet élément de pile va être stockée sur la même pile à la position suivante. Appuyons sur F8 jusqu'à la fin de l'exécution de `scanf()`. Pendant l'exécution de `scanf()`, entrons, par exemple, 123, dans la fenêtre de la console:

```
Enter X :
123
```

scanf() a déjà fini de s'exécuter:

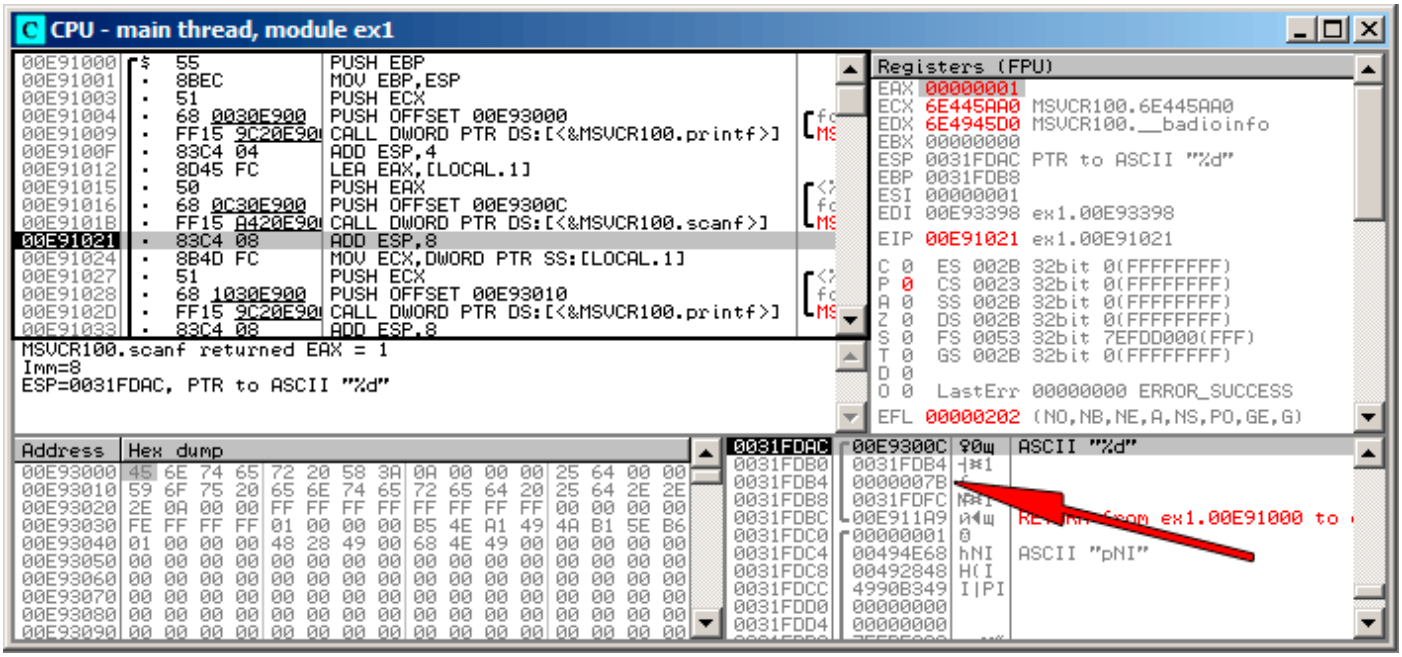


Fig. 1.13: OllyDbg : scanf() s'est exécutée

scanf() renvoie 1 dans EAX, ce qui indique qu'elle a lu avec succès une valeur. Si nous regardons de nouveau l'élément de la pile correspondant à la variable locale, il contient maintenant 0x7B (123).

Plus tard, cette valeur est copiée de la pile vers le registre ECX et passée à `printf()` :

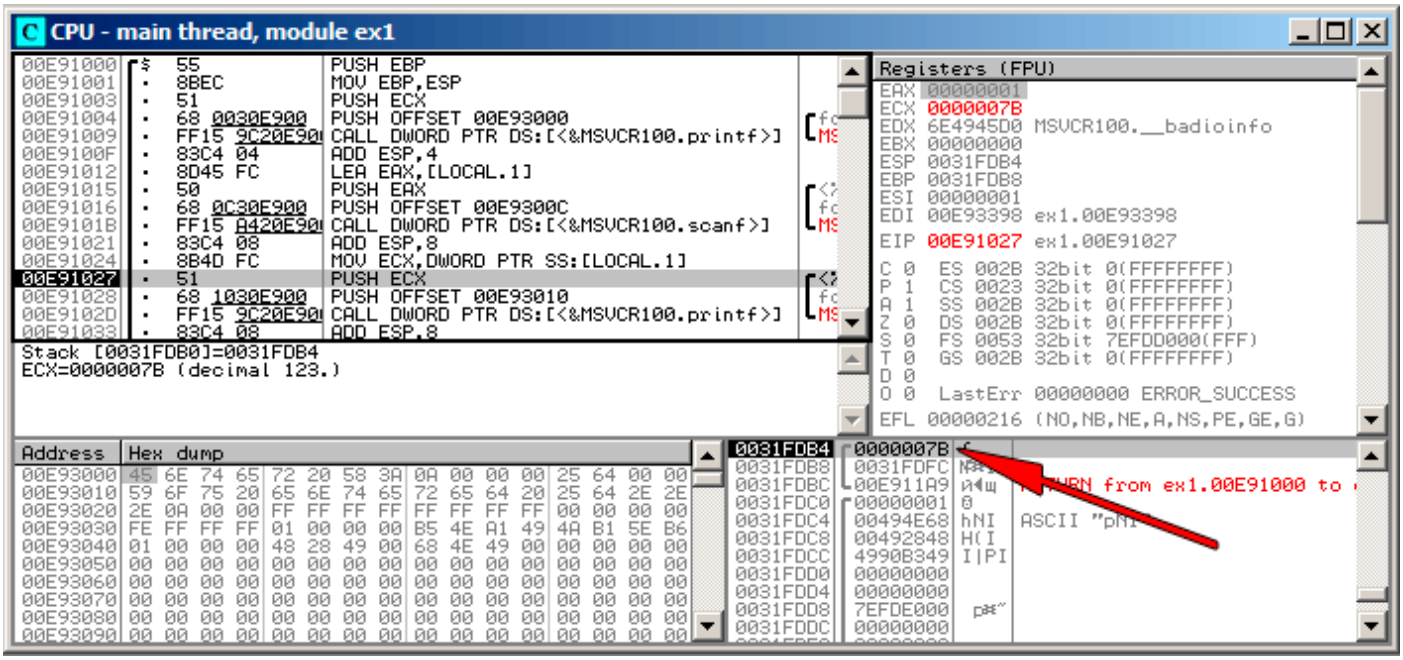


Fig. 1.14: OllyDbg : préparation de la valeur pour la passer à `printf()`

GCC

Compilons ce code avec GCC 4.4.1 sous Linux:

```
main            proc near

var_20          = dword ptr -20h
var_1C          = dword ptr -1Ch
var_4           = dword ptr -4

                push    ebp
                mov     ebp, esp
                and     esp, 0FFFFFFF0h
                sub     esp, 20h
                mov     [esp+20h+var_20], offset aEnterX ; "Enter X :"
                call    _puts
                mov     eax, offset aD  ; "%d"
                lea     edx, [esp+20h+var_4]
                mov     [esp+20h+var_1C], edx
                mov     [esp+20h+var_20], eax
                call    ___isoc99_scanf
                mov     edx, [esp+20h+var_4]
                mov     eax, offset aYouEnteredD___ ; "You entered %d...\n"
                mov     [esp+20h+var_1C], edx
                mov     [esp+20h+var_20], eax
                call    _printf
                mov     eax, 0
                leave
                retn
main            endp
```

GCC a remplacé l'appel à `printf()` avec un appel à `puts()`. La raison de cela a été expliquée dans (1.5.4 on page 22).

Comme dans l'exemple avec MSVC—les arguments sont placés dans la pile avec l'instruction `MOV`.

À propos

Ce simple exemple est la démonstration du fait que le compilateur traduit une liste d'expression en bloc-C/C++ en une liste séquentielle d'instructions. Il n'y a rien entre les expressions en C/C++, et le résultat en code machine, il n'y a rien entre le déroulement du flux de contrôle d'une expression à la suivante.

x64

Le schéma est ici similaire, avec la différence que les registres, plutôt que la pile, sont utilisés pour le passage des arguments.

MSVC

Listing 1.67: MSVC 2012 x64

```
_DATA   SEGMENT
$SG1289 DB      'Enter X :', 0aH, 00H
$SG1291 DB      '%d', 00H
$SG1292 DB      'You entered %d...', 0aH, 00H
_DATA   ENDS

_TEXT   SEGMENT
x$ = 32
main    PROC
$LN3 :
        sub     rsp, 56
        lea     rcx, OFFSET FLAT :$SG1289 ; 'Enter X :'
        call    printf
        lea     rdx, QWORD PTR x$[rsp]
        lea     rcx, OFFSET FLAT :$SG1291 ; '%d'
        call    scanf
        mov     edx, DWORD PTR x$[rsp]
        lea     rcx, OFFSET FLAT :$SG1292 ; 'You entered %d...'
        call    printf

        ; retourner 0
        xor     eax, eax
        add     rsp, 56
        ret     0
main    ENDP
_TEXT   ENDS
```

GCC

Listing 1.68: GCC 4.4.6 x64 avec optimisation

```
.LC0 :
        .string "Enter X :"
.LC1 :
        .string "%d"
.LC2 :
        .string "You entered %d...\n"

main :
        sub     rsp, 24
        mov     edi, OFFSET FLAT :.LC0 ; "Enter X :"
        call    puts
        lea     rsi, [rsp+12]
        mov     edi, OFFSET FLAT :.LC1 ; "%d"
        xor     eax, eax
        call    __isoc99_scanf
        mov     esi, DWORD PTR [rsp+12]
        mov     edi, OFFSET FLAT :.LC2 ; "You entered %d...\n"
        xor     eax, eax
        call    printf
```

```
        ; retourner 0
        xor     eax, eax
        add     rsp, 24
        ret
```

ARM

avec optimisation Keil 6/2013 (Mode Thumb)

```
.text :00000042             scanf_main
.text :00000042
.text :00000042             var_8           = -8
.text :00000042
.text :00000042 08 B5         PUSH    {R3,LR}
.text :00000044 A9 A0         ADR     R0, aEnterX  ; "Enter X :\n"
.text :00000046 06 F0 D3 F8   BL      __2printf
.text :0000004A 69 46         MOV     R1, SP
.text :0000004C AA A0         ADR     R0, aD  ; "%d"
.text :0000004E 06 F0 CD F8   BL      __0scanf
.text :00000052 00 99         LDR     R1, [SP,#8+var_8]
.text :00000054 A9 A0         ADR     R0, aYouEnteredD___  ; "You entered %d...\n"
.text :00000056 06 F0 CB F8   BL      __2printf
.text :0000005A 00 20         MOVS    R0, #0
.text :0000005C 08 BD         POP     {R3,PC}
```

Afin que `scanf()` puisse lire l'item, elle a besoin d'un paramètre—un pointeur sur un *int*. Le type *int* est 32-bit, donc nous avons besoin de 4 octets pour le stocker quelque part en mémoire, et il tient exactement dans un registre 32-bit. De l'espace pour la variable locale x est allouée sur la pile et IDA l'a nommée *var_8*. Il n'est toutefois pas nécessaire de définir cette macro, puisque le SP (pointeur de pile) pointe déjà sur cet espace et peut être utilisé directement.

Doc, la valeur de SP est copiée dans la registre R1 et, avec la chaîne de format, passée à `scanf()`. Plus tard, avec l'aide de l'instruction LDR, cette valeur est copiée depuis la pile vers le registre R1 afin de la passer à `printf()`.

ARM64

Listing 1.69: GCC 4.9.1 ARM64 sans optimisation

```
1  .LC0 :
2          .string "Enter X :"
3  .LC1 :
4          .string "%d"
5  .LC2 :
6          .string "You entered %d...\n"
7  scanf_main :
8  ; soustraire 32 de SP, puis sauver FP et LR dans la structure de pile :
9          stp     x29, x30, [sp, -32]!
10 ; utiliser la partie de pile (FP=SP)
11         add     x29, sp, 0
12 ; charger le pointeur sur la chaîne  "Enter X :":
13         adrp    x0, .LC0
14         add     x0, x0,  :lo12 :.LC0
15 ; X0=pointeur sur la chaîne "Enter X :"
16 ; l'afficher :
17         bl      puts
18 ; charger le pointeur sur la chaîne "%d":
19         adrp    x0, .LC1
20         add     x0, x0,  :lo12 :.LC1
21 ; trouver de l'espace dans la structure de pile pour la variable "x" (X1=FP+28) :
22         add     x1, x29, 28
23 ; X1=adresse de la variable "x"
24 ; passer l'adresse  de scanf() et l'appeler :
25         bl      __isoc99_scanf
26 ; charger la valeur 32-bit de la variable dans la partie de pile :
```

```
27          ldr     w1, [x29,28]
28  ; W1=x
29  ; charger le pointeur sur la chaîne "You entered %d...\n"
30  ; printf() va prendre la chaîne de texte de X0 et de la variable "x" de X1 (ou W1)
31          adrp    x0, .LC2
32          add     x0, x0,  :lo12 :.LC2
33          bl      printf
34  ; retourner 0
35          mov     w0, 0
36  ; restaurer FP et LR, puis ajouter 32 à SP:
37          ldp     x29, x30, [sp], 32
38          ret
```

Il y a 32 octets alloués pour la structure de pile, ce qui est plus que nécessaire. Peut-être dans un soucis d'alignement de mémoire? La partie la plus intéressante est de trouver de l'espace pour la variable x dans la structure de pile (ligne 22). Pourquoi 28? Pour une certaine raison, le compilateur a décidé de stocker cette variable à la fin de la structure de pile locale au lieu du début. L'adresse est passée à `scanf()`, qui stocke l'entrée de l'utilisateur en mémoire à cette adresse. Il s'agit d'une valeur sur 32-bit de type *int*. La valeur est prise à la ligne 27 puis passée à `printf()`.

MIPS

Une place est allouée sur la pile locale pour la variable x, et elle doit être appelée par $\$sp + 24$.

Son adresse est passée à `scanf()`, et la valeur entrée par l'utilisateur est chargée en utilisant l'instruction `LW` (« Load Word »), puis passée à `printf()`.

Listing 1.70: GCC 4.4.5 avec optimisation (résultat en sortie de l'assembleur)

```
$LC0 :
        .ascii  "Enter X :\000"
$LC1 :
        .ascii  "%d\000"
$LC2 :
        .ascii  "You entered %d...\012\000"
main :
; prologue de la fonction :
        lui     $28,%hi(__gnu_local_gp)
        addiu   $sp,$sp,-40
        addiu   $28,$28,%lo(__gnu_local_gp)
        sw      $31,36($sp)
; appel de puts() :
        lw      $25,%call16(puts)($28)
        lui     $4,%hi($LC0)
        jalr    $25
        addiu   $4,$4,%lo($LC0) ; slot de délai de branchement
; appel de scanf() :
        lw      $28,16($sp)
        lui     $4,%hi($LC1)
        lw      $25,%call16(__isoc99_scanf)($28)
; définir le 2nd argument de scanf(), $a1=$sp+24:
        addiu   $5,$sp,24
        jalr    $25
        addiu   $4,$4,%lo($LC1) ; slot de délai de branchement

; appel de printf() :
        lw      $28,16($sp)
; définir le 2nd argument de printf(),
; charger un mot à l'adresse $sp+24:
        lw      $5,24($sp)
        lw      $25,%call16(printf)($28)
        lui     $4,%hi($LC2)
        jalr    $25
        addiu   $4,$4,%lo($LC2) ; slot de délai de branchement

; épilogue de la fonction:
        lw      $31,36($sp)
; mettre la valeur de retour à 0:
        move    $2,$0
```

```
; retourner :
        j       $31
        addiu   $sp,$sp,40      ; slot de délai de branchement
```

IDA affiche la disposition de la pile comme suit:

Listing 1.71: GCC 4.4.5 avec optimisation (IDA)

```
.text :00000000 main :
.text :00000000
.text :00000000 var_18   = -0x18
.text :00000000 var_10   = -0x10
.text :00000000 var_4    = -4
.text :00000000
; prologue de la fonction :
.text :00000000          lui     $gp, (__gnu_local_gp >> 16)
.text :00000004          addiu   $sp, -0x28
.text :00000008          la      $gp, (__gnu_local_gp & 0xFFFF)
.text :0000000C          sw      $ra, 0x28+var_4($sp)
.text :00000010          sw      $gp, 0x28+var_18($sp)
; appel de puts() :
.text :00000014          lw      $t9, (puts & 0xFFFF)($gp)
.text :00000018          lui     $a0, ($LC0 >> 16)  # "Enter X :"
.text :0000001C          jalr    $t9
.text :00000020          la      $a0, ($LC0 & 0xFFFF)  # "Enter X :" ; slot de délai de
    branchement
; appel de scanf() :
.text :00000024          lw      $gp, 0x28+var_18($sp)
.text :00000028          lui     $a0, ($LC1 >> 16)  # "%d"
.text :0000002C          lw      $t9, (__isoc99_scanf & 0xFFFF)($gp)
; définir le 2nd argument de scanf(), $a1=$sp+24:
.text :00000030          addiu   $a1, $sp, 0x28+var_10
.text :00000034          jalr    $t9  ; slot de délai de branchement
.text :00000038          la      $a0, ($LC1 & 0xFFFF)  # "%d"
; appel de printf() :
.text :0000003C          lw      $gp, 0x28+var_18($sp)
; définir le 2nd argument de printf(),
; charger un mot à l'adresse $sp+24:
.text :00000040          lw      $a1, 0x28+var_10($sp)
.text :00000044          lw      $t9, (printf & 0xFFFF)($gp)
.text :00000048          lui     $a0, ($LC2 >> 16)  # "You entered %d...\n"
.text :0000004C          jalr    $t9
.text :00000050          la      $a0, ($LC2 & 0xFFFF)  # "You entered %d...\n" ; slot de délai
    de branchement
; épilogue de la fonction:
.text :00000054          lw      $ra, 0x28+var_4($sp)
; mettre la valeur de retour à 0:
.text :00000058          move    $v0, $zero
; retourner :
.text :0000005C          jr      $ra
.text :00000060          addiu   $sp, 0x28 ; slot de délai de branchement
```

1.9.2 Erreur courante

C'est une erreur très courante (et/ou une typo) de passer la valeur de *x* au lieu d'un pointeur sur *x* :

```
#include <stdio.h>

int main()
{
        int x;
        printf ("Enter X :\n");

        scanf ("%d", x); // BUG

        printf ("You entered %d...\n", x);

        return 0;
};
```

Donc que se passe-t-il ici? *x* n'est pas non-initialisée et contient des données aléatoires de la pile locale. Lorsque `scanf()` est appelée, elle prend la chaîne de l'utilisateur, la convertit en nombre et essaye de l'écrire dans *x*, la considérant comme une adresse en mémoire. Mais il s'agit de bruit aléatoire, donc `scanf()` va essayer d'écrire à une adresse aléatoire. Très probablement, le processus va planter.

Assez intéressant, certaines bibliothèques CRT compilées en debug, mettent un signe distinctif lors de l'allocation de la mémoire, comme 0xCCCCCCCC ou 0x0BADF00D etc. Dans ce cas, *x* peut contenir 0xCCCCCCCC, et `scanf()` va essayer d'écrire à l'adresse 0xCCCCCCCC. Et si vous remarquez que quelque chose dans votre processus essaye d'écrire à l'adresse 0xCCCCCCCC, vous saurez qu'une variable non initialisée (ou un pointeur) a été utilisée sans initialisation préalable. C'est mieux que si la mémoire nouvellement allouée est juste mise à zéro.

1.9.3 Variables globales

Que se passe-t-il si la variable x de l'exemple précédent n'est pas locale mais globale ? Alors, elle sera accessible depuis n'importe quel point, plus seulement depuis le corps de la fonction. Les variables globales sont considérées comme un anti-pattern, mais dans un but d'expérience, nous pouvons le faire.

```
#include <stdio.h>

// maintenant x est une variable globale
int x;

int main()
{
        printf ("Enter X :\n");

        scanf ("%d", &x);

        printf ("You entered %d...\n", x);

        return 0;
};
```

MSVC: x86

```
_DATA    SEGMENT
COMM    _x :DWORD
$SG2456    DB    'Enter X :', 0aH, 00H
$SG2457    DB    '%d', 00H
$SG2458    DB    'You entered %d...', 0aH, 00H
_DATA    ENDS
PUBLIC    _main
EXTRN    _scanf :PROC
EXTRN    _printf :PROC
; Function compile flags : /Odtp
_TEXT    SEGMENT
_main    PROC
    push   ebp
    mov    ebp, esp
    push   OFFSET $SG2456
    call   _printf
    add    esp, 4
    push   OFFSET _x
    push   OFFSET $SG2457
    call   _scanf
    add    esp, 8
    mov    eax, DWORD PTR _x
    push   eax
    push   OFFSET $SG2458
    call   _printf
    add    esp, 8
    xor    eax, eax
    pop    ebp
    ret    0
_main    ENDP
```

```
_TEXT    ENDS
```

Dans ce cas, la variable x est définie dans la section _DATA et il n'y a pas de mémoire allouée sur la pile locale. Elle est accédée directement, pas par la pile. Les variables globales non initialisées ne prennent pas de place dans le fichier exécutable (en effet, pourquoi aurait-on besoin d'allouer de l'espace pour des variables initialement mises à zéro ?), mais lorsque quelqu'un accède à leur adresse, l'OS va y allouer un bloc de zéros[73].

Maintenant, assignons explicitement une valeur à la variable:

```
int x=10; // valeur par défaut
```

Nous obtenons:

```
_DATA   SEGMENT
_x      DD      0aH

...
```

Ici nous voyons une valeur 0xA de type DWORD (DD signifie DWORD = 32 bit) pour cette variable.

Si vous ouvrez le .exe compilé dans IDA, vous pouvez voir la variable *x* placée au début du segment _DATA, et après elle vous pouvez voir la chaîne de texte.

Si vous ouvrez le .exe compilé de l'exemple précédent dans IDA, où la valeur de *x* n'était pas mise, vous verrez quelque chose comme ça:

Listing 1.72: IDA

```
.data :0040FA80 _x              dd ?    ; DATA XREF : _main+10
.data :0040FA80                         ; _main+22
.data :0040FA84 dword_40FA84    dd ?    ; DATA XREF : _memset+1E
.data :0040FA84                         ; unknown_libname_1+28
.data :0040FA88 dword_40FA88    dd ?    ; DATA XREF : ___sbh_find_block+5
.data :0040FA88                         ; ___sbh_free_block+2BC
.data :0040FA8C ; LPVOID lpMem
.data :0040FA8C lpMem           dd ?    ; DATA XREF : ___sbh_find_block+B
.data :0040FA8C                         ; ___sbh_free_block+2CA
.data :0040FA90 dword_40FA90    dd ?    ; DATA XREF : _V6_HeapAlloc+13
.data :0040FA90                         ; __calloc_impl+72
.data :0040FA94 dword_40FA94    dd ?    ; DATA XREF : ___sbh_free_block+2FE
```

_x est marquée avec ? avec le reste des variables qui ne doivent pas être initialisées. Ceci implique qu'après avoir chargé le .exe en mémoire, de l'espace pour toutes ces variables doit être alloué et rempli avec des zéros [*ISO/IEC 9899:TC3 (C C99 standard)*, (2007)6.7.8p10]. Mais dans le fichier .exe, ces variables non initialisées n'occupent rien du tout. C'est pratique pour les gros tableaux, par exemple.

[73] C'est comme ça que se comportent les VM

MSVC: x86 + OllyDbg

Les choses sont encore plus simple ici:

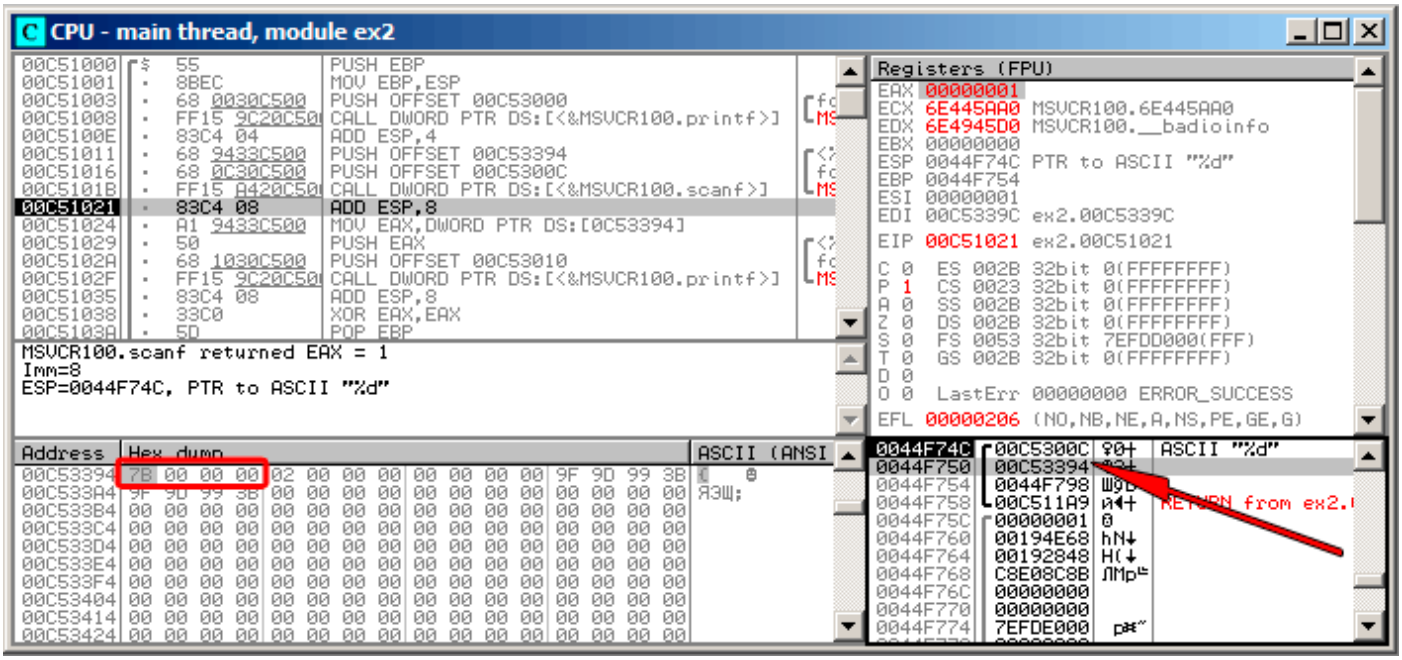


Fig. 1.15: OllyDbg : après l'exécution de `scanf()`

La variable se trouve dans le segment de données. Après que l'instruction PUSH (pousser l'adresse de x) ait été exécutée, l'adresse apparaît dans la fenêtre de la pile. Cliquer droit sur cette ligne et choisir « Follow in dump ». La variable va apparaître dans la fenêtre de la mémoire sur la gauche. Après que nous ayons entré 123 dans la console, `0x7B` apparaît dans la fenêtre de la mémoire (voir les régions surlignées dans la copie d'écran).

Mais pourquoi est-ce que le premier octet est `7B`? Logiquement, Il devrait y avoir `00 00 00 7B` ici. La cause de ceci est référé comme endianness, et x86 utilise *little-endian*. Cela implique que l'octet le plus faible poids est écrit en premier, et le plus fort en dernier. Voir à ce propos: 2.8 on page 468. Revenons à l'exemple, la valeur 32-bit est chargée depuis son adresse mémoire dans EAX et passée à `printf()`.

L'adresse mémoire de x est `0x00C53394`.

Dans OllyDbg nous pouvons examiner l'espace mémoire du processus (Alt-M) et nous pouvons voir que cette adresse se trouve dans le segment PE `.data` de notre programme:

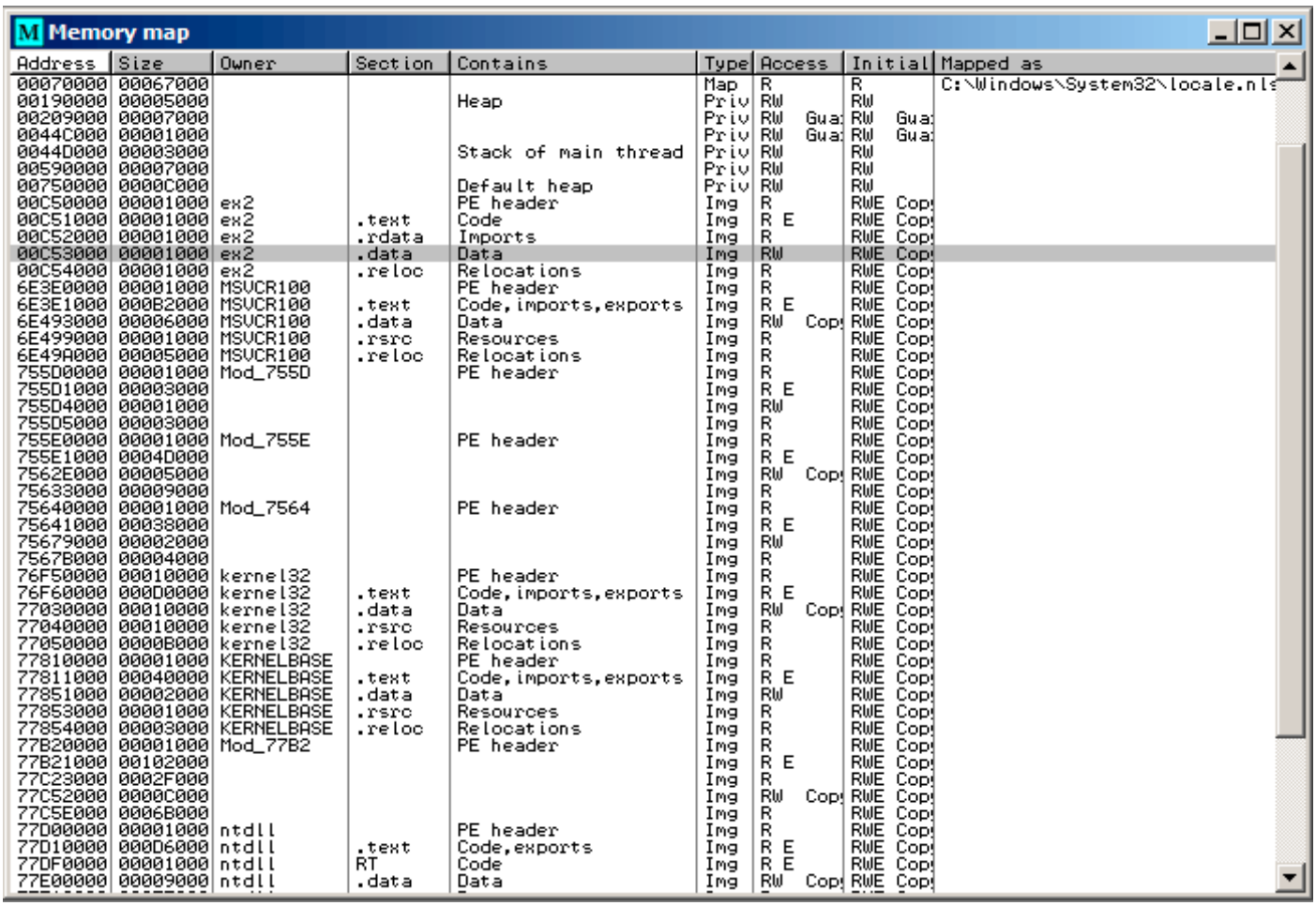


Fig. 1.16: OllyDbg : espace mémoire du processus

GCC: x86

Le schéma sous Linux est presque le même, avec la différence que les variables non initialisées se trouvent dans le segment `_bss`. Dans un fichier ELF[74] ce segment possède les attributs suivants:

```
; Segment type : Uninitialized
; Segment permissions : Read/Write
```

Si toutefois vous initialisez la variable avec une valeur quelconque, e.g. 10, elle sera placée dans le segment `_data`, qui possède les attributs suivants:

```
; Segment type : Pure data
; Segment permissions : Read/Write
```

MSVC: x64

Listing 1.73: MSVC 2012 x64

```
_DATA   SEGMENT
COMM    x :DWORD
$SG2924 DB      'Enter X :', 0aH, 00H
$SG2925 DB      '%d', 00H
$SG2926 DB      'You entered %d...', 0aH, 00H
_DATA   ENDS
```

[74] Format de fichier exécutable couramment utilisé sur les systèmes *NIX, Linux inclus

```
_TEXT   SEGMENT
main    PROC
$LN3 :
        sub     rsp, 40

        lea     rcx, OFFSET FLAT :$SG2924 ; 'Enter X :'
        call    printf
        lea     rdx, OFFSET FLAT :x
        lea     rcx, OFFSET FLAT :$SG2925 ; '%d'
        call    scanf
        mov     edx, DWORD PTR x
        lea     rcx, OFFSET FLAT :$SG2926 ; 'You entered %d...'
        call    printf

        ; retourner 0
        xor     eax, eax

        add     rsp, 40
        ret     0
main    ENDP
_TEXT   ENDS
```

Le code est presque le même qu'en x86. Notez toutefois que l'adresse de la variable x est passée à `scanf()` en utilisant une instruction `LEA`, tandis que la valeur de la variable est passée au second `printf()` en utilisant une instruction `MOV`. `DWORD PTR`—fait partie du langage d'assemblage (aucune relation avec le code machine), indique que la taille de la variable est 32-bit et que l'instruction `MOV` doit être encodée en conséquence.

ARM: avec optimisation Keil 6/2013 (Mode Thumb)

Listing 1.74: IDA

```
.text :00000000 ; Segment type : Pure code
.text :00000000         AREA .text, CODE
...
.text :00000000 main
.text :00000000         PUSH    {R4,LR}
.text :00000002         ADR     R0, aEnterX  ; "Enter X :\n"
.text :00000004         BL      __2printf
.text :00000008         LDR     R1, =x
.text :0000000A         ADR     R0, aD       ; "%d"
.text :0000000C         BL      __0scanf
.text :00000010         LDR     R0, =x
.text :00000012         LDR     R1, [R0]
.text :00000014         ADR     R0, aYouEnteredD___ ; "You entered %d...\n"
.text :00000016         BL      __2printf
.text :0000001A         MOVS    R0, #0
.text :0000001C         POP     {R4,PC}
...
.text :00000020 aEnterX DCB "Enter X :",0xA,0 ; DATA XREF : main+2
.text :0000002A         DCB    0
.text :0000002B         DCB    0
.text :0000002C off_2C  DCD x                ; DATA XREF : main+8
.text :0000002C                              ; main+10
.text :00000030 aD      DCB "%d",0           ; DATA XREF : main+A
.text :00000033         DCB    0
.text :00000034 aYouEnteredD___ DCB "You entered %d...",0xA,0 ; DATA XREF : main+14
.text :00000047         DCB 0
.text :00000047 ; .text ends
.text :00000047
...
.data :00000048 ; Segment type : Pure data
.data :00000048         AREA .data, DATA
.data :00000048          ; ORG 0x48
.data :00000048         EXPORT x
.data :00000048 x       DCD 0xA              ; DATA XREF : main+8
.data :00000048                              ; main+10
```

```
.data :00000048 ; .data ends
```

Donc, la variable x est maintenant globale, et pour cette raison, elle se trouve dans un autre segment, appelé le segment de données (*.data*). On pourrait demander pour quoi les chaînes de textes sont dans le segment de code (*.text*) et x là. C'est parque c'est une variable et que par définition sa valeur peut changer. En outre, elle peut même changer souvent. Alors que les chaînes de texte ont un type constant, elles ne changent pas, donc elles sont dans le segment *.text*.

Le segment de code peut parfois se trouver dans la ROM[75] d'un circuit (gardez à l'esprit que nous avons maintenant affaire avec de l'électronique embarquée, et que la pénurie de mémoire y est courante), et les variables —en RAM.

Il n'est pas très économique de stocker des constantes en RAM quand vous avez de la ROM.

En outre, les variables en RAM doivent être initialisées, car après le démarrage, la RAM, évidemment, contient des données aléatoires.

En avançant, nous voyons un pointeur sur la variable x (`off_2C`) dans le segment de code, et que toutes les opérations avec cette variable s'effectuent via ce pointeur.

Car la variable x peut se trouver loin de ce morceau de code, donc son adresse doit être sauvée proche du code.

L'instruction LDR en mode Thumb ne peut adresser des variables que dans un intervalle de 1020 octets de son emplacement.

et en mode ARM —l'intervalle des variables est de ± 4095 octets.

Et donc l'adresse de la variable x doit se trouver quelque part de très proche, car il n'y a pas de garantie que l'éditeur de liens pourra stocker la variable proche du code, elle peut même se trouver sur un module de mémoire externe.

Encore une chose: si une variable est déclarée comme *const*, le compilateur Keil va l'allouer dans le segment `.constdata`.

Peut-être qu'après, l'éditeur de liens mettra ce segment en ROM aussi, à côté du segment de code.

ARM64

Listing 1.75: GCC 4.9.1 ARM64 sans optimisation

```
1         .comm   x,4,4
2 .LC0 :
3         .string "Enter X :"
4 .LC1 :
5         .string "%d"
6 .LC2 :
7         .string "You entered %d...\n"
8 f5 :
9 ; sauver FP et LR dans la structure de pile locale :
10        stp     x29, x30, [sp, -16]!
11 ; définir la pile locale (FP=SP)
12        add     x29, sp, 0
13 ; charger le pointeur sur la chaîne "Enter X :":
14        adrp    x0, .LC0
15        add     x0, x0,  :lo12 :.LC0
16        bl      puts
17 ; charger le pointeur sur la chaîne "%d":
18        adrp    x0, .LC1
19        add     x0, x0,  :lo12 :.LC1
20 ; générer l'adresse de la variable globale x:
21        adrp    x1, x
22        add     x1, x1,  :lo12 :x
23        bl      __isoc99_scanf
24 ; générer à nouveau l'adresse de la variable globale x:
25        adrp    x0, x
26        add     x0, x0,  :lo12 :x
27 ; charger la valeur de la mémoire à cette adresse:
28        ldr     w1, [x0]
```

[75] Mémoire morte

```
29 ; charger le pointeur sur la chaîne "You entered %d...\n":
30         adrp    x0, .LC2
31         add     x0, x0,  :lo12 :.LC2
32         bl      printf
33 ; retourner 0
34         mov     w0, 0
35 ; restaurer FP et LR :
36         ldp     x29, x30, [sp], 16
37         ret
```

Dans ce car la variable x est déclarée comme étant globale et son adresse est calculée en utilisant la paire d'instructions ADRP/ADD (lignes 21 et 25).

MIPS

Variable globale non initialisée

Donc maintenant, la variable x est globale. Compilons en un exécutable plutôt qu'un fichier objet et chargeons-le dans IDA. IDA affiche la variable x dans la section ELF `.sbss` (vous vous rappelez du « Pointeur Global » ? 1.5.5 on page 25), puisque cette variable n'est pas initialisée au début.

Listing 1.76: GCC 4.4.5 avec optimisation (IDA)

```
.text :004006C0 main :
.text :004006C0
.text :004006C0 var_10          = -0x10
.text :004006C0 var_4           = -4
.text :004006C0
; prologue de la fonction :
.text :004006C0                 lui     $gp, 0x42
.text :004006C4                 addiu   $sp, -0x20
.text :004006C8                 li      $gp, 0x418940
.text :004006CC                 sw      $ra, 0x20+var_4($sp)
.text :004006D0                 sw      $gp, 0x20+var_10($sp)
; appel de puts() :
.text :004006D4                 la      $t9, puts
.text :004006D8                 lui     $a0, 0x40
.text :004006DC                 jalr    $t9 ; puts
.text :004006E0                 la      $a0, aEnterX     # "Enter X :" ; slot de délai de
    branchement
; appel de scanf() :
.text :004006E4                 lw      $gp, 0x20+var_10($sp)
.text :004006E8                 lui     $a0, 0x40
.text :004006EC                 la      $t9, __isoc99_scanf
; préparer l'adresse de x:
.text :004006F0                 la      $a1, x
.text :004006F4                 jalr    $t9 ; __isoc99_scanf
.text :004006F8                 la      $a0, aD          # "%d"       ; slot de délai de
    branchement
; appel de printf() :
.text :004006FC                 lw      $gp, 0x20+var_10($sp)
.text :00400700                 lui     $a0, 0x40
; prendre l'adresse de x :
.text :00400704                 la      $v0, x
.text :00400708                 la      $t9, printf
; charger la valeur de la variable "x" et la passer à printf() dans $a1 :
.text :0040070C                 lw      $a1, (x - 0x41099C)($v0)
.text :00400710                 jalr    $t9 ; printf
.text :00400714                 la      $a0, aYouEnteredD___  # "You entered %d...\n" ; slot de
    délai de branchement
; épilogue de la fonction:
.text :00400718                 lw      $ra, 0x20+var_4($sp)
.text :0040071C                 move    $v0, $zero
.text :00400720                 jr      $ra
.text :00400724                 addiu   $sp, 0x20  ; slot de délai de branchement

...

.sbss :0041099C  # Type de segment: Non initialisé
```

```
.sbss :0041099C                 .sbss
.sbss :0041099C                 .globl x
.sbss :0041099C x :              .space 4
.sbss :0041099C
```

IDA réduit le volume des informations, donc nous allons générer un listing avec objdump et le commenter:

Listing 1.77: GCC 4.4.5 avec optimisation (objdump)

```
1  004006c0 <main> :
2  ; prologue de la fonction :
3    4006c0 :       3c1c0042        lui     gp,0x42
4    4006c4 :       27bdffe0        addiu   sp,sp,-32
5    4006c8 :       279c8940        addiu   gp,gp,-30400
6    4006cc :       afbf001c        sw      ra,28(sp)
7    4006d0 :       afbc0010        sw      gp,16(sp)
8  ; appel de puts() :
9    4006d4 :       8f998034        lw      t9,-32716(gp)
10   4006d8 :       3c040040        lui     a0,0x40
11   4006dc :       0320f809        jalr    t9
12   4006e0 :       248408f0        addiu   a0,a0,2288 ; slot de délai de branchement
13 ; appel de scanf() :
14   4006e4 :       8fbc0010        lw      gp,16(sp)
15   4006e8 :       3c040040        lui     a0,0x40
16   4006ec :       8f998038        lw      t9,-32712(gp)
17 ; préparer l'adresse de x:
18   4006f0 :       8f858044        lw      a1,-32700(gp)
19   4006f4 :       0320f809        jalr    t9
20   4006f8 :       248408fc        addiu   a0,a0,2300 ; slot de délai de branchement
21 ; appel de printf() :
22   4006fc :       8fbc0010        lw      gp,16(sp)
23   400700:       3c040040        lui     a0,0x40
24 ; prendre l'adresse de x :
25   400704:       8f828044        lw      v0,-32700(gp)
26   400708:       8f99803c        lw      t9,-32708(gp)
27 ; charger la valeur de la variable "x" et la passer à printf() dans $a1 :
28   40070c :       8c450000        lw      a1,0(v0)
29   400710:       0320f809        jalr    t9
30   400714:       24840900        addiu   a0,a0,2304 ; slot de délai de branchement
31 ; épilogue de la fonction:
32   400718:       8fbf001c        lw      ra,28(sp)
33   40071c :       00001021        move    v0,zero
34   400720:       03e00008        jr      ra
35   400724:       27bd0020        addiu   sp,sp,32   ; slot de délai de branchement
36 ; groupe de NOPs servant à aligner la prochaine fonction sur un bloc de 16-octet:
37   400728:       00200825        move    at,at
38   40072c :       00200825        move    at,at
```

Nous voyons maintenant que l'adresse de la variable x est lue depuis un buffer de 64KiB en utilisant GP et en lui ajoutant un offset négatif (ligne 18). Plus que ça, les adresses des trois fonctions externes qui sont utilisées dans notre exemple (`puts()`, `scanf()`, `printf()`), sont aussi lues depuis le buffer de données globale en utilisant GP (lignes 9, 16 et 26). GP pointe sur le milieu du buffer, et de tels offsets suggèrent que les adresses des trois fonctions, et aussi l'adresse de la variable x, sont toutes stockées quelque part au début du buffer. Cela fait du sens, car notre exemple est minuscule.

Une autre chose qui mérite d'être mentionnée est que la fonction se termine avec deux NOPs (`MOVE $AT,$AT` — une instruction sans effet), afin d'aligner le début de la fonction suivante sur un bloc de 16-octet.

Variable globale initialisée

Modifions notre exemple en affectant une valeur par défaut à la variable x :

```
int x=10; // valeur par défaut
```

Maintenant IDA montre que la variable x se trouve dans la section .data:

Listing 1.78: GCC 4.4.5 avec optimisation (IDA)

```
.text :004006A0 main :
.text :004006A0
.text :004006A0 var_10          = -0x10
.text :004006A0 var_8           = -8
.text :004006A0 var_4           = -4
.text :004006A0
.text :004006A0                 lui     $gp, 0x42
.text :004006A4                 addiu   $sp, -0x20
.text :004006A8                 li      $gp, 0x418930
.text :004006AC                 sw      $ra, 0x20+var_4($sp)
.text :004006B0                 sw      $s0, 0x20+var_8($sp)
.text :004006B4                 sw      $gp, 0x20+var_10($sp)
.text :004006B8                 la      $t9, puts
.text :004006BC                 lui     $a0, 0x40
.text :004006C0                 jalr    $t9 ; puts
.text :004006C4                 la      $a0, aEnterX     # "Enter X :"
.text :004006C8                 lw      $gp, 0x20+var_10($sp)
; préparer la partie haute de l'adresse de x:
.text :004006CC                 lui     $s0, 0x41
.text :004006D0                 la      $t9, __isoc99_scanf
.text :004006D4                 lui     $a0, 0x40
; et ajouter la partie basse de l'adresse de x :
.text :004006D8                 addiu   $a1, $s0, (x - 0x410000)
; maintenant l'adresse de x est dans $a1.
.text :004006DC                 jalr    $t9 ; __isoc99_scanf
.text :004006E0                 la      $a0, aD          # "%d"
.text :004006E4                 lw      $gp, 0x20+var_10($sp)
; prendre un mot dans la mémoire:
.text :004006E8                 lw      $a1, x
; la valeur de x est maintenant dans $a1.
.text :004006EC                 la      $t9, printf
.text :004006F0                 lui     $a0, 0x40
.text :004006F4                 jalr    $t9 ; printf
.text :004006F8                 la      $a0, aYouEnteredD___  # "You entered %d...\n"
.text :004006FC                 lw      $ra, 0x20+var_4($sp)
.text :00400700                 move    $v0, $zero
.text :00400704                 lw      $s0, 0x20+var_8($sp)
.text :00400708                 jr      $ra
.text :0040070C                 addiu   $sp, 0x20

...

.data :00410920                 .globl x
.data :00410920 x :              .word 0xA
```

Pourquoi pas .sdata? Peut-être que cela dépend d'une option de GCC?

Néanmoins, maintenant x est dans .data, qui une zone mémoire générale, et nous pouvons regarder comment y travailler avec des variables.

L'adresse de la variable doit être formée en utilisant une paire d'instructions.

Dans notre cas, ce sont `LUI` (« Load Upper Immediate ») et `ADDIU` (« Add Immediate Unsigned Word »).

Voici le listing d'objdump pour y regarder de plus près:

Listing 1.79: GCC 4.4.5 avec optimisation (objdump)

```
004006a0 <main> :
  4006a0 :       3c1c0042        lui     gp,0x42
  4006a4 :       27bdffe0        addiu   sp,sp,-32
  4006a8 :       279c8930        addiu   gp,gp,-30416
  4006ac :       afbf001c        sw      ra,28(sp)
  4006b0 :       afb00018        sw      s0,24(sp)
  4006b4 :       afbc0010        sw      gp,16(sp)
  4006b8 :       8f998034        lw      t9,-32716(gp)
  4006bc :       3c040040        lui     a0,0x40
  4006c0 :       0320f809        jalr    t9
  4006c4 :       248408d0        addiu   a0,a0,2256
  4006c8 :       8fbc0010        lw      gp,16(sp)
```

```
; préparer la partie haute de l'adresse de x:
  4006cc :       3c100041        lui     s0,0x41
  4006d0 :       8f998038        lw      t9,-32712(gp)
  4006d4 :       3c040040        lui     a0,0x40
; ajouter la partie basse de l'adresse de x :
  4006d8 :       26050920        addiu   a1,s0,2336
; maintenant l'adresse de x est dans $a1.
  4006dc :       0320f809        jalr    t9
  4006e0 :       248408dc        addiu   a0,a0,2268
  4006e4 :       8fbc0010        lw      gp,16(sp)
; la partie haute de l'adresse de x est toujours dans $s0.
; lui ajouter la partie basse et charger un mot de la mémoire:
  4006e8 :       8e050920        lw      a1,2336(s0)
; la valeur de x est maintenant dans $a1.
  4006ec :       8f99803c        lw      t9,-32708(gp)
  4006f0 :       3c040040        lui     a0,0x40
  4006f4 :       0320f809        jalr    t9
  4006f8 :       248408e0        addiu   a0,a0,2272
  4006fc :       8fbf001c        lw      ra,28(sp)
  400700:       00001021        move    v0,zero
  400704:       8fb00018        lw      s0,24(sp)
  400708:       03e00008        jr      ra
  40070c :       27bd0020        addiu   sp,sp,32
```

Nous voyons que l'adresse est formée en utilisant LUI et ADDIU, mais la partie haute de l'adresse est toujours dans le registre $S0, et il est possible d'encoder l'offset en une instruction LW (« Load Word »), donc une seule instruction LW est suffisante pour charger une valeur de la variable et la passer à `printf()`.

Les registres contenant des données temporaires sont préfixés avec T-, mais ici nous en voyons aussi qui sont préfixés par S-, leur contenu doit être doit être sauvegardé quelque part avant de les utiliser dans d'autres fonctions.

C'est pourquoi la valeur de $S0 a été mise à l'adresse 0x4006cc et utilisée de nouveau à l'adresse 0x4006e8, après l'appel de `scanf()`. La fonction `scanf()` ne change pas cette valeur.

1.9.4 scanf()

Comme il a déjà été écrit, il est plutôt dépassé d'utiliser `scanf()` aujourd'hui. Mais si nous devons, il faut vérifier si `scanf()` se termine correctement sans erreur.

```
#include <stdio.h>

int main()
{
        int x;
        printf ("Enter X :\n");

        if (scanf ("%d", &x)==1)
                printf ("You entered %d...\n", x);
        else
                printf ("What you entered? Huh?\n");

        return 0;
};
```

Par norme, la fonction `scanf()`[76] renvoie le nombre de champs qui ont été lus avec succès.

Dans notre cas, si tout se passe bien et que l'utilisateur entre un nombre `scanf()` renvoie 1, ou en cas d'erreur (ou EOF[77]) — 0.

Ajoutons un peu de code C pour vérifier la valeur de retour de `scanf()` et afficher un message d'erreur en cas d'erreur.

Cela fonctionne comme attendu:

[76] scanf, wscanf: MSDN
[77] End of File (fin de fichier)

```
C :\...>ex3.exe
Enter X :
123
You entered 123...

C :\...>ex3.exe
Enter X :
ouch
What you entered? Huh?
```

MSVC: x86

Voici ce que nous obtenons dans la sortie assembleur (MSVC 2010):

```
        lea     eax, DWORD PTR _x$[ebp]
        push    eax
        push    OFFSET $SG3833 ; '%d', 00H
        call    _scanf
        add     esp, 8
        cmp     eax, 1
        jne     SHORT $LN2@main
        mov     ecx, DWORD PTR _x$[ebp]
        push    ecx
        push    OFFSET $SG3834 ; 'You entered %d...', 0aH, 00H
        call    _printf
        add     esp, 8
        jmp     SHORT $LN1@main
$LN2@main :
        push    OFFSET $SG3836 ; 'What you entered? Huh?', 0aH, 00H
        call    _printf
        add     esp, 4
$LN1@main :
        xor     eax, eax
```

La fonction appelante (`main()`) à besoin du résultat de la fonction appelée, donc la fonction appelée le renvoie dans la registre EAX.

Nous le vérifions avec l'aide de l'instruction `CMP EAX, 1` (*CoMPare*). En d'autres mots, nous comparons la valeur dans le registre EAX avec 1.

Une instruction de saut conditionnelle `JNE` suit l'instruction `CMP`. `JNE` signifie *Jump if Not Equal* (saut si non égal).

Donc, si la valeur dans le registre EAX n'est pas égale à 1, le CPU va poursuivre l'exécution à l'adresse mentionnée dans l'opérande `JNE`, dans notre cas `$LN2@main`. Passez le contrôle à cette adresse résulte en l'exécution par le CPU de `printf()` avec l'argument `What you entered? Huh?`. Mais si tout est bon, le saut conditionnel n'est pas pris, et un autre appel à `printf()` est exécuté, avec deux arguments: `'You entered %d...'` et la valeur de x.

Puisque dans ce cas le second `printf()` n'a pas été exécuté, il y a un `JMP` qui le précède (saut inconditionnel). Il passe le contrôle au point après le second `printf()` et juste avant l'instruction `XOR EAX, EAX`, qui implémente `return 0`.

Donc, on peut dire que comparer une valeur avec une autre est *usuellement* implémenté par la paire d'instructions `CMP`/`Jcc`, où *cc* est un *code de condition*. `CMP` compare deux valeurs et met les flags[78] du processeur. `Jcc` vérifie ces flags et décide de passer le contrôle à l'adresse spécifiée ou non.

Cela peut sembler paradoxal, mais l'instruction `CMP` est en fait un `SUB` (soustraction). Toutes les instructions arithmétiques mettent les flags du processeur, pas seulement `CMP`. Si nous comparons 1 et 1, $1-1$ donne 0 donc le flag ZF va être mis (signifiant que le dernier résultat est 0). Dans aucune autre circonstance ZF ne sera mis, à l'exception que les opérandes ne soient égaux. `JNE` vérifie seulement le flag ZF et saute seulement si il n'est pas mis. `JNE` est un synonyme pour `JNZ` (*Jump if Not Zero* (saut si non zéro)). L'assembleur génère le même opcode pour les instructions `JNE` et `JNZ`. Donc, l'instruction `CMP` peut être remplacée par une instruction `SUB` et presque tout ira bien, à la différence que `SUB` altère la valeur du premier opérande. `CMP` est un *SUB sans sauver le résultat, mais modifiant les flags*.

[78] flags x86, voir aussi: Wikipédia.

MSVC: x86: IDA

C'est le moment de lancer IDA et d'essayer de faire quelque chose avec. À propos, pour les débutants, c'est une bonne idée d'utiliser l'option /MD de MSVC, qui signifie que toutes les fonctions standards ne vont pas être liées avec le fichier exécutable, mais vont à la place être importées depuis le fichier MSVCR*.DLL. Ainsi il est plus facile de voir quelles fonctions standards sont utilisées et où.

En analysant du code dans IDA, il est très utile de laisser des notes pour soi-même (et les autres). En la circonstance, analysons cet exemple, nous voyons que JNZ sera déclenché en cas d'erreur. Donc il est possible de déplacer le curseur sur le label, de presser « n » et de lui donner le nom « error ». Créons un autre label—dans « exit ». Voici mon résultat:

```
.text :00401000 _main proc near
.text :00401000
.text :00401000 var_4 = dword ptr -4
.text :00401000 argc  = dword ptr  8
.text :00401000 argv  = dword ptr  0Ch
.text :00401000 envp  = dword ptr  10h
.text :00401000
.text :00401000       push    ebp
.text :00401001       mov     ebp, esp
.text :00401003       push    ecx
.text :00401004       push    offset Format ; "Enter X :\n"
.text :00401009       call    ds :printf
.text :0040100F       add     esp, 4
.text :00401012       lea     eax, [ebp+var_4]
.text :00401015       push    eax
.text :00401016       push    offset aD ; "%d"
.text :0040101B       call    ds :scanf
.text :00401021       add     esp, 8
.text :00401024       cmp     eax, 1
.text :00401027       jnz     short error
.text :00401029       mov     ecx, [ebp+var_4]
.text :0040102C       push    ecx
.text :0040102D       push    offset aYou ; "You entered %d...\n"
.text :00401032       call    ds :printf
.text :00401038       add     esp, 8
.text :0040103B       jmp     short exit
.text :0040103D
.text :0040103D error : ; CODE XREF : _main+27
.text :0040103D       push    offset aWhat ; "What you entered? Huh?\n"
.text :00401042       call    ds :printf
.text :00401048       add     esp, 4
.text :0040104B
.text :0040104B exit : ; CODE XREF : _main+3B
.text :0040104B       xor     eax, eax
.text :0040104D       mov     esp, ebp
.text :0040104F       pop     ebp
.text :00401050       retn
.text :00401050 _main endp
```

Maintenant, il est légèrement plus facile de comprendre le code. Toutefois, ce n'est pas une bonne idée de commenter chaque instruction.

Vous pouvez aussi cacher (replier) des parties d'une fonction dans IDA. Pour faire cela, marquez le bloc, puis appuyez sur « – » sur le pavé numérique et entrez le texte qui doit être affiché à la place.

Cachons deux blocs et donnons leurs un nom:

```
.text :00401000 _text segment para public 'CODE' use32
.text :00401000       assume cs :_text
.text :00401000       ;org 401000h
.text :00401000 ; ask for X
.text :00401012 ; get X
.text :00401024       cmp  eax, 1
.text :00401027       jnz  short error
.text :00401029 ; print result
.text :0040103B       jmp  short exit
.text :0040103D
.text :0040103D error : ; CODE XREF : _main+27
```

```
.text :0040103D       push offset aWhat ; "What you entered? Huh?\n"
.text :00401042       call ds :printf
.text :00401048       add  esp, 4
.text :0040104B
.text :0040104B exit : ; CODE XREF : _main+3B
.text :0040104B       xor  eax, eax
.text :0040104D       mov  esp, ebp
.text :0040104F       pop  ebp
.text :00401050       retn
.text :00401050 _main endp
```

Pour étendre les parties de code précédemment cachées. utilisez « + » sur le pavé numérique.

En appuyant sur « space », nous voyons comment IDA représente une fonction sous forme de graphe:

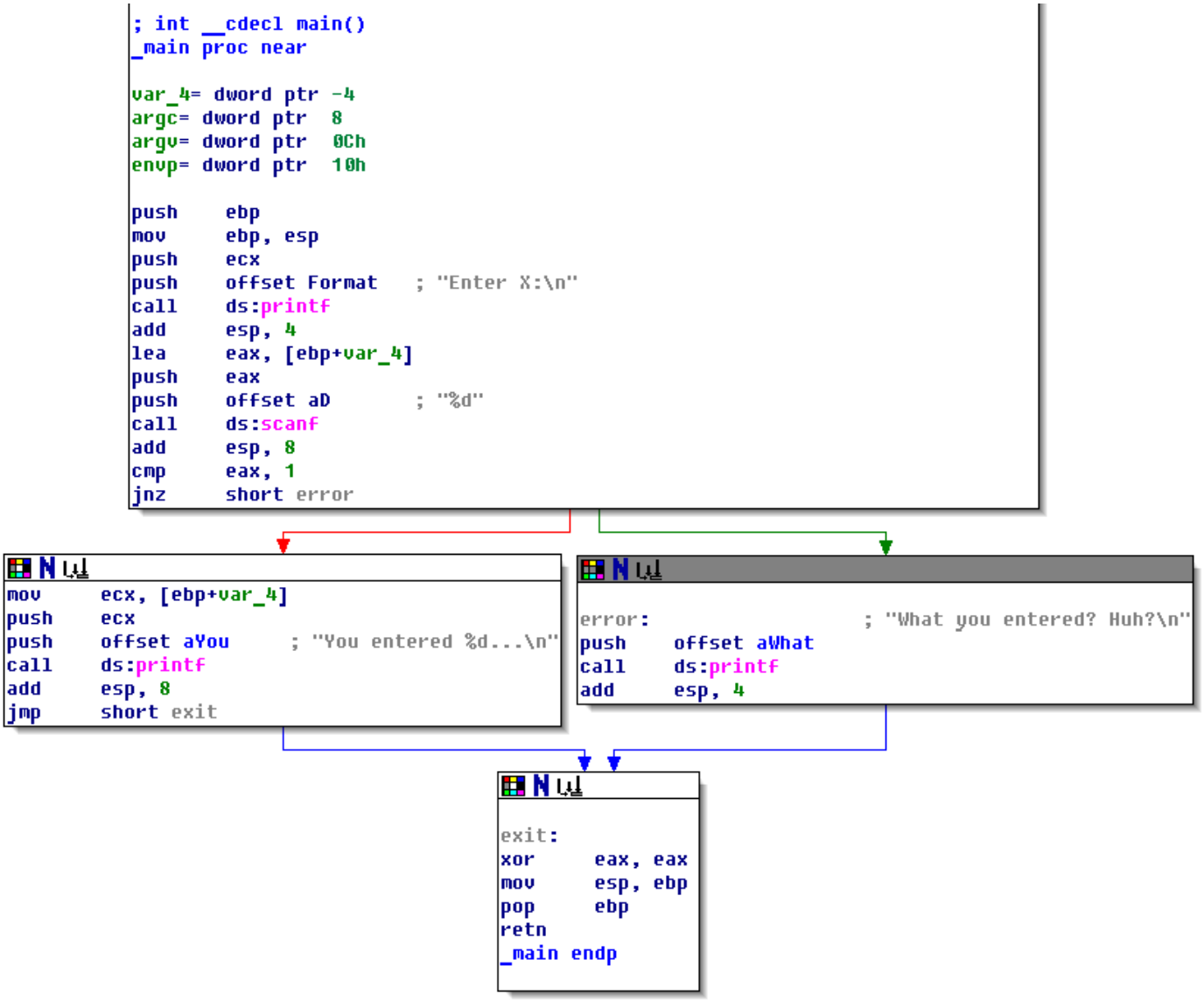


Fig. 1.17: IDA en mode graphe

Il y a deux flèches après chaque saut conditionnel: une verte et une rouge. La flèche verte pointe vers le bloc qui sera exécuté si le saut est déclenché, et la rouge sinon.

Il est possible de replier des nœuds dans ce mode et de leurs donner aussi un nom (« group nodes »). Essayons avec 3 blocs:

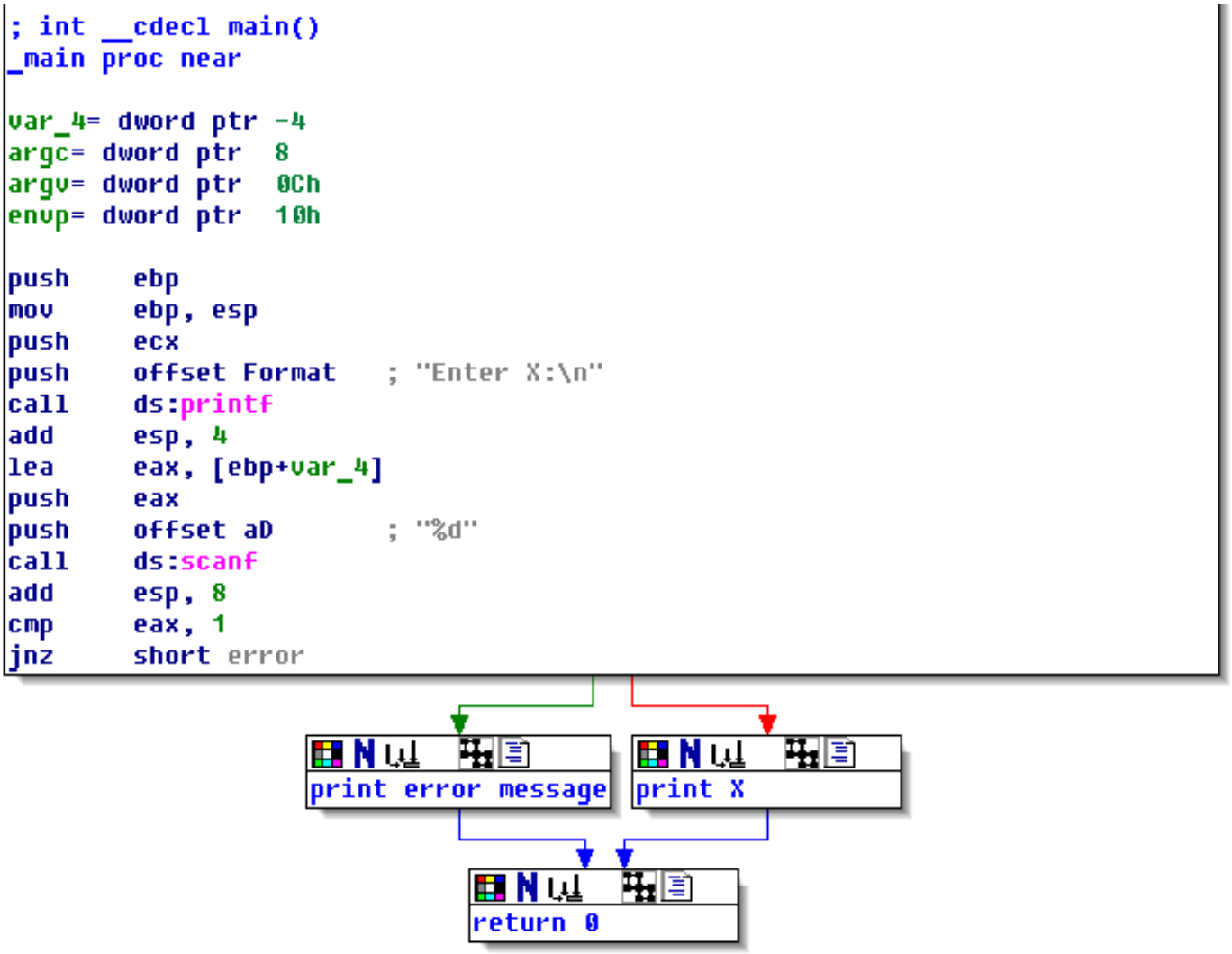


Fig. 1.18: IDA en mode graphe avec 3 nœuds repliés

C'est très pratique. On peut dire qu'une part importante du travail des rétro-ingénieurs (et de tout autre chercheur également) est de réduire la quantité d'information avec laquelle travailler.

MSVC: x86 + OllyDbg

Essayons de hacker notre programme dans OllyDbg, pour le forcer à penser que `scanf()` fonctionne toujours sans erreur. Lorsque l'adresse d'une variable locale est passée à `scanf()`, la variable contient initialement toujours des restes de données aléatoires, dans ce cas `0x6E494714` :

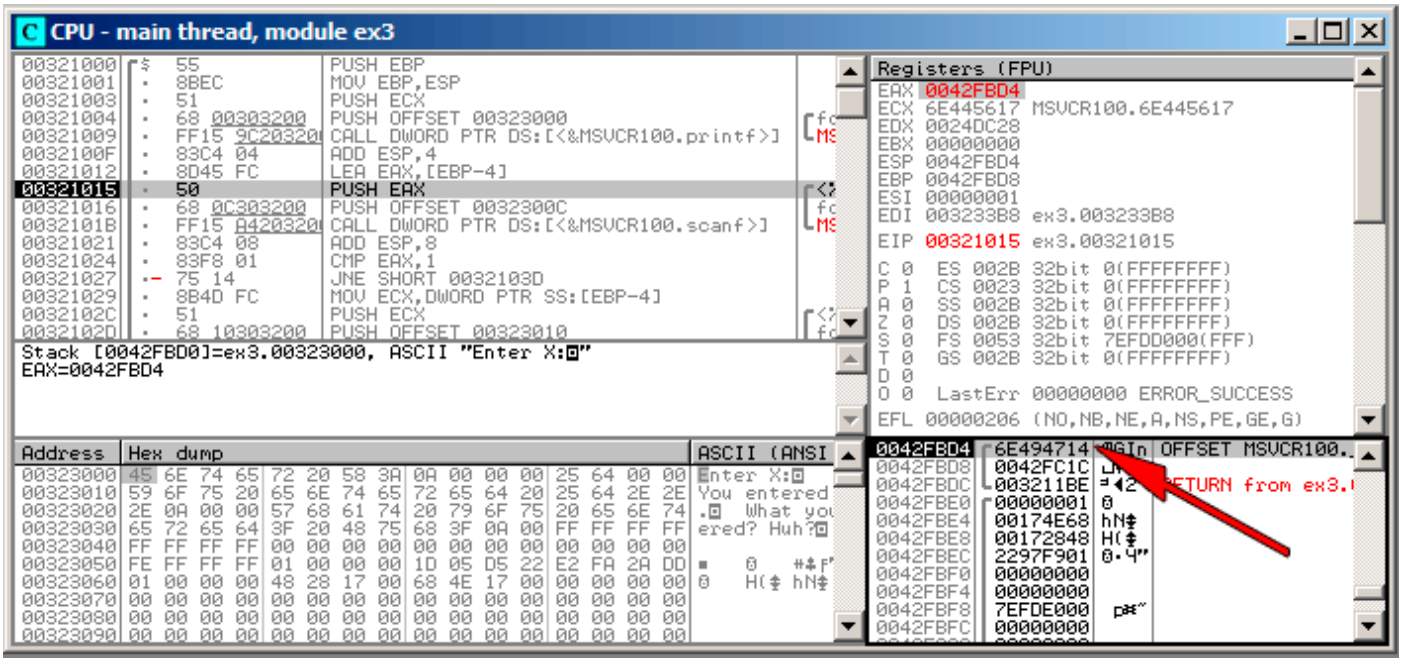


Fig. 1.19: OllyDbg : passer l'adresse de la variable à `scanf()`

Lorsque `scanf()` s'exécute dans la console, entrons quelque chose qui n'est pas du tout un nombre, comme « asdasd ». `scanf()` termine avec 0 dans EAX, ce qui indique qu'une erreur s'est produite:

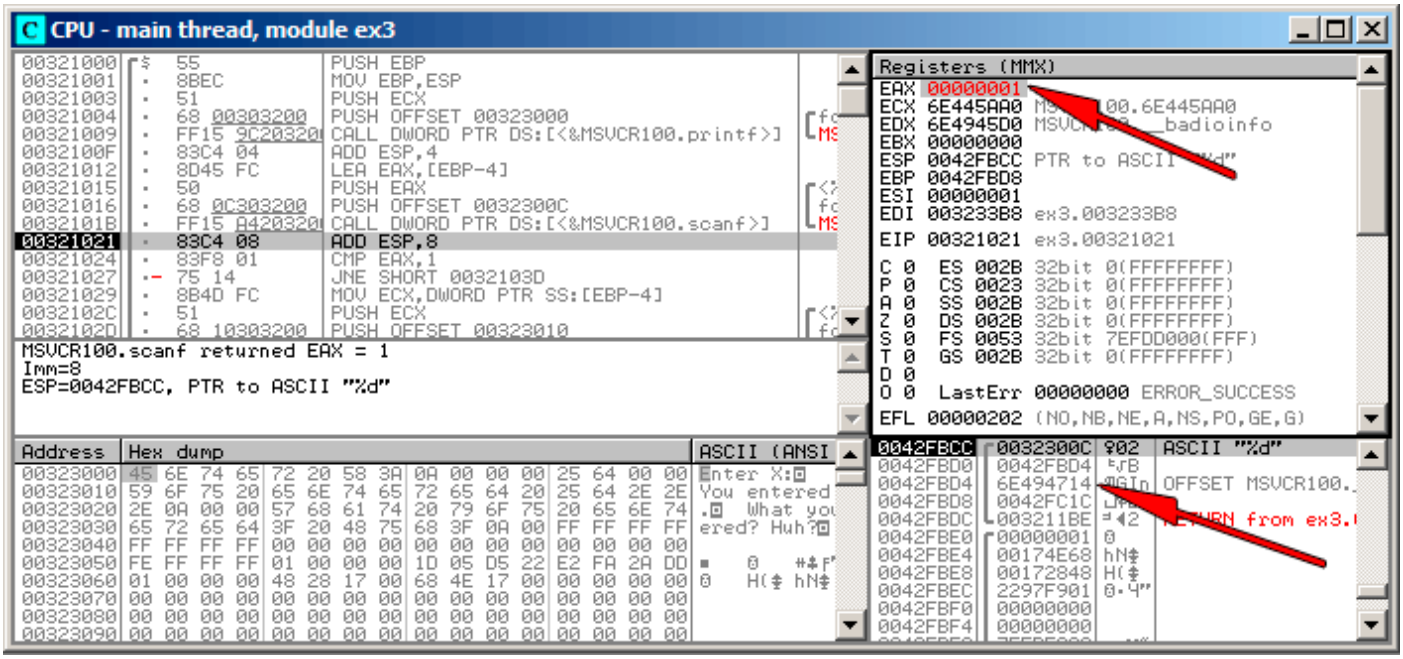


Fig. 1.20: OllyDbg : `scanf()` renvoyant une erreur

Nous pouvons vérifier la variable locale dans le pile et noter qu'elle n'a pas changé. En effet, qu'aurait écrit `scanf()` ici? Elle n'a simplement rien fait à part renvoyer zéro.

Essayons de « hacker » notre programme. Clique-droit sur EAX, parmi les options il y a « Set to 1 » (mettre à 1). C'est ce dont nous avons besoin.

Nous avons maintenant 1 dans EAX, donc la vérification suivante va s'exécuter comme souhaiter et `printf()` va afficher la valeur de la variable dans la pile.

Lorsque nous lançons le programme (F9) nous pouvons voir ceci dans la fenêtre de la console:

Listing 1.80: fenêtre console

```
Enter X :
asdasd
You entered 1850296084...
```

En effet, 1850296084 est la représentation en décimal du nombre dans la pile (0x6E494714)!

MSVC: x86 + Hiew

Cela peut également être utilisé comme un exemple simple de modification de fichier exécutable. Nous pouvons essayer de modifier l'exécutable de telle sorte que le programme va toujours afficher notre entrée, quelle quelle soit.

En supposant que l'exécutable est compilé avec la bibliothèque externe `MSVCR*.DLL` (i.e., avec l'option `/MD`) [79], nous voyons la fonction `main()` au début de la section `.text`. Ouvrons l'exécutable dans Hiew et cherchons le début de la section `.text` (Enter, F8, F6, Enter, Enter).

Nous pouvons voir cela:

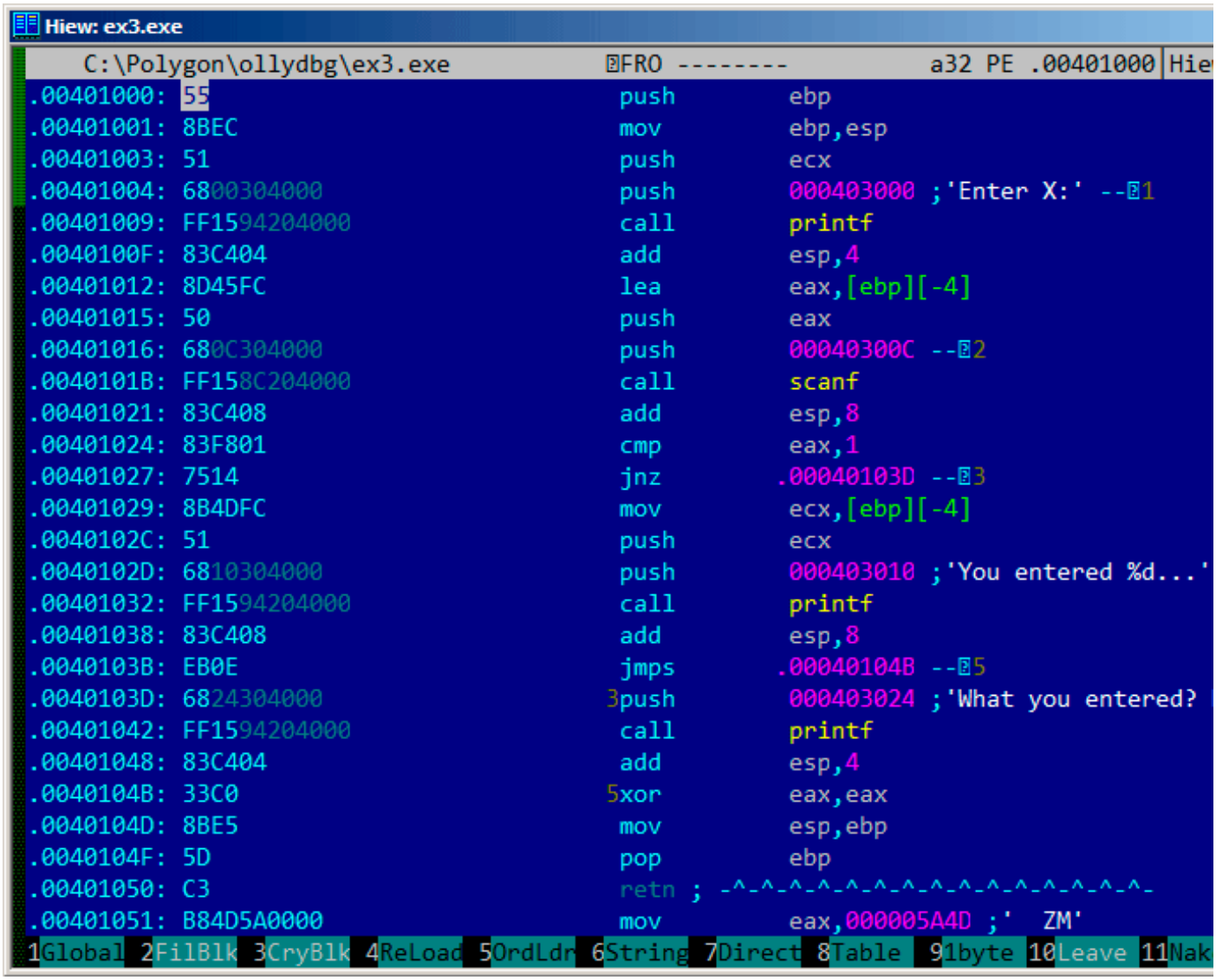


Fig. 1.21: Hiew: fonction `main()`

Hiew trouve les chaîne ASCIIZ[80] et les affiche, comme il le fait avec le nom des fonctions importées.

[79] c'est aussi appelé « dynamic linking »
[80] ASCII Zero (chaîne ASCII terminée par un octet nul (à zéro))

Déplacez le curseur à l'adresse .00401027 (où se trouve l'instruction JNZ, que l'on doit sauter), appuyez sur F3, et ensuite tapez « 9090 » (qui signifie deux NOPs):

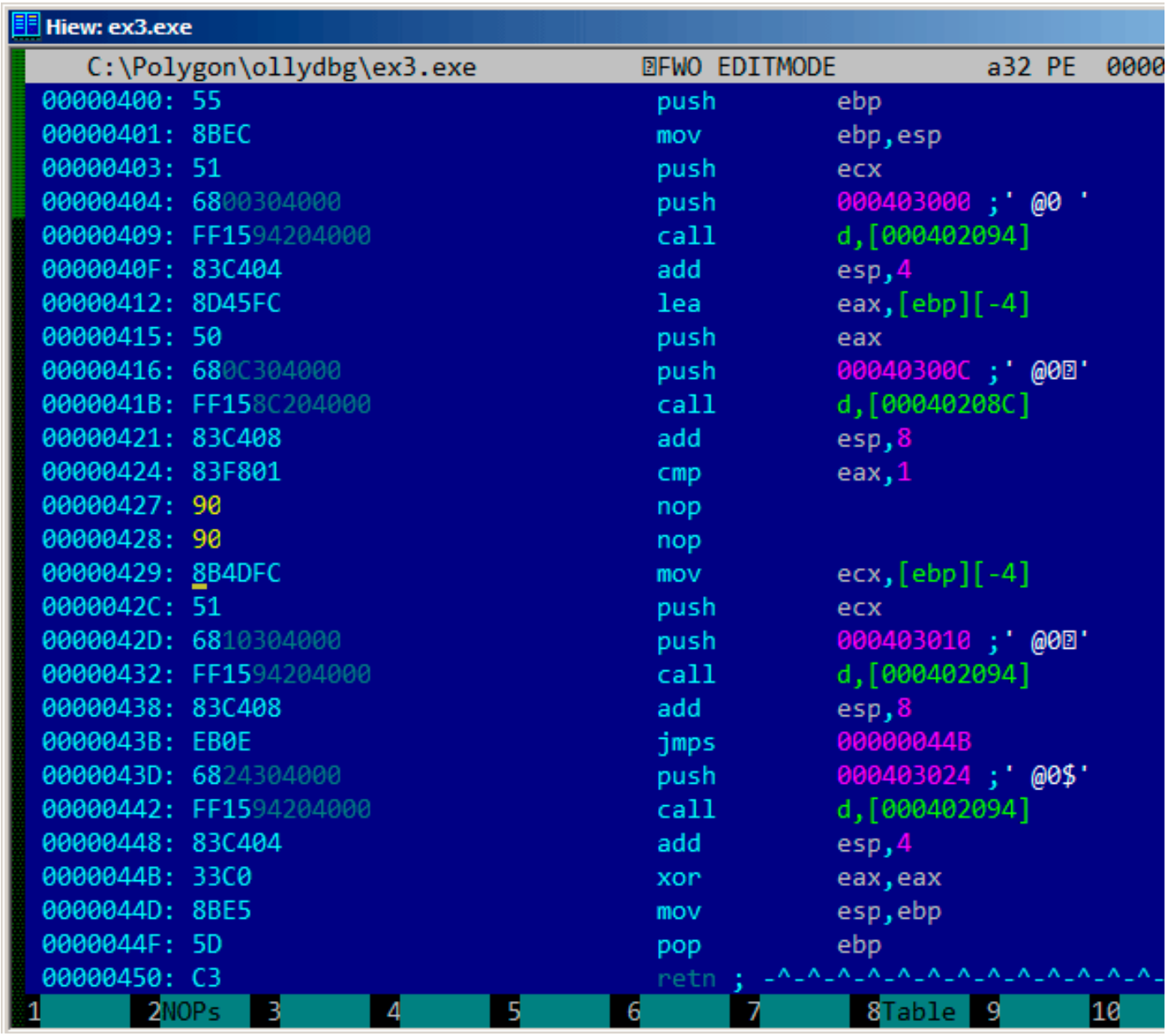


Fig. 1.22: Hiew: remplacement de JNZ par deux NOPs

Appuyez sur F9 (update). Maintenant, l'exécutable est sauvé sur le disque. Il va se comporter comme nous le voulions.

Deux NOPs ne constitue probablement pas l'approche la plus esthétique. Une autre façon de modifier cette instruction est d'écrire simplement 0 dans le second octet de l'opcode ((jump offset), donc ce JNZ va toujours sauter à l'instruction suivante.

Nous pouvons également faire le contraire: remplacer le premier octet avec EB sans modifier le second octet (jump offset). Nous obtiendrions un saut inconditionnel qui est toujours déclenché. Dans ce cas le message d'erreur sera affiché à chaque fois, peu importe l'entrée.

MSVC: x64

ARM

ARM: avec optimisation Keil 6/2013 (Mode Thumb)

Listing 1.81: avec optimisation Keil 6/2013 (Mode Thumb)

```
var_8    = -8

         PUSH    {R3,LR}
         ADR     R0, aEnterX     ; "Enter X :\n"
         BL      __2printf
         MOV     R1, SP
         ADR     R0, aD          ; "%d"
         BL      __0scanf
         CMP     R0, #1
         BEQ     loc_1E
         ADR     R0, aWhatYouEntered ; "What you entered? Huh?\n"
         BL      __2printf

loc_1A                           ; CODE XREF : main+26
         MOVS    R0, #0
         POP     {R3,PC}

loc_1E                           ; CODE XREF : main+12
         LDR     R1, [SP,#8+var_8]
         ADR     R0, aYouEnteredD___ ; "You entered %d...\n"
         BL      __2printf
         B       loc_1A
```

Les nouvelles instructions sont CMP et BEQ[81].

CMP est similaire à l'instruction x86 du même nom, elle soustrait l'un des arguments à l'autre et met à jour les flags si nécessaire.

BEQ saute à une autre adresse si les opérandes étaient égaux l'un à l'autre, ou, si le résultat du dernier calcul était 0, ou si le flag Z est à 1. Elle se comporte comme JZ en x86.

Tout le reste est simple: le flux d'exécution se sépare en deux branches, puis les branches convergent vers le point où 0 est écrit dans le registre R0 comme valeur de retour de la fonction, et cette dernière se termine.

ARM64

Listing 1.82: GCC 4.9.1 ARM64 sans optimisation

```
1  .LC0 :
2          .string "Enter X :"
3  .LC1 :
4          .string "%d"
5  .LC2 :
6          .string "You entered %d...\n"
7  .LC3 :
8          .string "What you entered? Huh?"
9  f6 :
10 ; sauver FP et LR dans la structure de pile locale :
11         stp     x29, x30, [sp, -32]!
12 ; définir la pile locale (FP=SP)
13         add     x29, sp, 0
14 ; charger le pointeur sur la chaîne "Enter X :":
15         adrp    x0, .LC0
16         add     x0, x0,  :lo12 :.LC0
17         bl      puts
18 ; charger le pointeur sur la chaîne "%d":
19         adrp    x0, .LC1
20         add     x0, x0,  :lo12 :.LC1
21 ; calculer l'adresse de la variable x dans la pile locale
22         add     x1, x29, 28
23         bl      __isoc99_scanf
24 ; scanf() renvoie son résultat dans W0.
25 ; le vérifier:
26         cmp     w0, 1
27 ; BNE est Branch if Not Equal (branchement si non égal)
```

[81] (PowerPC, ARM) Branch if Equal

```
; donc if W0<>0, un saut en L2 sera effectué
        bne     .L2
; à ce point W0=1, signifie pas d'erreur
; charger la valeur de x depuis la pile locale
        ldr     w1, [x29,28]
; charger le pointeur sur la chaîne "You entered %d...\n":
        adrp    x0, .LC2
        add     x0, x0,  :lo12 :.LC2
        bl      printf
; sauter le code, qui affiche la chaîne "What you entered? Huh?":
        b       .L3
.L2 :
; charger le pointeur sur la chaîne "What you entered? Huh?":
        adrp    x0, .LC3
        add     x0, x0,  :lo12 :.LC3
        bl      puts
.L3 :
; retourner 0
        mov     w0, 0
; restaurer FP et LR :
        ldp     x29, x30, [sp], 32
        ret
```

Dans ce cas, le flux de code se sépare avec l'utilisation de la paire d'instructions CMP/BNE (Branch if Not Equal) (branchement si non égal).

MIPS

Listing 1.83: avec optimisation GCC 4.4.5 (IDA)

```
.text :004006A0 main :
.text :004006A0
.text :004006A0 var_18    = -0x18
.text :004006A0 var_10    = -0x10
.text :004006A0 var_4     = -4
.text :004006A0
.text :004006A0           lui     $gp, 0x42
.text :004006A4           addiu   $sp, -0x28
.text :004006A8           li      $gp, 0x418960
.text :004006AC           sw      $ra, 0x28+var_4($sp)
.text :004006B0           sw      $gp, 0x28+var_18($sp)
.text :004006B4           la      $t9, puts
.text :004006B8           lui     $a0, 0x40
.text :004006BC           jalr    $t9 ; puts
.text :004006C0           la      $a0, aEnterX     # "Enter X :"
.text :004006C4           lw      $gp, 0x28+var_18($sp)
.text :004006C8           lui     $a0, 0x40
.text :004006CC           la      $t9, __isoc99_scanf
.text :004006D0           la      $a0, aD          # "%d"
.text :004006D4           jalr    $t9 ; __isoc99_scanf
.text :004006D8           addiu   $a1, $sp, 0x28+var_10  # branch delay slot
.text :004006DC           li      $v1, 1
.text :004006E0           lw      $gp, 0x28+var_18($sp)
.text :004006E4           beq     $v0, $v1, loc_40070C
.text :004006E8           or      $at, $zero       # branch delay slot, NOP
.text :004006EC           la      $t9, puts
.text :004006F0           lui     $a0, 0x40
.text :004006F4           jalr    $t9 ; puts
.text :004006F8           la      $a0, aWhatYouEntered  # "What you entered? Huh?"
.text :004006FC           lw      $ra, 0x28+var_4($sp)
.text :00400700           move    $v0, $zero
.text :00400704           jr      $ra
.text :00400708           addiu   $sp, 0x28

.text :0040070C loc_40070C :
.text :0040070C           la      $t9, printf
.text :00400710           lw      $a1, 0x28+var_10($sp)
.text :00400714           lui     $a0, 0x40
```

```
.text :00400718           jalr    $t9 ; printf
.text :0040071C           la      $a0, aYouEnteredD___  # "You entered %d...\n"
.text :00400720           lw      $ra, 0x28+var_4($sp)
.text :00400724           move    $v0, $zero
.text :00400728           jr      $ra
.text :0040072C           addiu   $sp, 0x28
```

Exercice

Comme nous pouvons voir, les instructions JNE/JNZ peuvent facilement être remplacées par JE/JZ et vice-versa (ou BNE par BEQ et vice-versa). Mais les blocs de base doivent aussi être échangés. Essayez de faire cela pour quelques exemples.

1.9.5 Exercice

- http://challenges.re/53

1.10 Accéder aux arguments passés

Maintenant nous savons que la fonction appelante passe les arguments à la fonction appelée par la pile. Mais comment est-ce que la fonction appelée y accède?

Listing 1.84: exemple simple

```
#include <stdio.h>

int f (int a, int b, int c)
{
        return a*b+c;
};

int main()
{
        printf ("%d\n", f(1, 2, 3));
        return 0;
};
```

1.10.1 x86

MSVC

Voici ce que l'on obtient après compilation (MSVC 2010 Express) :

Listing 1.85: MSVC 2010 Express

```
_TEXT   SEGMENT
_a$ = 8         ; taille = 4
_b$ = 12        ; taille = 4
_c$ = 16        ; taille = 4
_f      PROC
        push    ebp
        mov     ebp, esp
        mov     eax, DWORD PTR _a$[ebp]
        imul    eax, DWORD PTR _b$[ebp]
        add     eax, DWORD PTR _c$[ebp]
        pop     ebp
        ret     0
_f      ENDP

_main   PROC
        push    ebp
```

```
        mov     ebp, esp
        push    3 ; 3ème argument
        push    2 ; 2ème argument
        push    1 ; 1er argument
        call    _f
        add     esp, 12
        push    eax
        push    OFFSET $SG2463 ; '%d', 0aH, 00H
        call    _printf
        add     esp, 8
        ; retourner 0
        xor     eax, eax
        pop     ebp
        ret     0
_main   ENDP
```

Ce que l'on voit, c'est que la fonction `main()` pousse 3 nombres sur la pile et appelle `f(int,int,int)`.

L'accès aux arguments à l'intérieur de `f()` est organisé à l'aide de macros comme:
`_a$ = 8`, de la même façon que pour les variables locales, mais avec des offsets positifs (accédés avec *plus*). Donc, nous accédons à la partie *hors* de la structure locale de pile en ajoutant la macro `_a$` à la valeur du registre EBP.

Ensuite, la valeur de a est stockée dans EAX. Après l'exécution de l'instruction `IMUL`, la valeur de EAX est le produit de la valeur de EAX et du contenu de `_b`.

Après cela, `ADD` ajoute la valeur dans `_c` à EAX.

La valeur dans EAX n'a pas besoin d'être déplacée/copiée : elle est déjà là où elle doit être. Lors du retour dans la fonction appelante, elle prend la valeur dans EAX et l'utilise comme argument pour `printf()`.

MSVC + OllyDbg

Illustrons ceci dans OllyDbg. Lorsque nous traçons jusqu'à la première instruction de `f()` qui utilise un des arguments (le premier), nous voyons qu'EBP pointe sur la structure de pile locale, qui est entourée par un rectangle rouge.

Le premier élément de la structure de pile locale est la valeur sauvegardée de EBP, le second est RA, le troisième est le premier argument de la fonction, puis le second et le troisième.

Pour accéder au premier argument de la fonction, on doit ajouter exactement 8 (2 mots de 32-bit) à EBP.

OllyDbg est au courant de cela, c'est pourquoi il a ajouté des commentaires aux éléments de la pile comme

« RETURN from » et « Arg1 = ... », etc.

N.B.: Les arguments de la fonction ne font pas partie de la structure de pile de la fonction, ils font plutôt partie de celle de la fonction appelante.

Par conséquent, OllyDbg a marqué les éléments comme appartenant à une autre structure de pile.

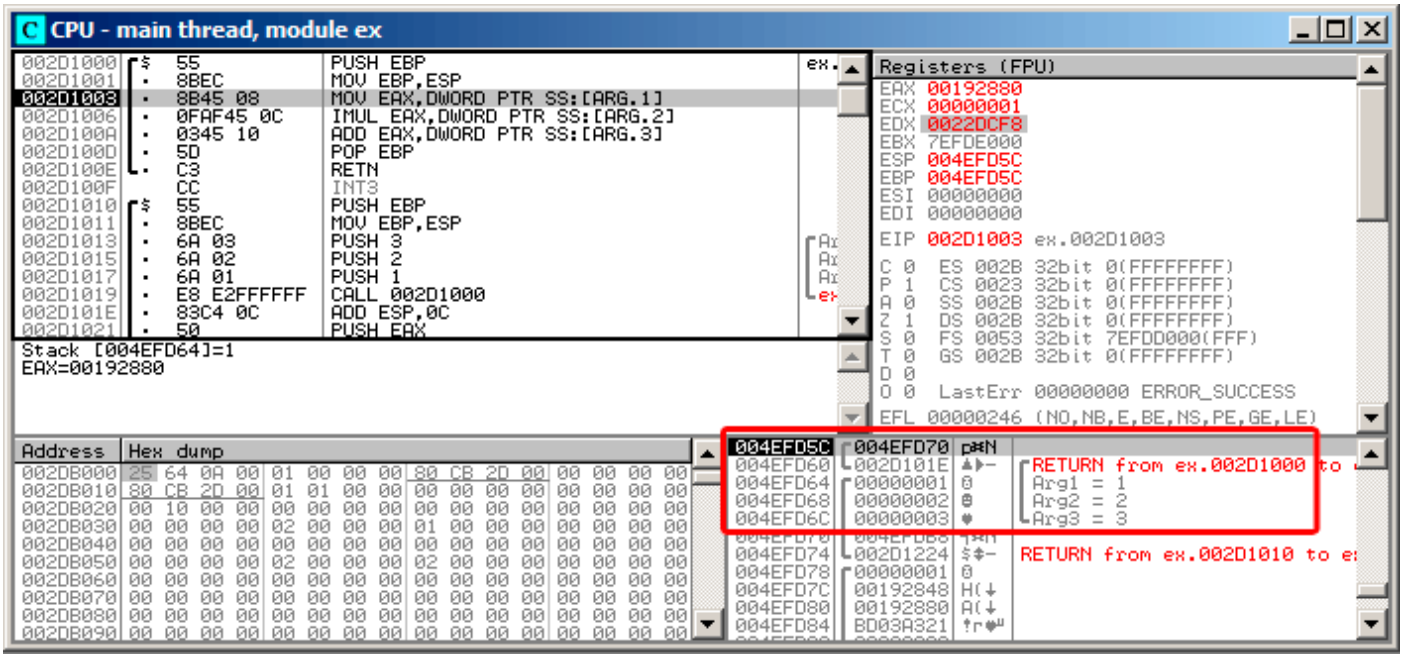


Fig. 1.23: OllyDbg : à l'intérieur de la fonction f()

GCC

Compilons le même code avec GCC 4.4.1 et regardons le résultat dans IDA :

Listing 1.86: GCC 4.4.1

```
        public f
f       proc near

arg_0   = dword ptr  8
arg_4   = dword ptr  0Ch
arg_8   = dword ptr  10h

        push    ebp
        mov     ebp, esp
        mov     eax, [ebp+arg_0] ; 1er argument
        imul    eax, [ebp+arg_4] ; 2ème argument
        add     eax, [ebp+arg_8] ; 3ème argument
        pop     ebp
        retn
f       endp

        public main
main    proc near

var_10  = dword ptr -10h
var_C   = dword ptr -0Ch
var_8   = dword ptr -8

        push    ebp
        mov     ebp, esp
        and     esp, 0FFFFFFF0h
        sub     esp, 10h
        mov     [esp+10h+var_8], 3 ; 3ème argument
        mov     [esp+10h+var_C], 2 ; 2ème argument
        mov     [esp+10h+var_10], 1 ; 1er argument
        call    f
        mov     edx, offset aD  ; "%d\n"
        mov     [esp+10h+var_C], eax
        mov     [esp+10h+var_10], edx
        call    _printf
        mov     eax, 0
        leave
```

```
        retn
main    endp
```

Le résultat est presque le même, avec quelques différences mineures discutées précédemment.

Le pointeur de pile n'est pas remis après les deux appels de fonction (f et printf), car la pénultième instruction LEAVE (**??** on page ??) s'en occupe à la fin.

1.10.2 x64

Le scénario est un peu différent en x86-64. Les arguments de la fonction (les 4 ou 6 premiers d'entre eux) sont passés dans des registres i.e. l'appelée les lit depuis des registres au lieu de les lire dans la pile.

MSVC

MSVC avec optimisation :

Listing 1.87: MSVC 2012 x64 avec optimisation

```
$SG2997 DB      '%d', 0aH, 00H

main    PROC
        sub     rsp, 40
        mov     edx, 2
        lea     r8d, QWORD PTR [rdx+1] ; R8D=3
        lea     ecx, QWORD PTR [rdx-1] ; ECX=1
        call    f
        lea     rcx, OFFSET FLAT :$SG2997 ; '%d'
        mov     edx, eax
        call    printf
        xor     eax, eax
        add     rsp, 40
        ret     0
main    ENDP

f       PROC
        ; ECX - 1er argument
        ; EDX - 2ème argument
        ; R8D - 3ème argument
        imul    ecx, edx
        lea     eax, DWORD PTR [r8+rcx]
        ret     0
f       ENDP
```

Comme on peut le voir, la fonction compacte `f()` prend tous ses arguments dans des registres.

La fonction LEA est utilisée ici pour l'addition, apparemment le compilateur considère qu'elle plus rapide que ADD.

LEA est aussi utilisée dans la fonction `main()` pour préparer le premier et le troisième argument de `f()`. Le compilateur doit avoir décidé que cela s'exécutera plus vite que la façon usuelle ce charger des valeurs dans les registres, qui utilise l'instruction MOV.

Regardons ce qu'a généré MSVC sans optimisation:

Listing 1.88: MSVC 2012 x64

```
f               proc near

; shadow space :
arg_0           = dword ptr  8
arg_8           = dword ptr  10h
arg_10          = dword ptr  18h

                ; ECX - 1er argument
                ; EDX - 2ème argument
                ; R8D - 3ème argument
                mov     [rsp+arg_10], r8d
```

```
                mov     [rsp+arg_8], edx
                mov     [rsp+arg_0], ecx
                mov     eax, [rsp+arg_0]
                imul    eax, [rsp+arg_8]
                add     eax, [rsp+arg_10]
                retn
f               endp

main            proc near
                sub     rsp, 28h
                mov     r8d, 3 ; 3rd argument
                mov     edx, 2 ; 2nd argument
                mov     ecx, 1 ; 1st argument
                call    f
                mov     edx, eax
                lea     rcx, $SG2931    ; "%d\n"
                call    printf

                ; retourner 0
                xor     eax, eax
                add     rsp, 28h
                retn
main            endp
```

C'est un peu déroutant, car les 3 arguments dans des registres sont sauvegardés sur la pile pour une certaine raison. Ceci est appelé « shadow space » [82] : chaque Win64 peut (mais ce n'est pas requis) y sauver les 4 registres. Ceci est fait pour deux raisons: 1) c'est trop généreux d'allouer un registre complet (et même 4 registres) pour un argument en entrée, donc il sera accédé par la pile; 2) le debugger sait toujours où trouver les arguments de la fonction lors d'un arrêt [83].

Donc, de grosses fonctions peuvent sauvegarder leurs arguments en entrée dans le « shadows space » si elle veulent les utiliser pendant l'exécution, mais quelques petites fonctions (comme la notre) peuvent ne pas le faire.

C'est la responsabilité de l'appelant d'allouer le « shadow space » sur la pile.

GCC

GCC avec optimisation génère du code plus ou moins compréhensible:

Listing 1.89: GCC 4.4.6 x64 avec optimisation

```
f :
        ; EDI - 1er argument
        ; ESI - 2ème argument
        ; EDX - 3ème argument
        imul    esi, edi
        lea     eax, [rdx+rsi]
        ret

main :
        sub     rsp, 8
        mov     edx, 3
        mov     esi, 2
        mov     edi, 1
        call    f
        mov     edi, OFFSET FLAT :.LC0 ; "%d\n"
        mov     esi, eax
        xor     eax, eax  ; nombre de registres vectoriel passés
        call    printf
        xor     eax, eax
        add     rsp, 8
        ret
```

GCC sans optimisation :

[82] MSDN
[83] MSDN

Listing 1.90: GCC 4.4.6 x64

```
f :
        ; EDI - 1er argument
        ; ESI - 2ème argument
        ; EDX - 3ème argument
        push    rbp
        mov     rbp, rsp
        mov     DWORD PTR [rbp-4], edi
        mov     DWORD PTR [rbp-8], esi
        mov     DWORD PTR [rbp-12], edx
        mov     eax, DWORD PTR [rbp-4]
        imul    eax, DWORD PTR [rbp-8]
        add     eax, DWORD PTR [rbp-12]
        leave
        ret
main :
        push    rbp
        mov     rbp, rsp
        mov     edx, 3
        mov     esi, 2
        mov     edi, 1
        call    f
        mov     edx, eax
        mov     eax, OFFSET FLAT :.LC0 ; "%d\n"
        mov     esi, edx
        mov     rdi, rax
        mov     eax, 0  ; nombre de registres vectoriel passés
        call    printf
        mov     eax, 0
        leave
        ret
```

Il n'y a pas d'exigeance de « shadow space » en System V *NIX ([Michael Matz, Jan Hubicka, Andreas Jaeger, Mark Mitchell, *System V Application Binary Interface. AMD64 Architecture Processor Supplement*, (2013)] [84]), mais l'appelée peut vouloir sauvegarder ses arguments quelque part en cas de manque de registres.

GCC: uint64_t au lieu de int

Notre exemple fonctionne avec des *int* 32-bit, c'est pourquoi c'est la partie 32-bit des registres qui est utilisée (préfixée par E-).

Il peut être légèrement modifié pour utiliser des valeurs 64-bit:

```
#include <stdio.h>
#include <stdint.h>

uint64_t f (uint64_t a, uint64_t b, uint64_t c)
{
        return a*b+c;
};

int main()
{
        printf ("%lld\n", f(0x1122334455667788,
                            0x1111111122222222,
                            0x3333333344444444));
        return 0;
};
```

Listing 1.91: GCC 4.4.6 x64 avec optimisation

```
f       proc near
        imul    rsi, rdi
        lea     rax, [rdx+rsi]
        retn
```

[84] Aussi disponible en https://software.intel.com/sites/default/files/article/402129/mpx-linux64-abi.pdf

```
f       endp

main    proc near
        sub     rsp, 8
        mov     rdx, 3333333344444444h ; 3ème argument
        mov     rsi, 1111111122222222h ; 2ème argument
        mov     rdi, 1122334455667788h ; 1er argument
        call    f
        mov     edi, offset format ; "%lld\n"
        mov     rsi, rax
        xor     eax, eax ; nombre de registres vectoriel passés
        call    _printf
        xor     eax, eax
        add     rsp, 8
        retn
main    endp
```

Le code est le même, mais cette fois les registres *complets* (préfixés par R-) sont utilisés.

1.10.3 ARM

sans optimisation Keil 6/2013 (Mode ARM)

```
.text :000000A4 00 30 A0 E1      MOV     R3, R0
.text :000000A8 93 21 20 E0      MLA     R0, R3, R1, R2
.text :000000AC 1E FF 2F E1      BX      LR
...
.text :000000B0              main
.text :000000B0 10 40 2D E9      STMFD   SP!, {R4,LR}
.text :000000B4 03 20 A0 E3      MOV     R2, #3
.text :000000B8 02 10 A0 E3      MOV     R1, #2
.text :000000BC 01 00 A0 E3      MOV     R0, #1
.text :000000C0 F7 FF FF EB      BL      f
.text :000000C4 00 40 A0 E1      MOV     R4, R0
.text :000000C8 04 10 A0 E1      MOV     R1, R4
.text :000000CC 5A 0F 8F E2      ADR     R0, aD_0        ; "%d\n"
.text :000000D0 E3 18 00 EB      BL      __2printf
.text :000000D4 00 00 A0 E3      MOV     R0, #0
.text :000000D8 10 80 BD E8      LDMFD   SP!, {R4,PC}
```

La fonction `main()` appelle simplement deux autres fonctions, avec trois valeurs passées à la première — (`f()`).

Comme il y déjà été écrit, en ARM les 4 premières valeurs sont en général passées par les 4 premiers registres (R0-R3).

La fonction `f()`, comme il semble, utilise les 3 premiers registres (R0-R2) comme arguments.

L'instruction `MLA` (*Multiply Accumulate*) multiplie ses deux premiers opérandes (R3 et R1), additionne le troisième opérande (R2) au produit et stocke le résultat dans le registre zéro (R0), par lequel, d'après le standard, les fonctions retournent leur résultat.

La multiplication et l'addition en une fois (*Fused multiply–add*) est une instruction très utile. À propos, il n'y avait pas une telle instruction en x86 avant les instructions FMA apparues en SIMD [85].

La toute première instruction `MOV R3, R0`, est, apparemment, redondante (car une seule instruction `MLA` pourrait être utilisée à la place ici). Le compilateur ne l'a pas optimisé, puisqu'il n'y a pas l'option d'optimisation.

L'instruction BX rend le contrôle à l'adresse stockée dans le registre LR et, si nécessaire, change le mode du processeur de Thumb à ARM et vice versa. Ceci peut être nécessaire puisque, comme on peut le voir, la fonction `f()` n'est pas au courant depuis quel sorte de code elle peut être appelée, ARM ou Thumb. Ainsi, si elle est appelée depuis du code Thumb, BX ne va pas seulement retourner le contrôle à la fonction appelante, mais également changer le mode du processeur à Thumb. Ou ne pas changer si la fonction a été appelée depuis du code ARM [*ARM(R) Architecture Reference Manual, ARMv7-A and ARMv7-R edition*, (2012)A2.3.2].

[85] Wikipédia

avec optimisation Keil 6/2013 (Mode ARM)

```
.text :00000098             f
.text :00000098 91 20 20 E0       MLA     R0, R1, R0, R2
.text :0000009C 1E FF 2F E1       BX      LR
```

Et voilà la fonction `f()` compilée par le compilateur Keil en mode optimisation maximale (`-03`).

L'instruction `MOV` a été supprimée par l'optimisation (ou réduite) et maintenant `MLA` utilise tout les registres contenant les données en entrée et place ensuite le résultat directement dans `R0`, exactement où la fonction appelante va le lire et l'utiliser.

avec optimisation Keil 6/2013 (Mode Thumb)

```
.text :0000005E 48 43          MULS    R0, R1
.text :00000060 80 18          ADDS    R0, R0, R2
.text :00000062 70 47          BX      LR
```

L'instruction `MLA` n'est pas disponible dans le mode Thumb, donc le compilateur génère le code effectuant ces deux opérations (multiplication et addition) séparément.

Tout d'abord, la première instruction `MULS` multiplie `R0` par `R1`, laissant le résultat dans le registre`R0`. La seconde instruction (`ADDS`) ajoute le résultat et `R2` laissant le résultat dans le registre `R0`.

ARM64

GCC (Linaro) 4.9 avec optimisation

Tout ce qu'il y a ici est simple. `MADD` est juste une instruction qui effectue une multiplication/addition fusionnées (similaire à l'instruction `MLA` que nous avons déjà vue). Tous les 3 arguments sont passés dans la partie 32-bit de X-registres. Effectivement, le type des arguments est *int* 32-bit. Le résultat est renvoyé dans `W0`.

Listing 1.92: GCC (Linaro) 4.9 avec optimisation

```
f :
        madd    w0, w0, w1, w2
        ret

main :
; sauver FP et LR dans la pile locale :
        stp     x29, x30, [sp, -16]!
        mov     w2, 3
        mov     w1, 2
        add     x29, sp, 0
        mov     w0, 1
        bl      f
        mov     w1, w0
        adrp    x0, .LC7
        add     x0, x0,  :lo12 :.LC7
        bl      printf
; retourner 0
        mov     w0, 0
; restaurer FP et LR
        ldp     x29, x30, [sp], 16
        ret

.LC7 :
        .string "%d\n"
```

Étendons le type de toutes les données à 64-bit `uint64_t` et testons:

```
#include <stdio.h>
#include <stdint.h>
```

```
uint64_t f (uint64_t a, uint64_t b, uint64_t c)
{
        return a*b+c;
};

int main()
{
        printf ("%lld\n", f(0x1122334455667788,
                            0x1111111122222222,
                            0x3333333344444444));
        return 0;
};
```

```
f :
        madd    x0, x0, x1, x2
        ret
main :
        mov     x1, 13396
        adrp    x0, .LC8
        stp     x29, x30, [sp, -16]!
        movk    x1, 0x27d0, lsl 16
        add     x0, x0,  :lo12 :.LC8
        movk    x1, 0x122, lsl 32
        add     x29, sp, 0
        movk    x1, 0x58be, lsl 48
        bl      printf
        mov     w0, 0
        ldp     x29, x30, [sp], 16
        ret

.LC8 :
        .string "%lld\n"
```

La fonction `f()` est la même, seulement les X-registres 64-bit sont utilisés entièrement maintenant. Les valeurs longues sur 64-bit sont chargées dans les registres par partie, c'est également décrit ici: 1.32.3 on page 443.

GCC (Linaro) 4.9 sans optimisation

Le code sans optimisation est plus redondant:

```
f :
        sub     sp, sp, #16
        str     w0, [sp,12]
        str     w1, [sp,8]
        str     w2, [sp,4]
        ldr     w1, [sp,12]
        ldr     w0, [sp,8]
        mul     w1, w1, w0
        ldr     w0, [sp,4]
        add     w0, w1, w0
        add     sp, sp, 16
        ret
```

Le code sauve ses arguments en entrée dans la pile locale, dans le cas où quelqu'un (ou quelque chose) dans cette fonction aurait besoin d'utiliser les registres `W0...W2`. Cela évite d'écraser les arguments originels de la fonction, qui pourraient être de nouveau utilisés par la suite.

Cela est appelé *Zone de sauvegarde de registre.* ([*Procedure Call Standard for the ARM 64-bit Architecture (AArch64)*, (2013)][86]). L'appelée, toutefois, n'est pas obligée de les sauvegarder. C'est un peu similaire au « Shadow Space » : 1.10.2 on page 101.

Pourquoi est-ce que GCC 4.9 avec l'option d'optimisation supprime ce code de sauvegarde? Parce qu'il a fait plus d'optimisation et en a conclu que les arguments de la fonction n'allaient pas être utilisés par la suite et donc que les registres `W0...W2` ne vont pas être utilisés.

Nous avons donc une paire d'instructions `MUL`/`ADD` au lieu d'un seul `MADD`.

[86] Aussi disponible en http://go.yurichev.com/17287

1.10.4 MIPS

Listing 1.93: GCC 4.4.5 avec optimisation

```
.text :00000000 f :
; $a0=a
; $a1=b
; $a2=c
.text :00000000        mult    $a1, $a0
.text :00000004        mflo    $v0
.text :00000008        jr      $ra
.text :0000000C        addu    $v0, $a2, $v0     ; slot de délai de branchement
; au retour le résultat est dans $v0
.text :00000010 main :
.text :00000010
.text :00000010 var_10 = -0x10
.text :00000010 var_4  = -4
.text :00000010
.text :00000010        lui     $gp, (__gnu_local_gp >> 16)
.text :00000014        addiu   $sp, -0x20
.text :00000018        la      $gp, (__gnu_local_gp & 0xFFFF)
.text :0000001C        sw      $ra, 0x20+var_4($sp)
.text :00000020        sw      $gp, 0x20+var_10($sp)
; définir c :
.text :00000024        li      $a2, 3
; définir a :
.text :00000028        li      $a0, 1
.text :0000002C        jal     f
; définir b :
.text :00000030        li      $a1, 2            ; slot de délai de branchement
; le résultat est maintenant dans $v0
.text :00000034        lw      $gp, 0x20+var_10($sp)
.text :00000038        lui     $a0, ($LC0 >> 16)
.text :0000003C        lw      $t9, (printf & 0xFFFF)($gp)
.text :00000040        la      $a0, ($LC0 & 0xFFFF)
.text :00000044        jalr    $t9
; prend le résultat de la fonction f() et le passe en second argument à printf():
.text :00000048        move    $a1, $v0          ; slot de délai de branchement
.text :0000004C        lw      $ra, 0x20+var_4($sp)
.text :00000050        move    $v0, $zero
.text :00000054        jr      $ra
.text :00000058        addiu   $sp, 0x20 ; slot de délai de branchement
```

Les quatre premiers arguments de la fonction sont passés par quatre registres préfixés par A-.

Il y a deux registres spéciaux en MIPS: HI et LO qui sont remplis avec le résultat 64-bit de la multiplication lors de l'exécution d'une instruction `MULT`.

Ces registres sont accessibles seulement en utilisant les instructions `MFLO` et `MFHI`. Ici `MFLO` prend la partie basse du résultat de la multiplication et le stocke dans $V0. Donc la partie haute du résultat de la multiplication est abandonnée (le contenu du registre HI n'est pas utilisé). Effectivement: nous travaillons avec des types de données *int* 32-bit ici.

Enfin, `ADDU` (« Add Unsigned » addition non signée) ajoute la valeur du troisième argument au résultat.

Il y a deux instructions différentes pour l'addition en MIPS: `ADD` et `ADDU`. La différence entre les deux n'est pas relative au fait d'être signé, mais aux exceptions. `ADD` peut déclencher une exception lors d'un débordement, ce qui est parfois utile[87] et supporté en ADA LP, par exemple. `ADDU` ne déclenche pas d'exception lors d'un débordement. Comme C/C++ ne supporte pas ceci, dans notre exemple nous voyons `ADDU` au lieu de `ADD`.

Le résultat 32-bit est laissé dans $V0.

Il y a une instruction nouvelle pour nous dans `main()` : `JAL` (« Jump and Link »).

La différence entre `JAL` et `JALR` est qu'un offset relatif est encodé dans la première instruction, tandis que `JALR` saute à l'adresse absolue stockée dans un registre (« Jump and Link Register »).

Les deux fonctions `f()` et `main()` sont stockées dans le même fichier objet, donc l'adresse relative de `f()` est connue et fixée.

[87] http://go.yurichev.com/17326

1.11 Plus loin sur le renvoi des résultats

En x86, le résultat de l'exécution d'une fonction est d'habitude renvoyé [88] dans le registre EAX.

Si il est de type octet ou un caractère (*char*), alors la partie basse du registre EAX (AL) est utilisée. Si une fonction renvoie un nombre de type *float*, le registre ST(0) du FPU est utilisé. En ARM, d'habitude, le résultat est renvoyé dans le registre R0.

1.11.1 Tentative d'utilisation du résultat d'une fonction renvoyant *void*

Donc, que se passe-t-il si le type de retour de la fonction `main()` a été déclaré du type *void* et non pas *int* ? Ce que l'on nomme le code de démarrage (startup-code) appelle `main()` grosso-modo de la façon suivante:

```
push envp
push argv
push argc
call main
push eax
call exit
```

En d'autres mots:

```
exit(main(argc,argv,envp));
```

Si vous déclarez `main()` comme renvoyant *void*, rien ne sera renvoyé explicitement (en utilisant la déclaration *return*), alors quelque chose d'inconnu, qui aura été stocké dans la registre EAX lors de l'exécution de `main()` sera l'unique argument de la fonction exit(). Il y aura probablement une valeur aléatoire, laissée lors de l'exécution de la fonction, donc le code de retour du programme est pseudo-aléatoire.

Illustrons ce fait: Notez bien que la fonction `main()` a un type de retour *void* :

```
#include <stdio.h>

void main()
{
        printf ("Hello, world!\n");
};
```

Compilons-le sous Linux.

GCC 4.8.1 a remplacé `printf()` par `puts()` (nous avons vu ceci avant: 1.5.4 on page 22), mais c'est OK, puisque `puts()` renvoie le nombre de caractères écrit, tout comme `printf()`. Remarquez que le registre EAX n'est pas mis à zéro avant la fin de `main()`.

Ceci implique que la valeur de EAX à la fin de `main()` contient ce que `puts()` y avait mis.

Listing 1.94: GCC 4.8.1

```
.LC0 :
        .string "Hello, world!"
main :
        push    ebp
        mov     ebp, esp
        and     esp, -16
        sub     esp, 16
        mov     DWORD PTR [esp], OFFSET FLAT :.LC0
        call    puts
        leave
        ret
```

Écrivons un script bash affichant le code de retour:

Listing 1.95: tst.sh

```
#!/bin/sh
./hello_world
echo \$?
```

[88] Voir également : MSDN: Return Values (C++): MSDN

Et lançons le:

```
\$ tst.sh
Hello, world!
14
```

14 est le nombre de caractères écrits. Le nombre de caractères affichés est *passé* de `printf()`, à travers EAX/RAX, dans le « code de retour ».

À propos, lorsque l'on décompile du C++ dans Hex-Rays, nous rencontrons souvent une fonction qui se termine par un destructeur d'une classe:

```
...

call    ??1CString@@QAE@XZ ; CString ::~CString(void)
mov     ecx, [esp+30h+var_C]
pop     edi
pop     ebx
mov     large fs :0, ecx
add     esp, 28h
retn
```

Dans le standard C++, le destructeur ne renvoie rien, mais lorsque Hex-Rays n'en sait rien, et pense que le destructeur et cette fonction renvoient tout deux un *int*

```
...

        return CString ::~CString(&Str);
}
```

1.11.2 Que se passe-t-il si on n'utilise pas le résultat de la fonction?

`printf()` renvoie le nombre de caractères écrit avec succès, mais, en pratique, ce résultat est rarement utilisé.

Il est aussi possible d'appeler une fonction dont la finalité est de renvoyer une valeur, et de ne pas l'utiliser:

```
int f()
{
    // skip first 3 random values :
    rand();
    rand();
    rand();
    // and use 4th :
    return rand();
};
```

Le résultat de la fonction rand() est mis dans EAX, dans les quatre cas.

Mais dans les 3 premiers, la valeur dans EAX n'est pas utilisée.

1.11.3 Renvoyer une structure

Revenons au fait que la valeur de retour est passée par le registre EAX.

C'est pourquoi les vieux compilateurs C ne peuvent pas créer de fonction capable de renvoyer quelque chose qui ne tient pas dans un registre (d'habitude *int*), mais si besoin, les informations doivent être renvoyées via un pointeur passé en argument.

Donc, d'habitude, si une fonction doit renvoyer plusieurs valeurs, elle en renvoie une seule, et le reste—par des pointeurs.

Maintenant, il est possible de renvoyer, disons, une structure entière, mais ce n'est pas encore très populaire. Si une fonction doit renvoyer une grosse structure, la fonction appelante doit l'allouer et passer un pointeur sur cette dernière via le premier argument, de manière transparente pour le programmeur. C'est presque la même chose que de passer un pointeur manuellement dans le premier argument, mais le compilateur le cache.

Petit exemple:

```
struct s
{
    int a;
    int b;
    int c;
};

struct s get_some_values (int a)
{
    struct s rt;

    rt.a=a+1;
    rt.b=a+2;
    rt.c=a+3;

    return rt;
};
```

...ce que nous obtenons (MSVC 2010 /0x):

```
$T3853 = 8                      ; size = 4
_a$ = 12                        ; size = 4
?get_some_values@@YA?AUs@@H@Z PROC              ; get_some_values
    mov    ecx, DWORD PTR _a$[esp-4]
    mov    eax, DWORD PTR $T3853[esp-4]
    lea    edx, DWORD PTR [ecx+1]
    mov    DWORD PTR [eax], edx
    lea    edx, DWORD PTR [ecx+2]
    add    ecx, 3
    mov    DWORD PTR [eax+4], edx
    mov    DWORD PTR [eax+8], ecx
    ret    0
?get_some_values@@YA?AUs@@H@Z ENDP              ; get_some_values
```

Ici, le nom de la macro interne pour passer le pointeur sur une structure est $T3853.

Cet exemple peut être récrit en utilisant les extensions C99 du langage:

```
struct s
{
    int a;
    int b;
    int c;
};

struct s get_some_values (int a)
{
    return (struct s){.a=a+1, .b=a+2, .c=a+3};
};
```

Listing 1.96: GCC 4.8.1

```
_get_some_values proc near

ptr_to_struct   = dword ptr  4
a               = dword ptr  8

                mov     edx, [esp+a]
                mov     eax, [esp+ptr_to_struct]
                lea     ecx, [edx+1]
                mov     [eax], ecx
                lea     ecx, [edx+2]
                add     edx, 3
                mov     [eax+4], ecx
                mov     [eax+8], edx
                retn
_get_some_values endp
```

Comme on le voit, la fonction remplit simplement les champs de la structure allouée par la fonction appelante, comme si un pointeur sur la structure avait été passé. Donc, il n'y a pas d'impact négatif sur les performances.

1.12 Pointeurs

1.12.1 Échanger les valeurs en entrée

Ceci fait ce que l'on veut:

```
#include <memory.h>
#include <stdio.h>

void swap_bytes (unsigned char* first, unsigned char* second)
{
        unsigned char tmp1;
        unsigned char tmp2;

        tmp1=*first;
        tmp2=*second;

        *first=tmp2;
        *second=tmp1;
};

int main()
{
        // copy string into heap, so we will be able to modify it
        char *s=strdup("string");

        // swap 2nd and 3rd characters
        swap_bytes (s+1, s+2);

        printf ("%s\n", s);
};
```

Comme on le voit, les octets sont chargés dans la partie 8-bit basse de ECX et EBX en utilisant MOVZX (donc les parties hautes de ces registres vont être effacées) et ensuite les octets échangés sont récrits.

Listing 1.97: GCC 5.4 avec optimisation

```
swap_bytes :
        push    ebx
        mov     edx, DWORD PTR [esp+8]
        mov     eax, DWORD PTR [esp+12]
        movzx   ecx, BYTE PTR [edx]
        movzx   ebx, BYTE PTR [eax]
        mov     BYTE PTR [edx], bl
        mov     BYTE PTR [eax], cl
        pop     ebx
        ret
```

Les adresses des deux octets sont lues depuis les arguments et durant l'exécution de la fonction sont copiés dans EDX et EAX.

Donc nous utilisons des pointeurs, il n'y a sans doute pas de meilleure façon de réaliser cette tâche sans eux.

1.12.2 Renvoyer des valeurs

Les pointeurs sont souvent utilisés pour renvoyer des valeurs depuis les fonctions (rappelez-vous le cas (1.9 on page 66) de `scanf()`).

Par exemple, lorsqu'une fonction doit renvoyer deux valeurs.

Exemple avec des variables globales

```
#include <stdio.h>

void f1 (int x, int y, int *sum, int *product)
{
        *sum=x+y;
        *product=x*y;
};

int sum, product;

void main()
{
        f1(123, 456, &sum, &product);
        printf ("sum=%d, product=%d\n", sum, product);
};
```

Ceci se compile en:

Listing 1.98: MSVC 2010 avec optimisation (/Ob0)

```
COMM    _product :DWORD
COMM    _sum :DWORD
$SG2803 DB      'sum=%d, product=%d', 0aH, 00H

_x$ = 8                 ; size = 4
_y$ = 12                ; size = 4
_sum$ = 16              ; size = 4
_product$ = 20          ; size = 4
_f1     PROC
        mov     ecx, DWORD PTR _y$[esp-4]
        mov     eax, DWORD PTR _x$[esp-4]
        lea     edx, DWORD PTR [eax+ecx]
        imul    eax, ecx
        mov     ecx, DWORD PTR _product$[esp-4]
        push    esi
        mov     esi, DWORD PTR _sum$[esp]
        mov     DWORD PTR [esi], edx
        mov     DWORD PTR [ecx], eax
        pop     esi
        ret     0
_f1     ENDP

_main   PROC
        push    OFFSET _product
        push    OFFSET _sum
        push    456     ; 000001c8H
        push    123     ; 0000007bH
        call    _f1
        mov     eax, DWORD PTR _product
        mov     ecx, DWORD PTR _sum
        push    eax
        push    ecx
        push    OFFSET $SG2803
        call    DWORD PTR __imp__printf
        add     esp, 28
        xor     eax, eax
        ret     0
_main   ENDP
```

Regardons ceci dans OllyDbg :

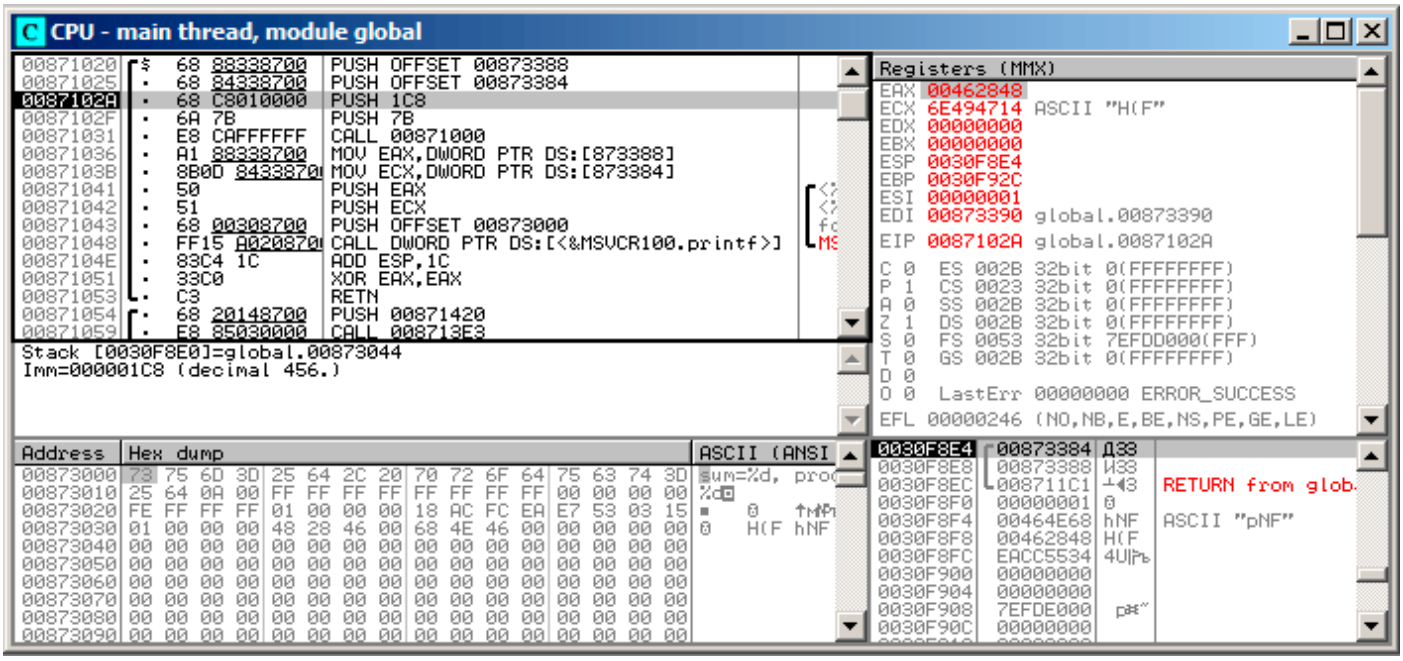


Fig. 1.24: OllyDbg : les adresses des variables globales sont passées à `f1()`

Tout d'abord, les adresses des variables globales sont passées à `f1()`. Nous pouvons cliquer sur « Follow in dump » sur l'élément de la pile, et nous voyons l'espace alloué dans le segment de données pour les deux variables.

Ces variables sont mises à zéro, car les données non-initialisées (de BSS) sont effacées avant le début de l'exécution, [voir *ISO/IEC 9899:TC3 (C C99 standard)*, (2007) 6.7.8p10].

Elles se trouvent dans le segment de données, nous pouvons le vérifier en appuyant sur Alt-M et en regardant la carte de la mémoire:

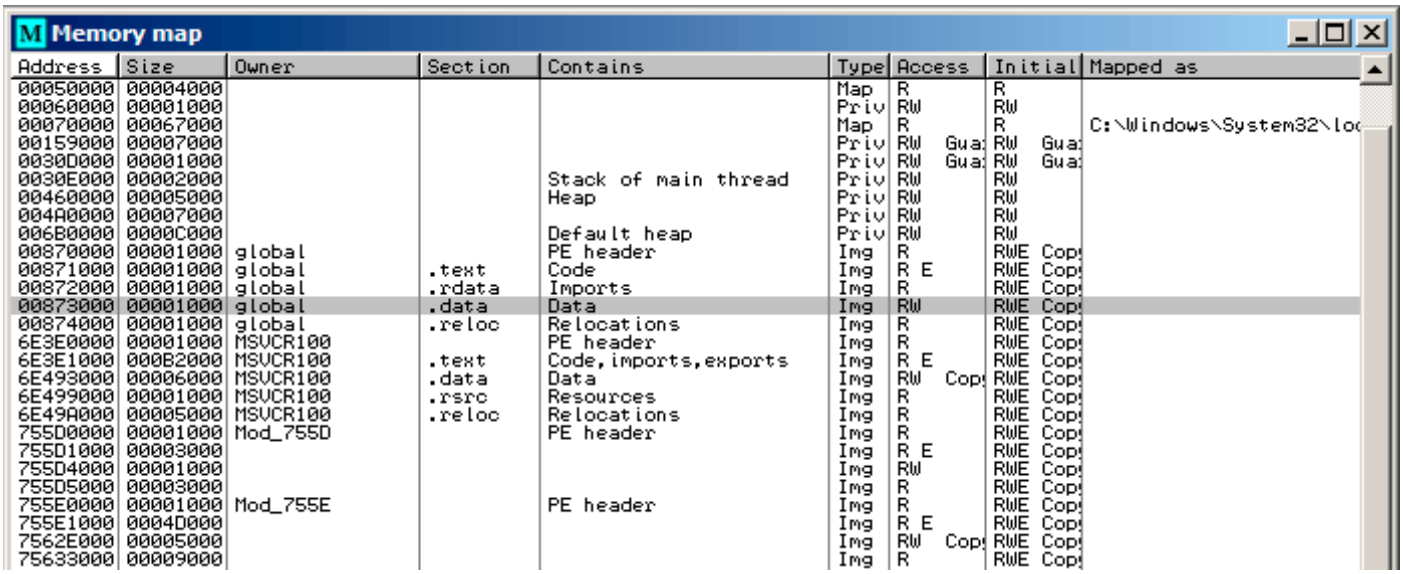


Fig. 1.25: OllyDbg : carte de la mémoire

Traçons l'exécution (F7) jusqu'au début de `f1()` :

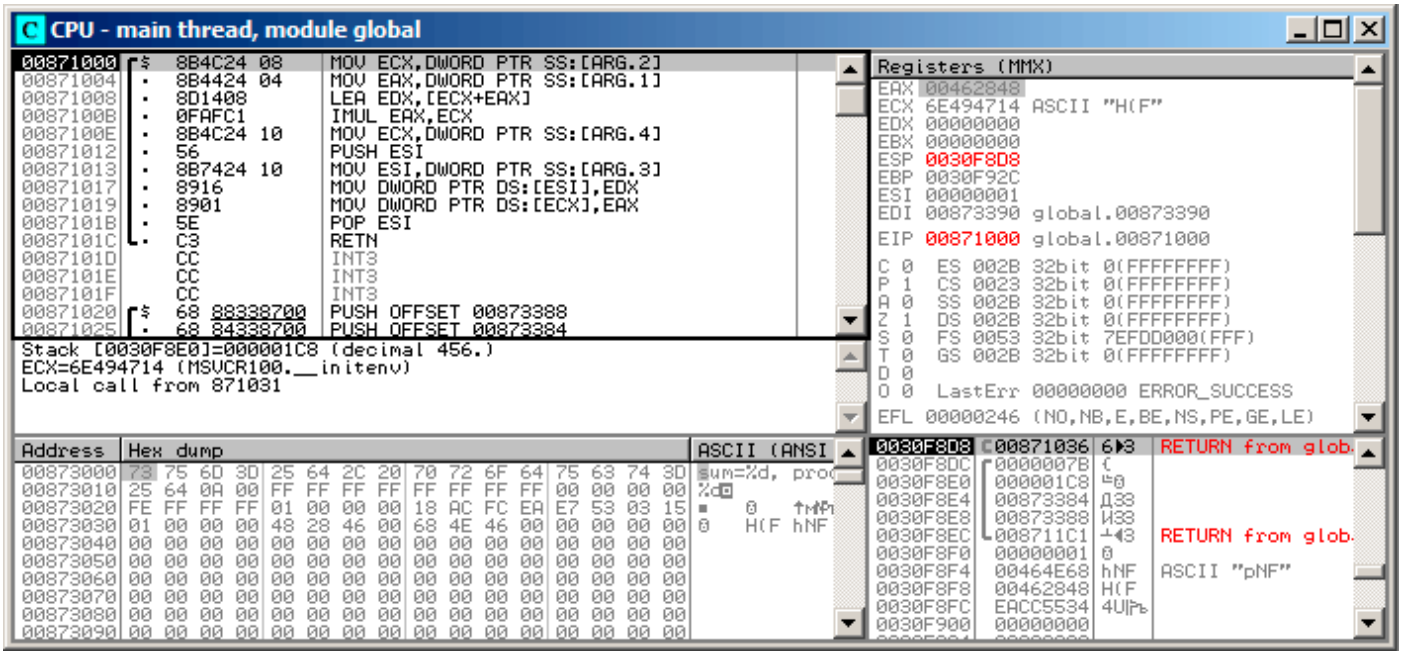


Fig. 1.26: OllyDbg : `f1()` commence

Deux valeurs sont visibles sur la pile: 456 (`0x1C8`) et 123 (`0x7B`), et aussi les adresses des deux variables globales.

Suivons l'exécution jusqu'à la fin de `f1()`. Dans la fenêtre en bas à gauche, nous voyons comment le résultat du calcul apparaît dans les variables globales:

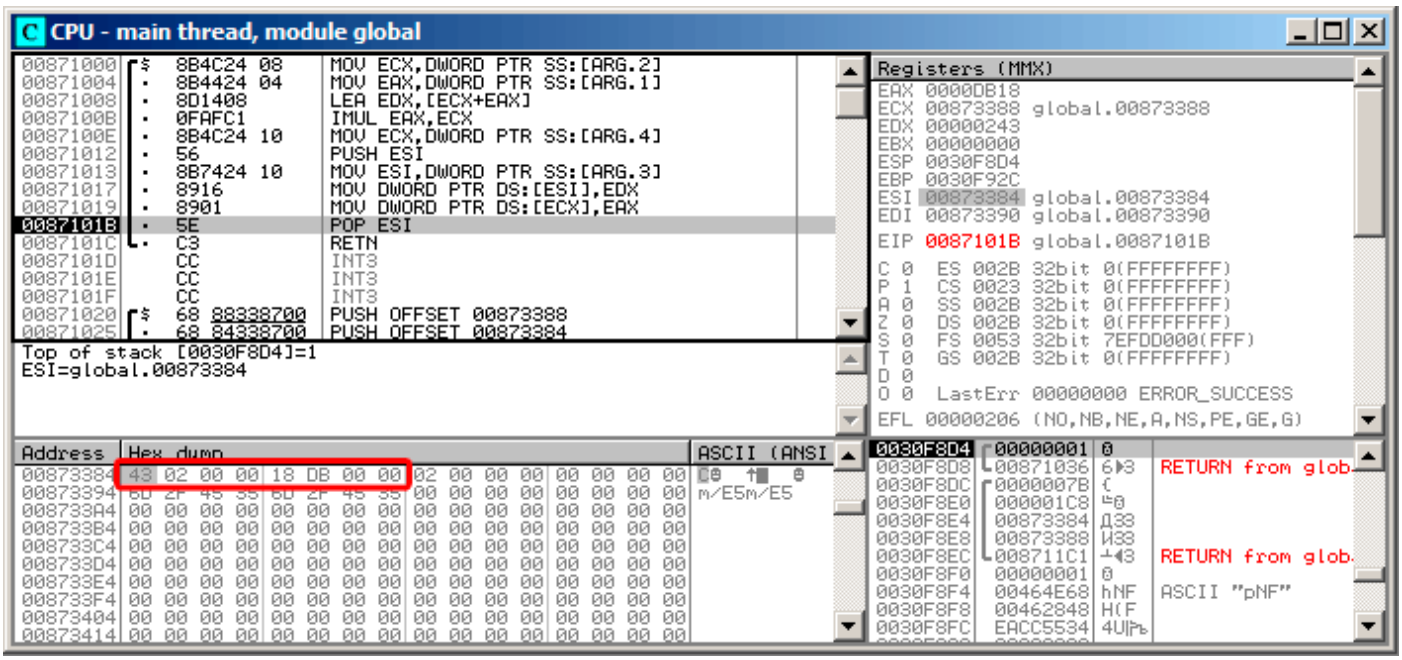


Fig. 1.27: OllyDbg : l'exécution de `f1()` est terminée

Maintenant les valeurs des variables globales sont chargées dans des registres, prêtes à être passées à `printf()` (via la pile):

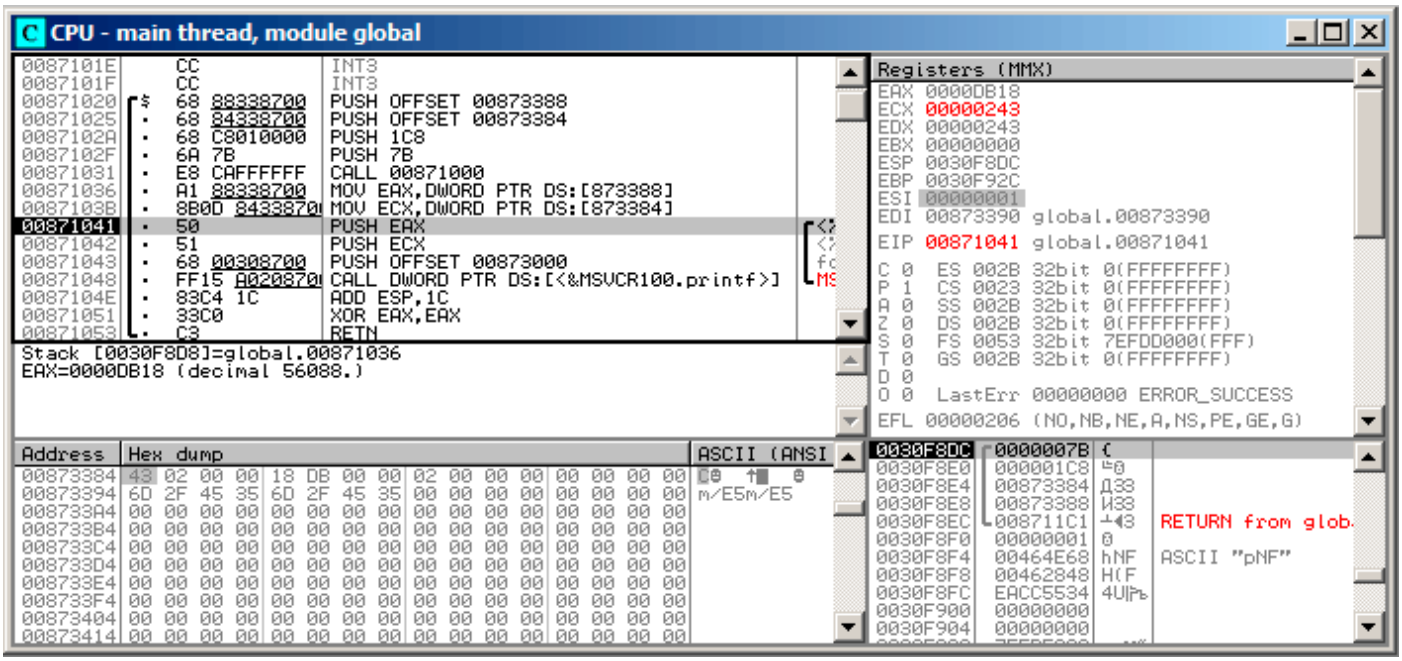


Fig. 1.28: OllyDbg : les adresses des variables globales sont passées à `printf()`

Exemple avec des variables locales

Modifions légèrement notre exemple:

Listing 1.99: maintenant les variables `sum` et `product` sont locales

```
void main()
{
        int sum, product; // maintenant, les variables sont locales à la fonction

        f1(123, 456, &sum, &product);
        printf ("sum=%d, product=%d\n", sum, product);
};
```

Le code de `f1()` ne va pas changer. Seul celui de `main()` va changer:

Listing 1.100: MSVC 2010 avec optimisation (/Ob0)

```
_product$ = -8          ; size = 4
_sum$ = -4              ; size = 4
_main   PROC
; Line 10
        sub     esp, 8
; Line 13
        lea     eax, DWORD PTR _product$[esp+8]
        push    eax
        lea     ecx, DWORD PTR _sum$[esp+12]
        push    ecx
        push    456     ; 000001c8H
        push    123     ; 0000007bH
        call    _f1
; Line 14
        mov     edx, DWORD PTR _product$[esp+24]
        mov     eax, DWORD PTR _sum$[esp+24]
        push    edx
        push    eax
        push    OFFSET $SG2803
        call    DWORD PTR __imp__printf
; Line 15
        xor     eax, eax
```

```
        add     esp, 36
        ret     0
```

Regardons à nouveau avec OllyDbg. Les adresses des variables locales dans la pile sont 0x2EF854 et 0x2EF858. Voyons comment elles sont poussées sur la pile:

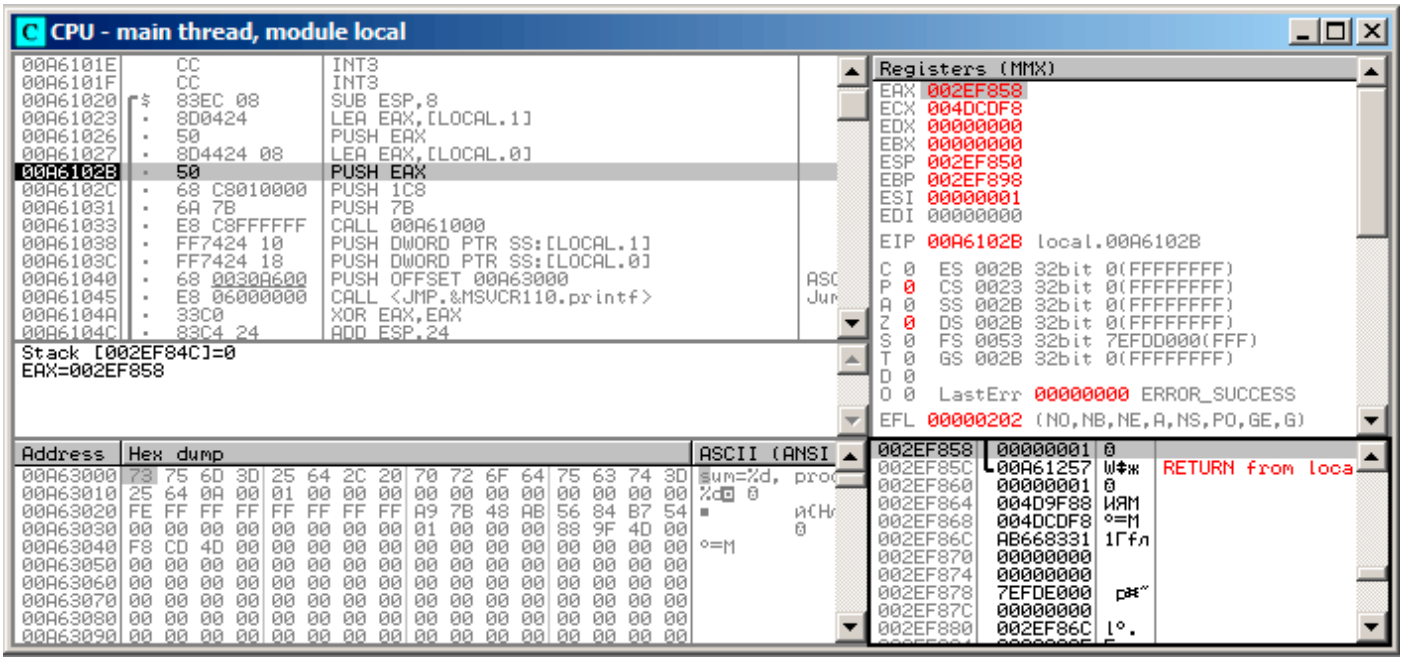


Fig. 1.29: OllyDbg : les adresses des variables locales sont poussées sur la pile

f1() commence. Jusqu'ici, il n'y a que des restes de données sur la pile en 0x2EF854 et 0x2EF858 :

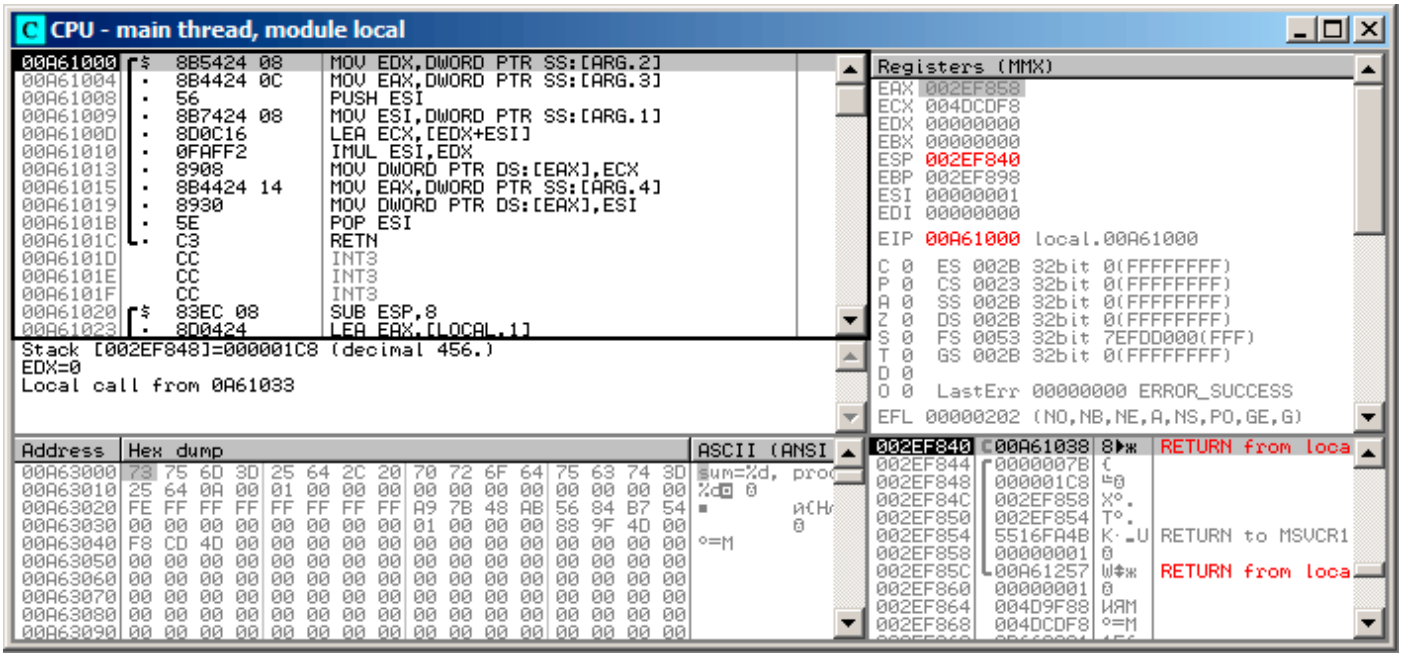


Fig. 1.30: OllyDbg : f1() commence

f1() se termine:

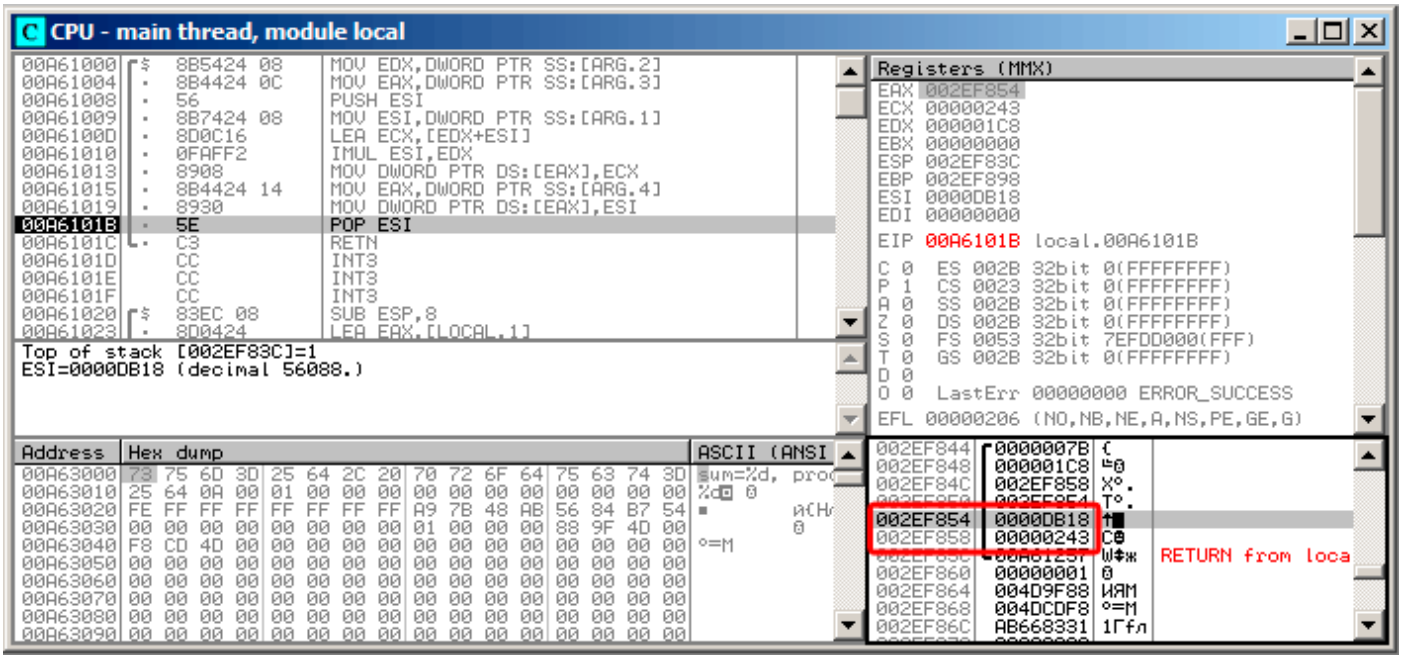


Fig. 1.31: OllyDbg : f1() termine son exécution

Nous trouvons maintenant 0xDB18 et 0x243 aux adresses 0x2EF854 et 0x2EF858. Ces valeurs sont les résultats de f1().

Conclusion

f1() peut renvoyer des pointeurs sur n'importe quel emplacement en mémoire, situés n'importe où.

C'est par essence l'utilité des pointeurs.

À propos, les *references* C++ fonctionnent exactement pareil. Voir à ce propos: (**??** on page ??).

1.13 Opérateur GOTO

L'opérateur GOTO est en général considéré comme un anti-pattern, voir [Edgar Dijkstra, *Go To Statement Considered Harmful* (1968)[89]]. Néanmoins, il peut être utilisé raisonnablement, voir [Donald E. Knuth, *Structured Programming with go to Statements* (1974)[90]] [91].

Voici un exemple très simple:

```
#include <stdio.h>

int main()
{
        printf ("begin\n");
        goto exit;
        printf ("skip me!\n");
exit :
        printf ("end\n");
};
```

Voici ce que nous obtenons avec MSVC 2012:

[89] http://yurichev.com/mirrors/Dijkstra68.pdf
[90] http://yurichev.com/mirrors/KnuthStructuredProgrammingGoTo.pdf
[91] [Dennis Yurichev, *C/C++ programming language notes*] a aussi quelques exemples.

Listing 1.101: MSVC 2012

```
$SG2934 DB      'begin', 0aH, 00H
$SG2936 DB      'skip me!', 0aH, 00H
$SG2937 DB      'end', 0aH, 00H

_main   PROC
        push    ebp
        mov     ebp, esp
        push    OFFSET $SG2934 ; 'begin'
        call    _printf
        add     esp, 4
        jmp     SHORT $exit$3
        push    OFFSET $SG2936 ; 'skip me!'
        call    _printf
        add     esp, 4
$exit$3 :
        push    OFFSET $SG2937 ; 'end'
        call    _printf
        add     esp, 4
        xor     eax, eax
        pop     ebp
        ret     0
_main   ENDP
```

L'instruction *goto* a simplement été remplacée par une instruction JMP, qui a le même effet: un saut inconditionnel à un autre endroit. Le second `printf()` peut seulement être exécuté avec une intervention humaine, en utilisant un débogueur ou en modifiant le code.

Cela peut être utile comme exercice simple de patching. Ouvrons l'exécutable généré dans Hiew:

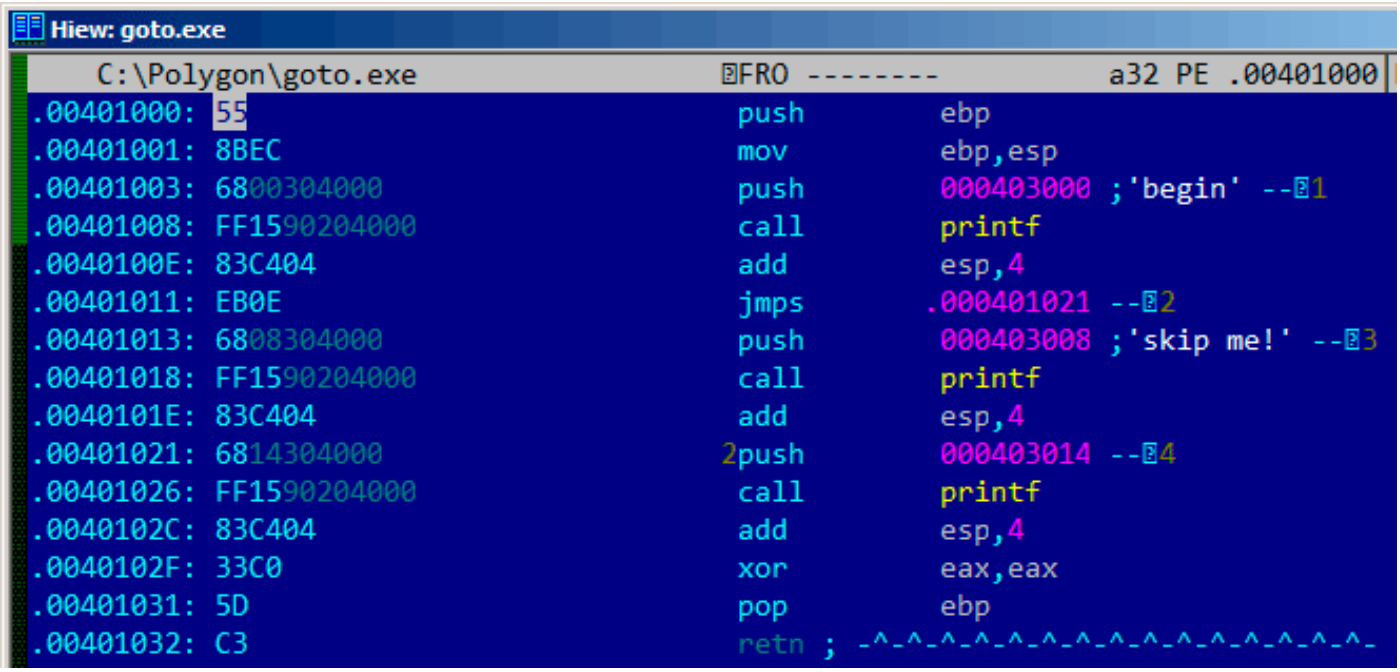


Fig. 1.32: Hiew

Placez le curseur à l'adresse du JMP (0x410), pressez F3 (edit), pressez deux fois zéro, donc l'opcode devient EB 00 :

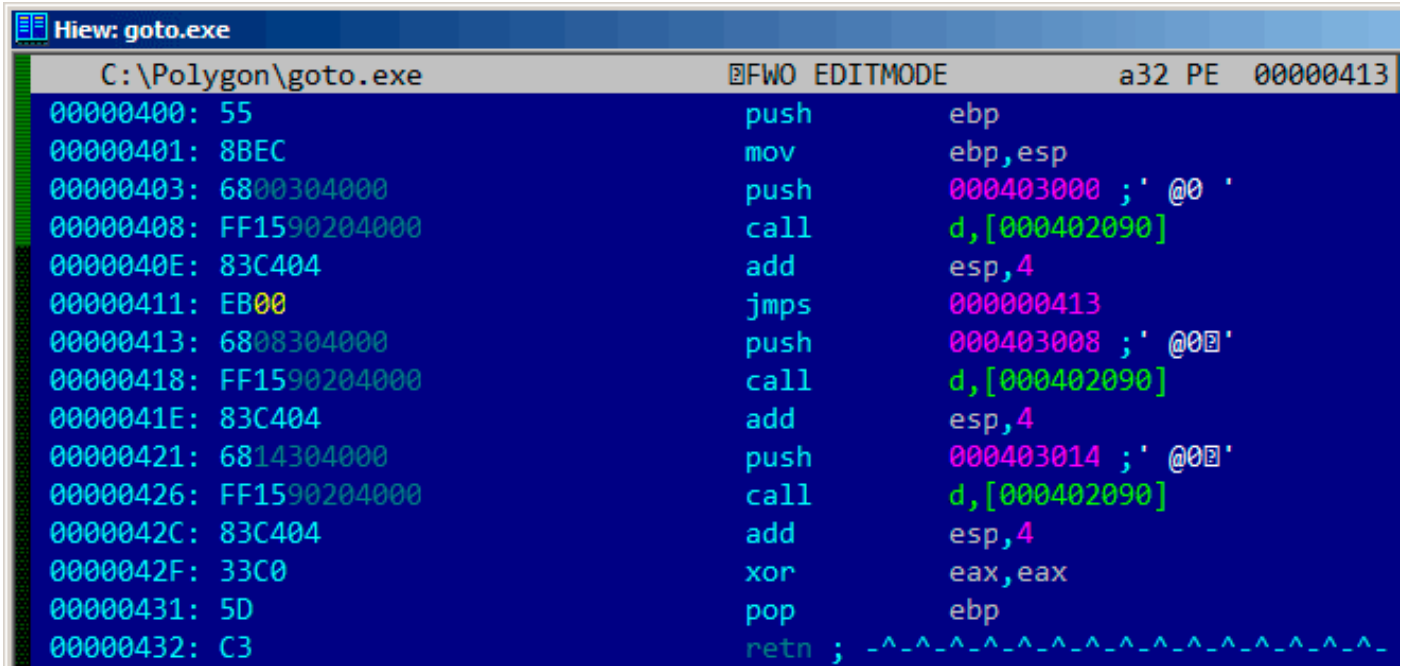


Fig. 1.33: Hiew

Le second octet de l'opcode de JMP indique l'offset relatif du saut, 0 signifie le point juste après l'instruction courante.

Donc maintenant JMP n'évite plus le second `printf()`.

Pressez F9 (save) et quittez. Maintenant, si nous lançons l'exécutable, nous verrons ceci:

Listing 1.102: Sortie de l'exécutable modifié

```
C :\...>goto.exe

begin
skip me!
end
```

Le même résultat peut être obtenu en remplaçant l'instruction JMP par 2 instructions NOP.

NOP a un opcode de 0x90 et une longueur de 1 octet, donc nous en avons besoin de 2 pour remplacer JMP (qui a une taille de 2 octets).

1.13.1 Code mort

Le second appel à `printf()` est aussi appelé « code mort » en terme de compilateur.

Cela signifie que le code ne sera jamais exécuté. Donc lorsque vous compilez cet exemple avec les optimisations, le compilateur supprime le « code mort », n'en laissant aucune trace:

Listing 1.103: MSVC 2012 avec optimisation

```
$SG2981 DB      'begin', 0aH, 00H
$SG2983 DB      'skip me!', 0aH, 00H
$SG2984 DB      'end', 0aH, 00H

_main   PROC
        push    OFFSET $SG2981 ; 'begin'
        call    _printf
        push    OFFSET $SG2984 ; 'end'
$exit$4 :
        call    _printf
        add     esp, 8
```

```
        xor     eax, eax
        ret     0
_main   ENDP
```

Toutefois, le compilateur a oublié de supprimer la chaîne « skip me! ».

1.13.2 Exercice

Essayez d'obtenir le même résultat en utilisant votre compilateur et votre débogueur favoris.

1.14 Saut conditionnels

1.14.1 Exemple simple

```
#include <stdio.h>

void f_signed (int a, int b)
{
    if (a>b)
        printf ("a>b\n");
    if (a==b)
        printf ("a==b\n");
    if (a<b)
        printf ("a<b\n");
};

void f_unsigned (unsigned int a, unsigned int b)
{
    if (a>b)
        printf ("a>b\n");
    if (a==b)
        printf ("a==b\n");
    if (a<b)
        printf ("a<b\n");
};

int main()
{
    f_signed(1, 2);
    f_unsigned(1, 2);
    return 0;
};
```

x86

x86 + MSVC

Voici à quoi ressemble la fonction `f_signed()` :

Listing 1.104: MSVC 2010 sans optimisation

```
_a$ = 8
_b$ = 12
_f_signed PROC
    push   ebp
    mov    ebp, esp
    mov    eax, DWORD PTR _a$[ebp]
    cmp    eax, DWORD PTR _b$[ebp]
    jle    SHORT $LN3@f_signed
    push   OFFSET $SG737        ; 'a>b'
    call   _printf
    add    esp, 4
```

```
$LN3@f_signed :
    mov    ecx, DWORD PTR _a$[ebp]
    cmp    ecx, DWORD PTR _b$[ebp]
    jne    SHORT $LN2@f_signed
    push   OFFSET $SG739        ; 'a==b'
    call   _printf
    add    esp, 4
$LN2@f_signed :
    mov    edx, DWORD PTR _a$[ebp]
    cmp    edx, DWORD PTR _b$[ebp]
    jge    SHORT $LN4@f_signed
    push   OFFSET $SG741        ; 'a<b'
    call   _printf
    add    esp, 4
$LN4@f_signed :
    pop    ebp
    ret    0
_f_signed ENDP
```

La première instruction, JLE, représente *Jump if Less or Equal* (saut si inférieur ou égal). En d'autres mots, si le deuxième opérande est plus grand ou égal au premier, le flux d'exécution est passé à l'adresse ou au label spécifié dans l'instruction. Si la condition ne déclenche pas le saut, car le second opérande est plus petit que le premier, le flux d'exécution ne sera pas altéré et le premier `printf()` sera exécuté. Le second test est JNE : *Jump if Not Equal* (saut si non égal). Le flux d'exécution ne changera pas si les opérandes sont égaux.

Le troisième test est JGE : *Jump if Greater or Equal*—saute si le premier opérande est supérieur ou égal au deuxième. Donc, si les trois sauts conditionnels sont effectués, aucun des appels à `printf()` ne sera exécuté. Ceci est impossible sans intervention spéciale. Regardons maintenant la fonction `f_unsigned()`. La fonction `f_unsigned()` est la même que `f_signed()`, à la différence que les instructions JBE et JAE sont utilisées à la place de JLE et JGE, comme suit:

Listing 1.105: GCC

```
_a$ = 8   ; size = 4
_b$ = 12  ; size = 4
_f_unsigned PROC
    push   ebp
    mov    ebp, esp
    mov    eax, DWORD PTR _a$[ebp]
    cmp    eax, DWORD PTR _b$[ebp]
    jbe    SHORT $LN3@f_unsigned
    push   OFFSET $SG2761    ; 'a>b'
    call   _printf
    add    esp, 4
$LN3@f_unsigned :
    mov    ecx, DWORD PTR _a$[ebp]
    cmp    ecx, DWORD PTR _b$[ebp]
    jne    SHORT $LN2@f_unsigned
    push   OFFSET $SG2763    ; 'a==b'
    call   _printf
    add    esp, 4
$LN2@f_unsigned :
    mov    edx, DWORD PTR _a$[ebp]
    cmp    edx, DWORD PTR _b$[ebp]
    jae    SHORT $LN4@f_unsigned
    push   OFFSET $SG2765    ; 'a<b'
    call   _printf
    add    esp, 4
$LN4@f_unsigned :
    pop    ebp
    ret    0
_f_unsigned ENDP
```

Comme déjà mentionné, les instructions de branchement sont différentes: JBE—*Jump if Below or Equal* (saut si inférieur ou égal) et JAE—*Jump if Above or Equal* (saut si supérieur ou égal). Ces instructions (JA/JAE/JB/JBE) diffèrent de JG/JGE/JL/JLE par le fait qu'elles travaillent avec des nombres non signés.

Voir aussi la section sur la représentation des nombres signés (2.2 on page 456). C'est pourquoi si nous voyons que JG/JL sont utilisés à la place de JA/JB ou vice-versa, nous pouvons être presque sûr que les

variables sont signées ou non signées, respectivement. Voici la fonction `main()`, où presque rien n'est nouveau pour nous:

Listing 1.106: `main()`

```
_main   PROC
        push    ebp
        mov     ebp, esp
        push    2
        push    1
        call    _f_signed
        add     esp, 8
        push    2
        push    1
        call    _f_unsigned
        add     esp, 8
        xor     eax, eax
        pop     ebp
        ret     0
_main   ENDP
```

x86 + MSVC + OllyDbg

Nous pouvons voir comment les flags sont mis en lançant cet exemple dans OllyDbg. Commençons par `f_unsigned()`, qui utilise des entiers non signés.

`CMP` est exécuté trois fois ici, mais avec les même arguments, donc les flags sont les même à chaque fois.

Résultat de la première comparaison:

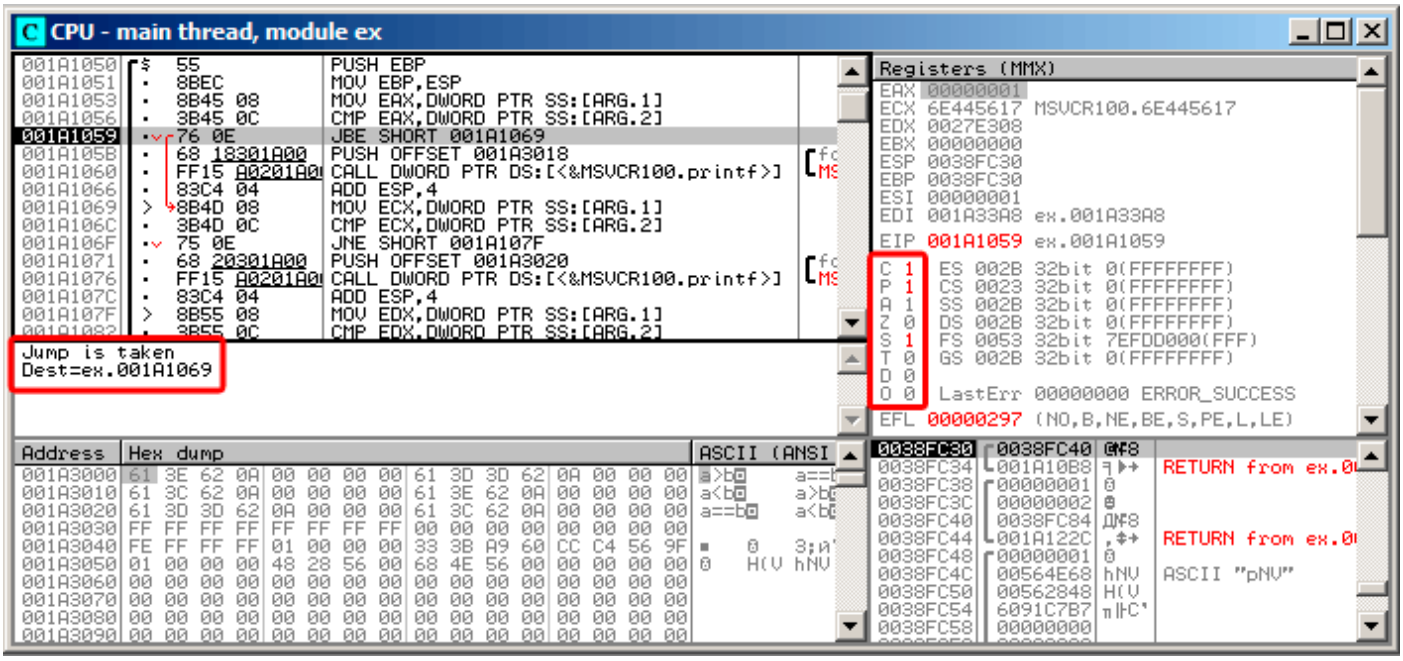


Fig. 1.34: OllyDbg : `f_unsigned()` : premier saut conditionnel

Donc, les flags sont: C=1, P=1, A=1, Z=0, S=1, T=0, D=0, O=0.

Ils sont nommés avec une seule lettre dans OllyDbg par concision.

OllyDbg indique que le saut (JBE) va être suivi maintenant. En effet, si nous regardons dans le manuel d'Intel (8.1.4 on page 649), nous pouvons lire que JBE est déclenchée si CF=1 ou ZF=1. La condition est vraie ici, donc le saut est effectué.

Le saut conditionnel suivant:

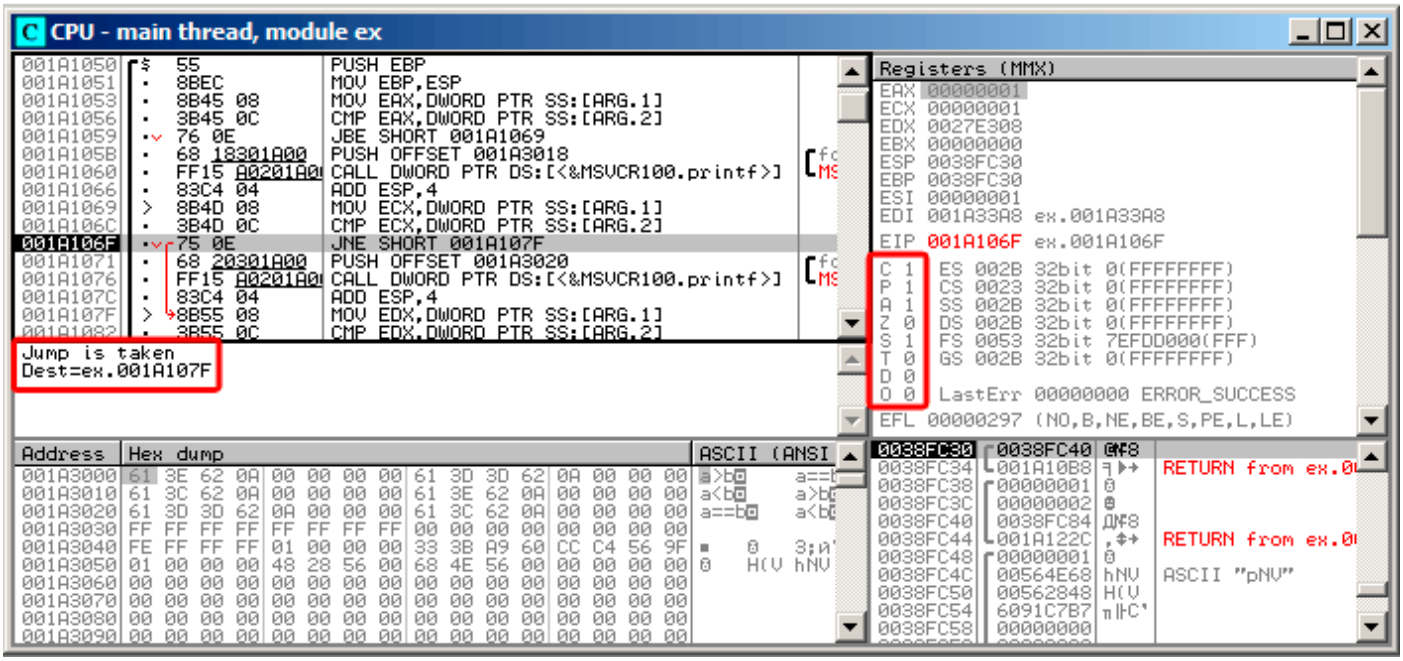


Fig. 1.35: OllyDbg : `f_unsigned()` : second saut conditionnel

OllyDbg indique que le saut JNZ va être effectué maintenant. En effet, JNZ est déclenché si ZF=0 (Zero Flag).

Le troisième saut conditionnel, JNB :

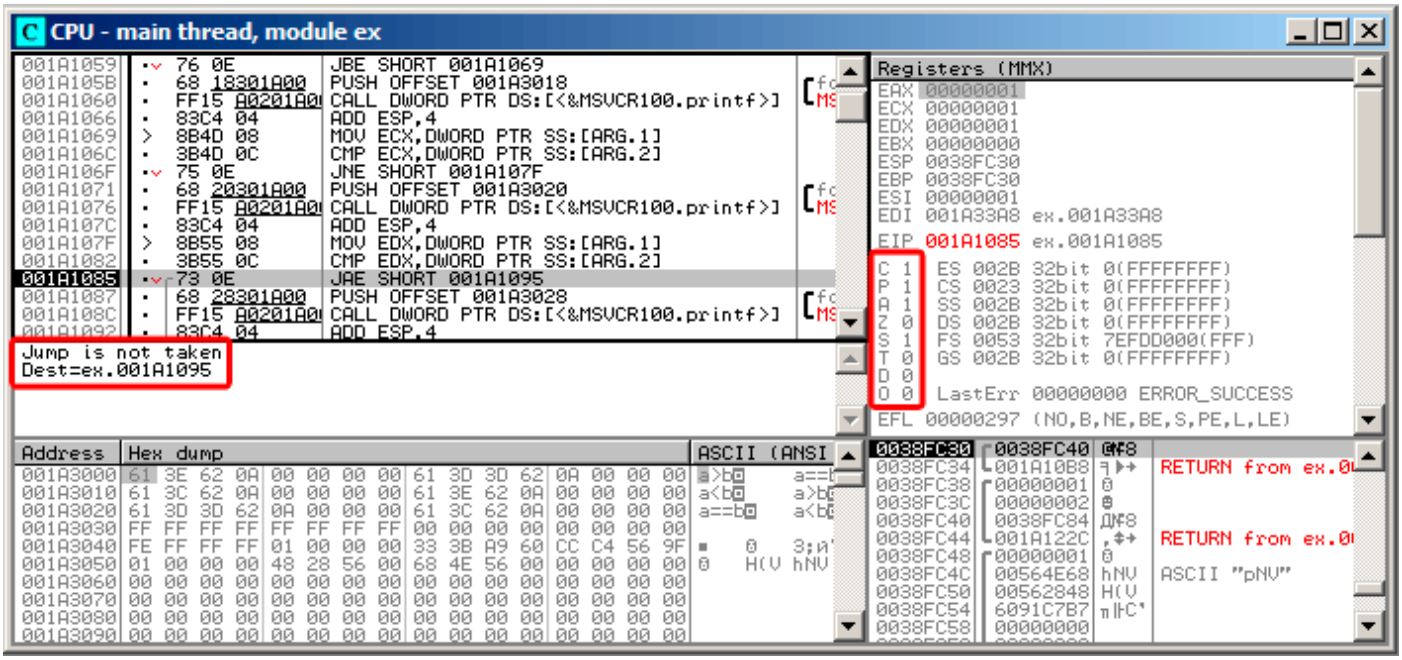


Fig. 1.36: OllyDbg : `f_unsigned()` : troisième saut conditionnel

Dans le manuel d'Intel (8.1.4 on page 649) nous pouvons voir que JNB est déclenché si CF=0 (Carry Flag - flag de retenue). Ce qui n'est pas vrai dans notre cas, donc le troisième `printf()` sera exécuté.

Maintenant, regardons la fonction `f_signed()`, qui fonctionne avec des entiers non signés. Les flags sont mis de la même manière: C=1, P=1, A=1, Z=0, S=1, T=0, D=0, O=0. Le premier saut conditionnel JLE est effectué:

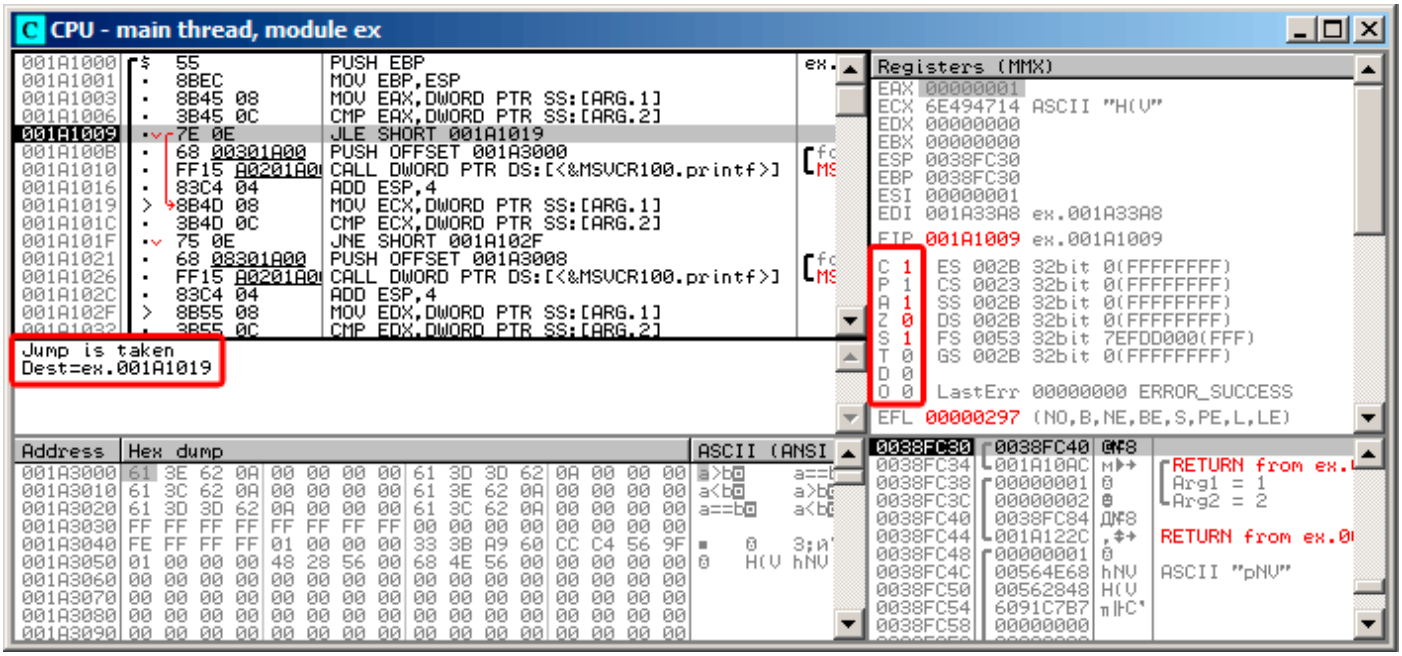


Fig. 1.37: OllyDbg : `f_signed()` : premier saut conditionnel

Dans les manuels d'Intel (8.1.4 on page 649) nous trouvons que cette instruction est déclenchée si ZF=1 ou SF≠OF. SF≠OF dans notre cas, donc le saut est effectué.

Le second saut conditionnel, JNZ, est effectué: si ZF=0 (Zero Flag):

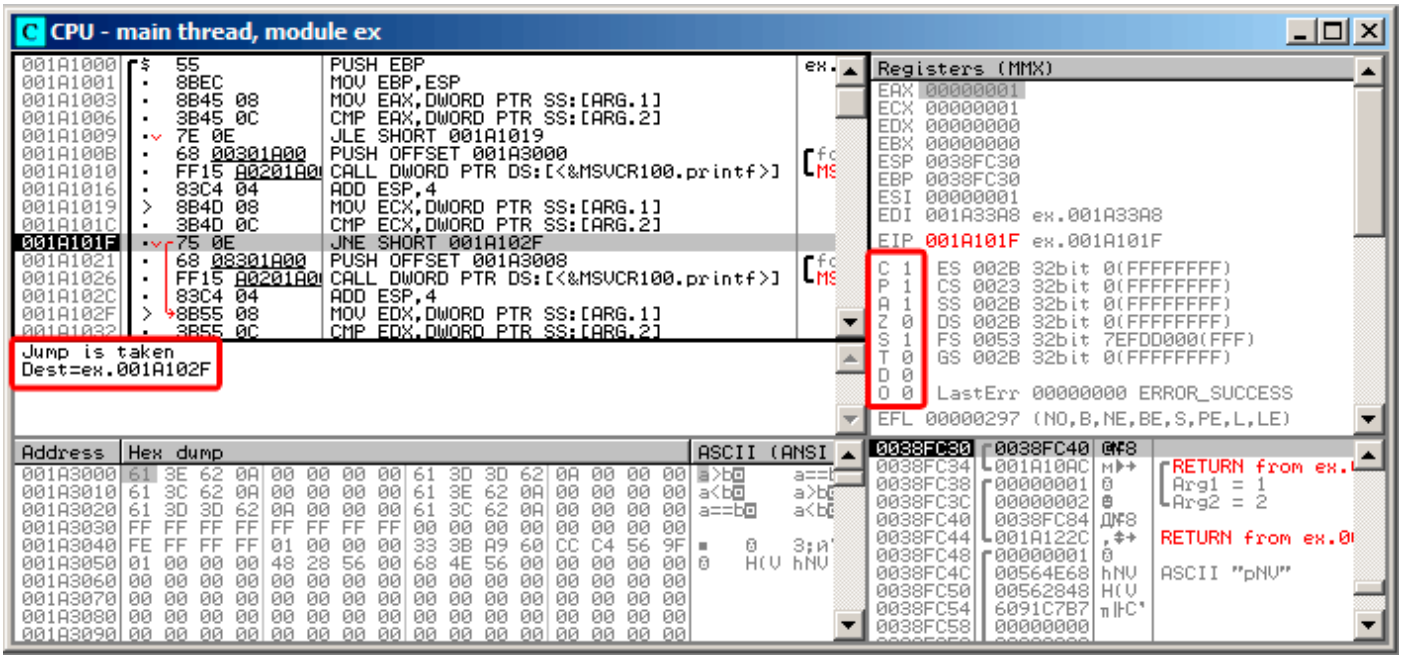


Fig. 1.38: OllyDbg : `f_signed()` : second saut conditionnel

Le troisième saut conditionnel, JGE, ne sera pas effectué car il ne l'est que si SF=OF, et ce n'est pas vrai dans notre cas:

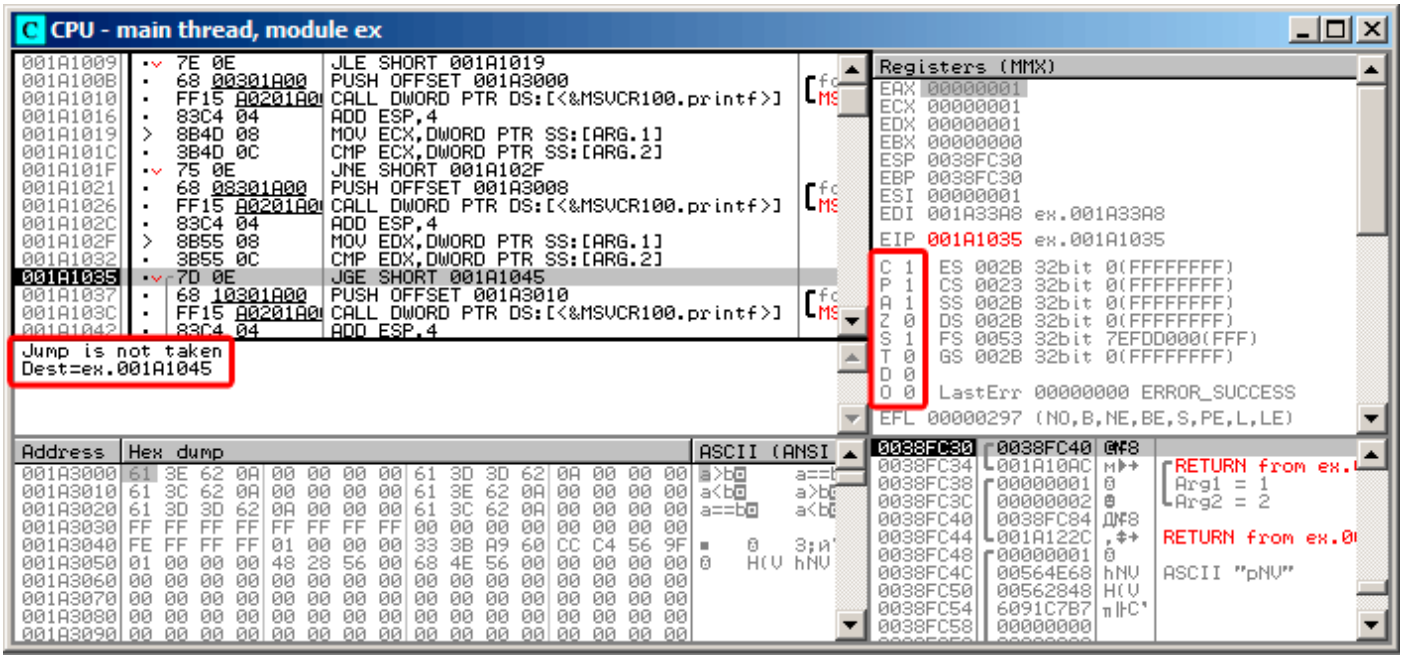


Fig. 1.39: OllyDbg : `f_signed()` : troisième saut conditionnel

x86 + MSVC + Hiew

Nous pouvons essayer de patcher l'exécutable afin que la fonction `f_unsigned()` affiche toujours « a==b », quelque soient les valeurs en entrée. Voici à quoi ça ressemble dans Hiew:

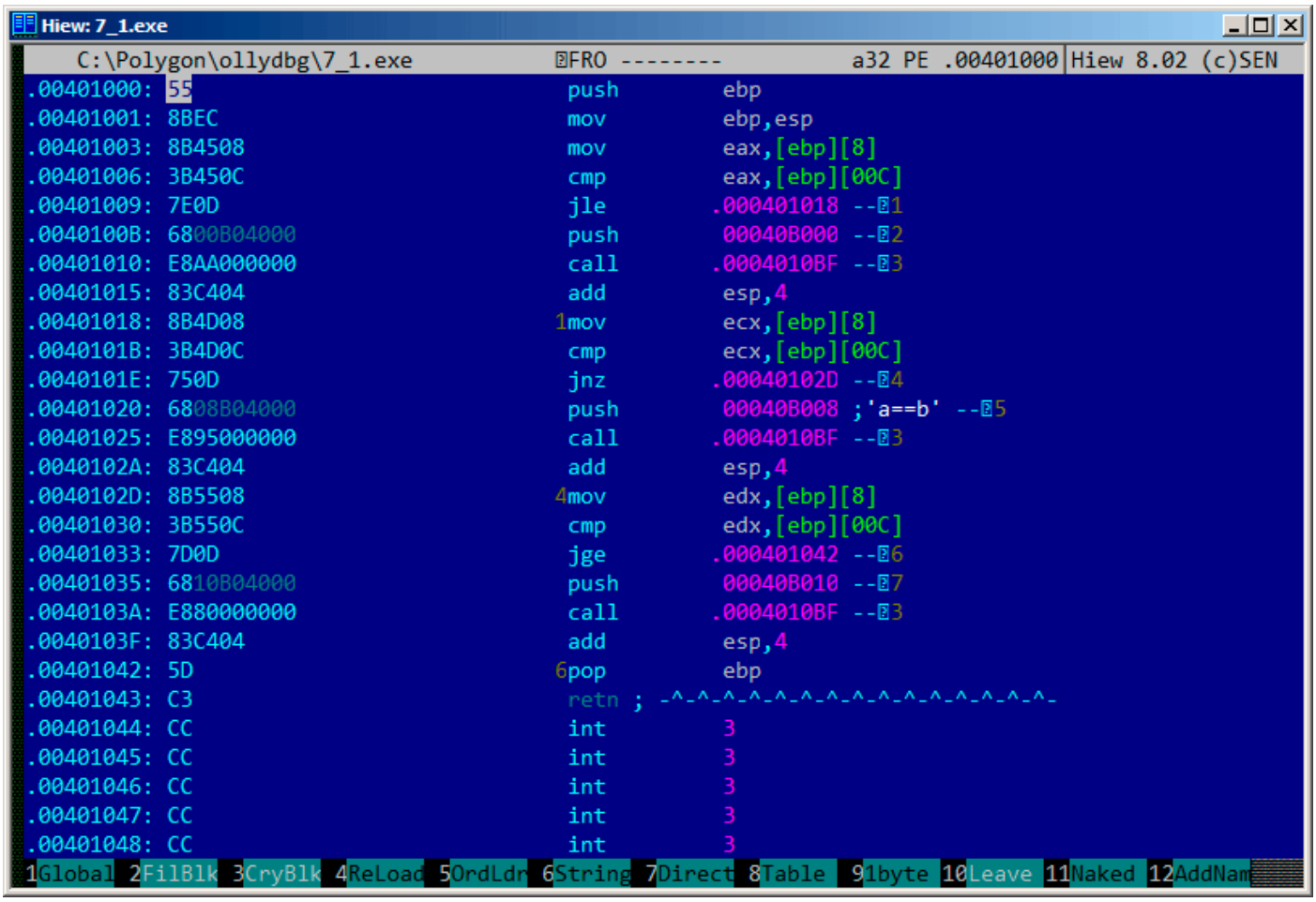


Fig. 1.40: Hiew: fonction `f_unsigned()`

Essentiellement, nous devons accomplir ces trois choses:

- forcer le premier saut à toujours être effectué;
- forcer le second saut à n'être jamais effectué;
- forcer le troisième saut à toujours être effectué.

Nous devons donc diriger le déroulement du code pour toujours effectuer le second `printf()`, et afficher « a==b ».

Trois instructions (ou octets) doivent être modifiées:

- Le premier saut devient un JMP, mais l'offset reste le même.
- Le second saut peut être parfois suivi, mais dans chaque cas il sautera à l'instruction suivante, car nous avons mis l'offset à 0.

  Dans cette instruction, l'offset est ajouté à l'adresse de l'instruction suivante. Donc si l'offset est 0, le saut va transférer l'exécution à l'instruction suivante.
- Le troisième saut est remplacé par JMP comme nous l'avons fait pour le premier, il sera donc toujours effectué.

Voici le code modifié:

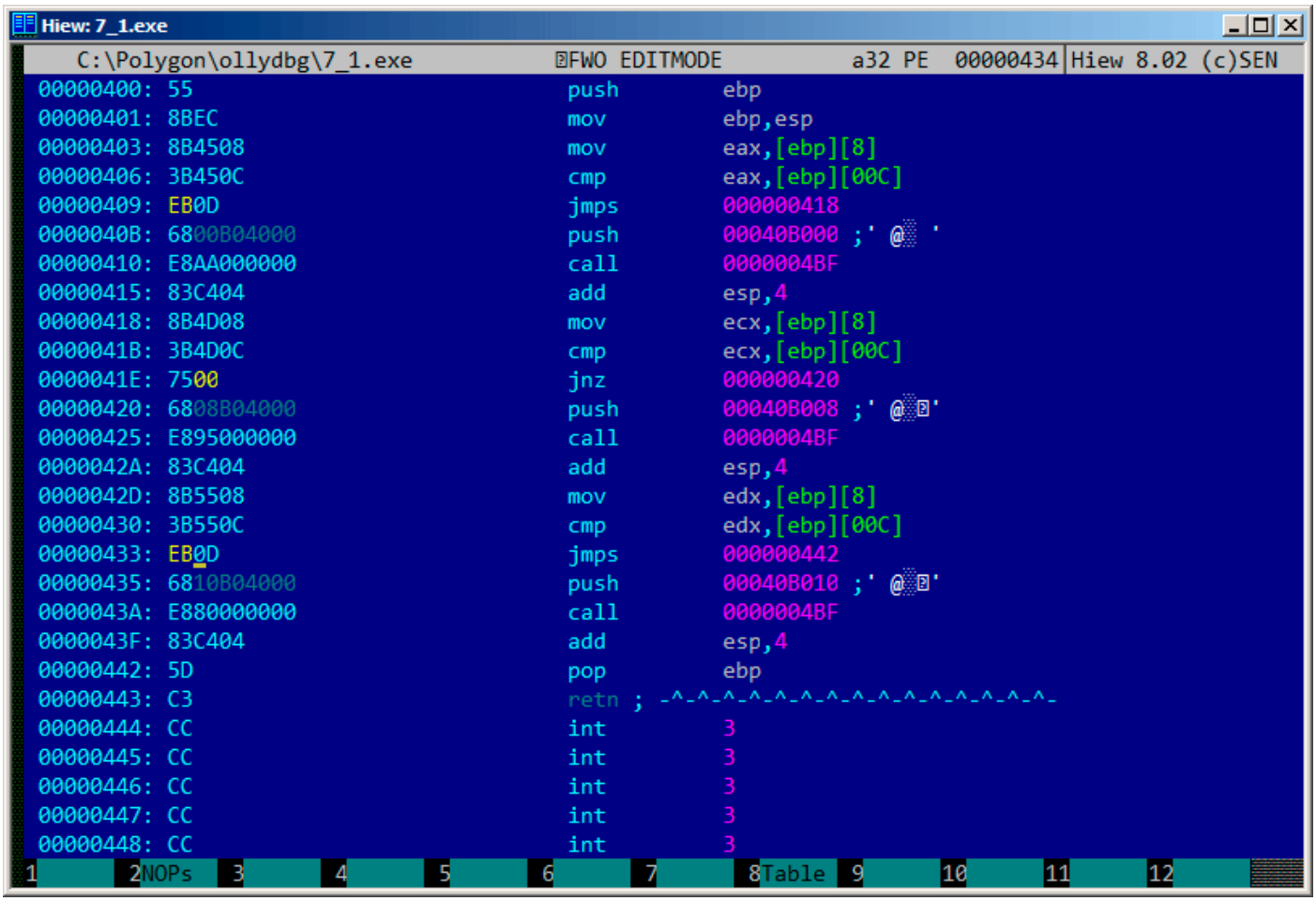


Fig. 1.41: Hiew: modifions la fonction `f_unsigned()`

Si nous oublions de modifier une de ces sauts conditionnels, plusieurs appels à `printf()` pourraient être faits, alors que nous voulons qu'un seul soit exécuté.

GCC sans optimisation

GCC 4.4.1 sans optimisation produit presque le même code, mais avec `puts()` (1.5.4 on page 22) à la place de `printf()`.

GCC avec optimisation

Le lecteur attentif pourrait demander pourquoi exécuter CMP plusieurs fois, si les flags ont les mêmes valeurs après l'exécution ?

Peut-être que l'optimiseur de de MSVC ne peut pas faire cela, mais celui de GCC 4.8.1 peut aller plus loin:

Listing 1.107: GCC 4.8.1 f_signed()

```
f_signed :
        mov     eax, DWORD PTR [esp+8]
        cmp     DWORD PTR [esp+4], eax
        jg      .L6
        je      .L7
        jge     .L1
        mov     DWORD PTR [esp+4], OFFSET FLAT :.LC2 ; "a<b"
        jmp     puts
.L6 :
        mov     DWORD PTR [esp+4], OFFSET FLAT :.LC0 ; "a>b"
```

```
        jmp     puts
.L1 :
        rep ret
.L7 :
        mov     DWORD PTR [esp+4], OFFSET FLAT :.LC1 ; "a==b"
        jmp     puts
```

Nous voyons ici `JMP puts` au lieu de `CALL puts / RETN`.

Ce genre de truc sera expliqué plus loin: 1.15.1 on page 154.

Ce genre de code x86 est plutôt rare. Il semble que MSVC 2012 ne puisse pas générer un tel code. D'un autre côté, les programmeurs en langage d'assemblage sont parfaitement conscients du fait que les instructions `Jcc` peuvent être empilées.

Donc si vous voyez ce genre d'empilement, il est très probable que le code a été écrit à la main.

La fonction `f_unsigned()` n'est pas si esthétiquement courte:

Listing 1.108: GCC 4.8.1 f_unsigned()

```
f_unsigned :
        push    esi
        push    ebx
        sub     esp, 20
        mov     esi, DWORD PTR [esp+32]
        mov     ebx, DWORD PTR [esp+36]
        cmp     esi, ebx
        ja      .L13
        cmp     esi, ebx   ; cette instruction peut être supprimée
        je      .L14
.L10 :
        jb      .L15
        add     esp, 20
        pop     ebx
        pop     esi
        ret
.L15 :
        mov     DWORD PTR [esp+32], OFFSET FLAT :.LC2 ; "a<b"
        add     esp, 20
        pop     ebx
        pop     esi
        jmp     puts
.L13 :
        mov     DWORD PTR [esp], OFFSET FLAT :.LC0 ; "a>b"
        call    puts
        cmp     esi, ebx
        jne     .L10
.L14 :
        mov     DWORD PTR [esp+32], OFFSET FLAT :.LC1 ; "a==b"
        add     esp, 20
        pop     ebx
        pop     esi
        jmp     puts
```

Néanmoins, il y a deux instructions `CMP` au lieu de trois.

Donc les algorithmes d'optimisation de GCC 4.8.1 ne sont probablement pas encore parfaits.

ARM

ARM 32-bit

avec optimisation Keil 6/2013 (Mode ARM)

Listing 1.109: avec optimisation Keil 6/2013 (Mode ARM)

```
.text :000000B8                     EXPORT f_signed
.text :000000B8             f_signed                ; CODE XREF : main+C
.text :000000B8 70 40 2D E9         STMFD   SP!, {R4-R6,LR}
.text :000000BC 01 40 A0 E1         MOV     R4, R1
.text :000000C0 04 00 50 E1         CMP     R0, R4
.text :000000C4 00 50 A0 E1         MOV     R5, R0
.text :000000C8 1A 0E 8F C2         ADRGT   R0, aAB         ; "a>b\n"
.text :000000CC A1 18 00 CB         BLGT    __2printf
.text :000000D0 04 00 55 E1         CMP     R5, R4
.text :000000D4 67 0F 8F 02         ADREQ   R0, aAB_0       ; "a==b\n"
.text :000000D8 9E 18 00 0B         BLEQ    __2printf
.text :000000DC 04 00 55 E1         CMP     R5, R4
.text :000000E0 70 80 BD A8         LDMGEFD SP!, {R4-R6,PC}
.text :000000E4 70 40 BD E8         LDMFD   SP!, {R4-R6,LR}
.text :000000E8 19 0E 8F E2         ADR     R0, aAB_1       ; "a<b\n"
.text :000000EC 99 18 00 EA         B       __2printf
.text :000000EC             ; End of function f_signed
```

Beaucoup d'instructions en mode ARM ne peuvent être exécutées que lorsque certains flags sont mis. E.g, ceci est souvent utilisé lorsque l'on compare les nombres

Par exemple, l'instruction ADD est en fait appelée ADDAL en interne, où AL signifie *Always*, i.e., toujours exécuter. Les prédicats sont encodés dans les 4 bits du haut des instructions ARM 32-bit. (*condition field*). L'instruction de saut inconditionnel B est en fait conditionnelle et encodée comme toutes les autres instructions de saut conditionnel, mais a AL dans le *champ de condition*, et *s'exécute toujours* (ALways), ignorants les flags.

L'instruction ADRGT fonctionne comme ADR mais ne s'exécute que dans le cas où l'instruction CMP précédente a trouvé un des nombres plus grand que l'autre, en comparant les deux (*Greater Than*).

L'instruction BLGT se comporte exactement comme BL et n'est effectuée que si le résultat de la comparaison était *Greater Than* (plus grand). ADRGT écrit un pointeur sur la chaîne a>b\n dans R0 et BLGT appelle `printf()`. Donc, les instructions suffixées par -GT ne sont exécutées que si la valeur dans R0 (qui est a) est plus grande que la valeur dans R4 (qui est b).

En avançant, nous voyons les instructions ADREQ et BLEQ. Elles se comportent comme ADR et BL, mais ne sont exécutées que si les opérandes étaient égaux lors de la dernière comparaison. Un autre CMP se trouve avant elles (car l'exécution de `printf()` pourrait avoir modifiée les flags).

Ensuite nous voyons LDMGEFD, cette instruction fonctionne comme LDMFD[92], mais n'est exécutée que si l'une des valeurs est supérieure ou égale à l'autre (*Greater or Equal*).
L'instruction LDMGEFD SP!, {R4-R6,PC} se comporte comme une fonction épilogue, mais elle ne sera exécutée que si $a >= b$, et seulement lorsque l'exécution de la fonction se terminera.

Mais si cette condition n'est pas satisfaite, i.e., $a < b$, alors le flux d'exécution continue à l'instruction suivante, « LDMFD SP!, {R4-R6,LR} », qui est aussi un épilogue de la fonction. Cette instruction ne restaure pas seulement l'état des registres R4-R6, mais aussi LR au lieu de PC, donc il ne retourne pas de la fonction. Les deux dernières instructions appellent `printf()` avec la chaîne «a<b\n» comme unique argument. Nous avons déjà examiné un saut inconditionnel à la fonction `printf()` au lieu d'un appel avec retour dans «printf() avec plusieurs arguments» section (1.8.2 on page 54).

f_unsigned est très similaire, à part les instructions ADRHI, BLHI, et LDMCSFD utilisées ici, ces prédicats (*HI = Unsigned higher, CS = Carry Set (greater than or equal)*) sont analogues à ceux examinés avant, mais pour des valeurs non signées.

Il n'y a pas grand chose de nouveau pour nous dans la fonction `main()` :

Listing 1.110: `main()`

```
.text :00000128                     EXPORT main
.text :00000128             main
.text :00000128 10 40 2D E9         STMFD   SP!, {R4,LR}
.text :0000012C 02 10 A0 E3         MOV     R1, #2
.text :00000130 01 00 A0 E3         MOV     R0, #1
.text :00000134 DF FF FF EB         BL      f_signed
.text :00000138 02 10 A0 E3         MOV     R1, #2
.text :0000013C 01 00 A0 E3         MOV     R0, #1
.text :00000140 EA FF FF EB         BL      f_unsigned
```

[92] LDMFD

```
.text :00000144 00 00 A0 E3      MOV     R0, #0
.text :00000148 10 80 BD E8      LDMFD   SP!, {R4,PC}
.text :00000148              ; End of function main
```

C'est ainsi que vous pouvez vous débarrasser des sauts conditionnels en mode ARM.

Pourquoi est-ce que c'est si utile? Lire ici: 2.10.1 on page 470.

Il n'y a pas de telle caractéristique en x86, exceptée l'instruction CMOVcc, qui est comme un MOV, mais effectuée seulement lorsque certains flags sont mis, en général mis par CMP.

avec optimisation Keil 6/2013 (Mode Thumb)

Listing 1.111: avec optimisation Keil 6/2013 (Mode Thumb)

```
.text :00000072              f_signed ; CODE XREF : main+6
.text :00000072 70 B5          PUSH    {R4-R6,LR}
.text :00000074 0C 00          MOVS    R4, R1
.text :00000076 05 00          MOVS    R5, R0
.text :00000078 A0 42          CMP     R0, R4
.text :0000007A 02 DD          BLE     loc_82
.text :0000007C A4 A0          ADR     R0, aAB         ; "a>b\n"
.text :0000007E 06 F0 B7 F8    BL      __2printf
.text :00000082
.text :00000082              loc_82 ; CODE XREF : f_signed+8
.text :00000082 A5 42          CMP     R5, R4
.text :00000084 02 D1          BNE     loc_8C
.text :00000086 A4 A0          ADR     R0, aAB_0       ; "a==b\n"
.text :00000088 06 F0 B2 F8    BL      __2printf
.text :0000008C
.text :0000008C              loc_8C ; CODE XREF : f_signed+12
.text :0000008C A5 42          CMP     R5, R4
.text :0000008E 02 DA          BGE     locret_96
.text :00000090 A3 A0          ADR     R0, aAB_1       ; "a<b\n"
.text :00000092 06 F0 AD F8    BL      __2printf
.text :00000096
.text :00000096              locret_96 ; CODE XREF : f_signed+1C
.text :00000096 70 BD          POP     {R4-R6,PC}
.text :00000096              ; End of function f_signed
```

En mode Thumb, seules les instructions B peuvent être complétées par un *condition codes*, (code de condition) donc le code Thumb paraît plus ordinaire.

BLE est un saut conditionnel normal *Less than or Equal* (inférieur ou égal), BNE—*Not Equal* (non égal), BGE—*Greater than or Equal* (supérieur ou égal).

f_unsigned est similaire, seules d'autres instructions sont utilisées pour travailler avec des valeurs non-signées: BLS (*Unsigned lower or same* non signée, inférieur ou égal) et BCS (*Carry Set (Greater than or equal)* supérieur ou égal).

ARM64: GCC (Linaro) 4.9 avec optimisation

Listing 1.112: f_signed()

```
f_signed :
; W0=a, W1=b
        cmp     w0, w1
        bgt     .L19    ; Branch if Greater Than
                  ; branchement is supérieur (a>b)
        beq     .L20    ; Branch if Equal
            ; branchement si égal (a==b)
        bge     .L15    ; Branch if Greater than or Equal
            ; branchement si supérieur ou égal (a>=b) (impossible ici)
        ; a<b
        adrp    x0, .LC11       ; "a<b"
        add     x0, x0,  :lo12 :.LC11
        b       puts
```

```
.L19 :
        adrp    x0, .LC9        ; "a>b"
        add     x0, x0,  :lo12 :.LC9
        b       puts
.L15 :   ; impossible d'arriver ici
        ret
.L20 :
        adrp    x0, .LC10       ; "a==b"
        add     x0, x0,  :lo12 :.LC10
        b       puts
```

Listing 1.113: f_unsigned()

```
f_unsigned :
        stp     x29, x30, [sp, -48]!
; W0=a, W1=b
        cmp     w0, w1
        add     x29, sp, 0
        str     x19, [sp,16]
        mov     w19, w0
        bhi     .L25    ; Branch if HIgher
                ; branchement si supérieur (a>b)
        cmp     w19, w1
        beq     .L26    ; Branch if Equal
                  ; branchement si égal (a==b)
.L23 :
        bcc     .L27    ; Branch if Carry Clear
                  ; branchement si le flag de retenue est à zéro (si inférieur) (a<b)
; épilogue de la fonction, impossible d'arriver ici
        ldr     x19, [sp,16]
        ldp     x29, x30, [sp], 48
        ret
.L27 :
        ldr     x19, [sp,16]
        adrp    x0, .LC11       ; "a<b"
        ldp     x29, x30, [sp], 48
        add     x0, x0,  :lo12 :.LC11
        b       puts
.L25 :
        adrp    x0, .LC9        ; "a>b"
        str     x1, [x29,40]
        add     x0, x0,  :lo12 :.LC9
        bl      puts
        ldr     x1, [x29,40]
        cmp     w19, w1
        bne     .L23    ; Branch if Not Equal
            ; branchement si non égal
.L26 :
        ldr     x19, [sp,16]
        adrp    x0, .LC10       ; "a==b"
        ldp     x29, x30, [sp], 48
        add     x0, x0,  :lo12 :.LC10
        b       puts
```

Les commentaires ont été ajoutés par l'auteur de ce livre. Ce qui frappe ici, c'est que le compilateur n'est pas au courant que certaines conditions ne sont pas possible du tout, donc il y a du code mort par endroit, qui ne sera jamais exécuté.

Exercice

Essayez d'optimiser manuellement la taille de ces fonctions, en supprimant les instructions redondantes, sans en ajouter de nouvelles.

MIPS

Une des caractéristiques distinctives de MIPS est l'absence de flag. Apparemment, cela a été fait pour simplifier l'analyse des dépendances de données.

Il y a des instructions similaires à SETcc en x86: SLT (« Set on Less Than » : mettre si plus petit que, version signée) et SLTU (version non signée). Ces instructions mettent le registre de destination à 1 si la condition est vraie ou à 0 autrement.

Le registre de destination est ensuite testé avec BEQ (« Branch on Equal » branchement si égal) ou BNE (« Branch on Not Equal » branchement si non égal) et un saut peut survenir. Donc, cette paire d'instructions doit être utilisée en MIPS pour comparer et effectuer un branchement. Essayons avec la version signée de notre fonction:

Listing 1.114: GCC 4.4.5 sans optimisation (IDA)

```
.text :00000000 f_signed :                              # CODE XREF : main+18
.text :00000000
.text :00000000 var_10          = -0x10
.text :00000000 var_8           = -8
.text :00000000 var_4           = -4
.text :00000000 arg_0           =  0
.text :00000000 arg_4           =  4
.text :00000000
.text :00000000                 addiu   $sp, -0x20
.text :00000004                 sw      $ra, 0x20+var_4($sp)
.text :00000008                 sw      $fp, 0x20+var_8($sp)
.text :0000000C                 move    $fp, $sp
.text :00000010                 la      $gp, __gnu_local_gp
.text :00000018                 sw      $gp, 0x20+var_10($sp)
; stocker les valeurs en entrée sur la pile locale:
.text :0000001C                 sw      $a0, 0x20+arg_0($fp)
.text :00000020                 sw      $a1, 0x20+arg_4($fp)
; reload them.
.text :00000024                 lw      $v1, 0x20+arg_0($fp)
.text :00000028                 lw      $v0, 0x20+arg_4($fp)
; $v0=b
; $v1=a
.text :0000002C                 or      $at, $zero ; NOP
; ceci est une pseudo-instructions. en fait, c'est "slt $v0,$v0,$v1" ici.
; donc $v0 sera mis à 1 si $v0<$v1 (b<a) ou à 0 autrement:
.text :00000030                 slt     $v0, $v1
; saut en loc_5c, si la condition n'est pas vraie.
; ceci est une pseudo-instruction. en fait, c'est "beq $v0,$zero,loc_5c" ici :
.text :00000034                 beqz    $v0, loc_5C
; afficher "a>b" et terminer
.text :00000038                 or      $at, $zero ; slot de délai de branchement, NOP
.text :0000003C                 lui     $v0, (unk_230 >> 16) # "a>b"
.text :00000040                 addiu   $a0, $v0, (unk_230 & 0xFFFF) # "a>b"
.text :00000044                 lw      $v0, (puts & 0xFFFF)($gp)
.text :00000048                 or      $at, $zero ; NOP
.text :0000004C                 move    $t9, $v0
.text :00000050                 jalr    $t9
.text :00000054                 or      $at, $zero ; slot de délai de branchement, NOP
.text :00000058                 lw      $gp, 0x20+var_10($fp)
.text :0000005C
.text :0000005C loc_5C :                                # CODE XREF : f_signed+34
.text :0000005C                 lw      $v1, 0x20+arg_0($fp)
.text :00000060                 lw      $v0, 0x20+arg_4($fp)
.text :00000064                 or      $at, $zero ; NOP
; tester si a==b, sauter en loc_90 si ce n'est pas vrai :
.text :00000068                 bne     $v1, $v0, loc_90
.text :0000006C                 or      $at, $zero ; slot de délai de branchement, NOP
; la condition est vraie, donc afficher "a==b" et terminer :
.text :00000070                 lui     $v0, (aAB >> 16)  # "a==b"
.text :00000074                 addiu   $a0, $v0, (aAB & 0xFFFF)  # "a==b"
.text :00000078                 lw      $v0, (puts & 0xFFFF)($gp)
.text :0000007C                 or      $at, $zero ; NOP
.text :00000080                 move    $t9, $v0
.text :00000084                 jalr    $t9
.text :00000088                 or      $at, $zero ; slot de délai de branchement, NOP
```

```
.text :0000008C                 lw      $gp, 0x20+var_10($fp)
.text :00000090
.text :00000090 loc_90 :                                # CODE XREF : f_signed+68
.text :00000090                 lw      $v1, 0x20+arg_0($fp)
.text :00000094                 lw      $v0, 0x20+arg_4($fp)
.text :00000098                 or      $at, $zero ; NOP
; tester si $v1<$v0 (a<b), mettre $v0 à 1 si la condition est vraie:
.text :0000009C                 slt     $v0, $v1, $v0
; si la condition n'est pas vraie (i.e., $v0==0), sauter en loc_c8 :
.text :000000A0                 beqz    $v0, loc_C8
.text :000000A4                 or      $at, $zero ; slot de délai de branchement, NOP
; la condition est vraie, afficher "a<b" et terminer
.text :000000A8                 lui     $v0, (aAB_0 >> 16)  # "a<b"
.text :000000AC                 addiu   $a0, $v0, (aAB_0 & 0xFFFF)  # "a<b"
.text :000000B0                 lw      $v0, (puts & 0xFFFF)($gp)
.text :000000B4                 or      $at, $zero ; NOP
.text :000000B8                 move    $t9, $v0
.text :000000BC                 jalr    $t9
.text :000000C0                 or      $at, $zero ; slot de délai de branchement, NOP
.text :000000C4                 lw      $gp, 0x20+var_10($fp)
.text :000000C8
; toutes les 3 conditions étaient fausses, donc simplement terminer:
.text :000000C8 loc_C8 :                                # CODE XREF : f_signed+A0
.text :000000C8                 move    $sp, $fp
.text :000000CC                 lw      $ra, 0x20+var_4($sp)
.text :000000D0                 lw      $fp, 0x20+var_8($sp)
.text :000000D4                 addiu   $sp, 0x20
.text :000000D8                 jr      $ra
.text :000000DC                 or      $at, $zero ; slot de délai de branchement, NOP
.text :000000DC  # Fin de la fonction f_signed
```

SLT REG0, REG0, REG1 est réduit par IDA à sa forme plus courte:
SLT REG0, REG1.

Nous voyons également ici la pseudo instruction BEQZ (« Branch if Equal to Zero » branchement si égal à zéro),
qui est en fait BEQ REG, $ZERO, LABEL.

La version non signée est la même, mais SLTU (version non signée, d'où le « U » de unsigned) est utilisée au lieu de SLT :

Listing 1.115: GCC 4.4.5 sans optimisation (IDA)

```
.text :000000E0 f_unsigned :                            # CODE XREF : main+28
.text :000000E0
.text :000000E0 var_10          = -0x10
.text :000000E0 var_8           = -8
.text :000000E0 var_4           = -4
.text :000000E0 arg_0           =  0
.text :000000E0 arg_4           =  4
.text :000000E0
.text :000000E0                 addiu   $sp, -0x20
.text :000000E4                 sw      $ra, 0x20+var_4($sp)
.text :000000E8                 sw      $fp, 0x20+var_8($sp)
.text :000000EC                 move    $fp, $sp
.text :000000F0                 la      $gp, __gnu_local_gp
.text :000000F8                 sw      $gp, 0x20+var_10($sp)
.text :000000FC                 sw      $a0, 0x20+arg_0($fp)
.text :00000100                 sw      $a1, 0x20+arg_4($fp)
.text :00000104                 lw      $v1, 0x20+arg_0($fp)
.text :00000108                 lw      $v0, 0x20+arg_4($fp)
.text :0000010C                 or      $at, $zero
.text :00000110                 sltu    $v0, $v1
.text :00000114                 beqz    $v0, loc_13C
.text :00000118                 or      $at, $zero
.text :0000011C                 lui     $v0, (unk_230 >> 16)
.text :00000120                 addiu   $a0, $v0, (unk_230 & 0xFFFF)
.text :00000124                 lw      $v0, (puts & 0xFFFF)($gp)
.text :00000128                 or      $at, $zero
.text :0000012C                 move    $t9, $v0
.text :00000130                 jalr    $t9
```

```
.text :00000134                 or      $at, $zero
.text :00000138                 lw      $gp, 0x20+var_10($fp)
.text :0000013C
.text :0000013C loc_13C :                               # CODE XREF : f_unsigned+34
.text :0000013C                 lw      $v1, 0x20+arg_0($fp)
.text :00000140                 lw      $v0, 0x20+arg_4($fp)
.text :00000144                 or      $at, $zero
.text :00000148                 bne     $v1, $v0, loc_170
.text :0000014C                 or      $at, $zero
.text :00000150                 lui     $v0, (aAB >> 16)  # "a==b"
.text :00000154                 addiu   $a0, $v0, (aAB & 0xFFFF)  # "a==b"
.text :00000158                 lw      $v0, (puts & 0xFFFF)($gp)
.text :0000015C                 or      $at, $zero
.text :00000160                 move    $t9, $v0
.text :00000164                 jalr    $t9
.text :00000168                 or      $at, $zero
.text :0000016C                 lw      $gp, 0x20+var_10($fp)
.text :00000170
.text :00000170 loc_170 :                               # CODE XREF : f_unsigned+68
.text :00000170                 lw      $v1, 0x20+arg_0($fp)
.text :00000174                 lw      $v0, 0x20+arg_4($fp)
.text :00000178                 or      $at, $zero
.text :0000017C                 sltu    $v0, $v1, $v0
.text :00000180                 beqz    $v0, loc_1A8
.text :00000184                 or      $at, $zero
.text :00000188                 lui     $v0, (aAB_0 >> 16)  # "a<b"
.text :0000018C                 addiu   $a0, $v0, (aAB_0 & 0xFFFF)  # "a<b"
.text :00000190                 lw      $v0, (puts & 0xFFFF)($gp)
.text :00000194                 or      $at, $zero
.text :00000198                 move    $t9, $v0
.text :0000019C                 jalr    $t9
.text :000001A0                 or      $at, $zero
.text :000001A4                 lw      $gp, 0x20+var_10($fp)
.text :000001A8
.text :000001A8 loc_1A8 :                               # CODE XREF : f_unsigned+A0
.text :000001A8                 move    $sp, $fp
.text :000001AC                 lw      $ra, 0x20+var_4($sp)
.text :000001B0                 lw      $fp, 0x20+var_8($sp)
.text :000001B4                 addiu   $sp, 0x20
.text :000001B8                 jr      $ra
.text :000001BC                 or      $at, $zero
.text :000001BC  # End of function f_unsigned
```

1.14.2 Calcul de valeur absolue

Une fonction simple:

```
int my_abs (int i)
{
        if (i<0)
                return -i;
        else
                return i;
};
```

MSVC avec optimisation

Ceci est le code généré habituellement:

Listing 1.116: MSVC 2012 x64 avec optimisation

```
i$ = 8
my_abs  PROC
; ECX = valeur en entrée
        test    ecx, ecx
; tester le signe de la valeur en entrée
```

```
; sauter l'instruction NEG si le signe est positif
        jns     SHORT $LN2@my_abs
; inverser la valeur
        neg     ecx
$LN2@my_abs :
; copier le résultat dans EAX:
        mov     eax, ecx
        ret     0
my_abs  ENDP
```

GCC 4.9 génère en gros le même code:

avec optimisation Keil 6/2013 : Mode Thumb

Listing 1.117: avec optimisation Keil 6/2013 : Mode Thumb

```
my_abs PROC
        CMP      r0,#0
; est-ce que la valeur en entrée est égale ou plus grande que zéro?
; si oui, sauter l'instruction RSBS
        BGE      |L0.6|
; soustraire la valeur en entrée de 0:
        RSBS     r0,r0,#0
|L0.6|
        BX       lr
        ENDP
```

Il manque une instruction de négation en ARM, donc le compilateur Keil utilise l'instruction « Reverse Subtract », qui soustrait la valeur du registre de l'opérande.

avec optimisation Keil 6/2013 : Mode ARM

Il est possible d'ajouter un code de condition à certaines instructions en mode ARM, c'est donc ce que fait le compilateur Keil:

Listing 1.118: avec optimisation Keil 6/2013 : Mode ARM

```
my_abs PROC
        CMP      r0,#0
; exécuter l'instruction "Reverse Subtract" seulement si la valeur en entrée
; est plus petite que 0:
        RSBLT    r0,r0,#0
        BX       lr
        ENDP
```

Maintenant, il n'y a plus de saut conditionnel et c'est mieux: 2.10.1 on page 470.

GCC 4.9 sans optimisation (ARM64)

ARM64 possède l'instruction NEG pour effectuer la négation:

Listing 1.119: GCC 4.9 avec optimisation (ARM64)

```
my_abs :
        sub     sp, sp, #16
        str     w0, [sp,12]
        ldr     w0, [sp,12]
; comparer la valeur en entrée avec le contenu du registre WZR
; (qui contient toujours zéro)
        cmp     w0, wzr
        bge     .L2
        ldr     w0, [sp,12]
        neg     w0, w0
        b       .L3
.L2 :
```

```
        ldr     w0, [sp,12]
.L3 :
        add     sp, sp, 16
        ret
```

MIPS

Listing 1.120: GCC 4.4.5 avec optimisation (IDA)

```
my_abs :
; saut si $a0<0:
                bltz    $a0, locret_10
; simplement renvoyer la valeur en entrée ($a0) dans $v0:
                move    $v0, $a0
                jr      $ra
                or      $at, $zero ; slot de délai de branchement, NOP
locret_10 :
; prendre l'opposée de la valeur entrée et la stocker dans $v0:
                jr      $ra
; ceci est une pseudo-instruction. En fait, ceci est "subu $v0,$zero,$a0" ($v0=0-$a0)
                negu    $v0, $a0
```

Nous voyons ici une nouvelle instruction: BLTZ (« Branch if Less Than Zero » branchement si plus petit que zéro).

Il y a aussi la pseudo-instruction NEGU, qui effectue une soustraction à zéro. Le suffixe « U » dans les deux instructions SUBU et NEGU indique qu'aucune exception ne sera levée en cas de débordement de la taille d'un entier.

Version sans branchement?

Vous pouvez aussi avoir une version sans branchement de ce code. Ceci sera revu plus tard: 3.13 on page 522.

1.14.3 Opérateur conditionnel ternaire

L'opérateur conditionnel ternaire en C/C++ a la syntaxe suivante:

```
expression ? expression  : expression
```

Voici un exemple:

```
const char* f (int a)
{
        return a==10 ? "it is ten"  : "it is not ten";
};
```

x86

Les vieux compilateurs et ceux sans optimisation génèrent du code assembleur comme si des instructions `if/else` avaient été utilisées:

Listing 1.121: MSVC 2008 sans optimisation

```
$SG746  DB      'it is ten', 00H
$SG747  DB      'it is not ten', 00H

tv65 = -4 ; ceci sera utilisé comme variable temporaire
_a$ = 8
_f      PROC
        push    ebp
        mov     ebp, esp
        push    ecx
```

```
; comparer la valeur en entrée avec 10
        cmp     DWORD PTR _a$[ebp], 10
; sauter en $LN3@f si non égal
        jne     SHORT $LN3@f
; stocker le pointeur sur la chaîne dans la variable temporaire:
        mov     DWORD PTR tv65[ebp], OFFSET $SG746 ; 'it is ten'
; sauter à la sortie
        jmp     SHORT $LN4@f
$LN3@f :
; stocker le pointeur sur la chaîne dans la variable temporaire:
        mov     DWORD PTR tv65[ebp], OFFSET $SG747 ; 'it is not ten'
$LN4@f :
; ceci est la sortie. copier le pointeur sur la chaîne depuis la variable temporaire dans EAX.
        mov     eax, DWORD PTR tv65[ebp]
        mov     esp, ebp
        pop     ebp
        ret     0
_f      ENDP
```

Listing 1.122: MSVC 2008 avec optimisation

```
$SG792  DB      'it is ten', 00H
$SG793  DB      'it is not ten', 00H

_a$ = 8 ; taille = 4
_f      PROC
; comparer la valeur en entrée avec 10
        cmp     DWORD PTR _a$[esp-4], 10
        mov     eax, OFFSET $SG792 ; 'it is ten'
; sauter en $LN4@f si égal
        je      SHORT $LN4@f
        mov     eax, OFFSET $SG793 ; 'it is not ten'
$LN4@f :
        ret     0
_f      ENDP
```

Les nouveaux compilateurs sont plus concis:

Listing 1.123: MSVC 2012 x64 avec optimisation

```
$SG1355 DB      'it is ten', 00H
$SG1356 DB      'it is not ten', 00H

a$ = 8
f       PROC
; charger les pointeurs sur les deux chaînes
        lea     rdx, OFFSET FLAT :$SG1355 ; 'it is ten'
        lea     rax, OFFSET FLAT :$SG1356 ; 'it is not ten'
; comparer la valeur en entrée avec 10
        cmp     ecx, 10
; si égal, copier la valeur dans RDX ("it is ten")
; si non, ne rien faire. le pointeur sur la chaîne "it is not ten" est encore dans RAX à ce
    stade.
        cmove   rax, rdx
        ret     0
f       ENDP
```

GCC 4.8 avec optimisation pour x86 utilise également l'instruction `CMOVcc`, tandis que GCC 4.8 sans optimisation utilise des sauts conditionnels.

ARM

Keil avec optimisation pour le mode ARM utilise les instructions conditionnelles `ADRcc` :

Listing 1.124: avec optimisation Keil 6/2013 (Mode ARM)

```
f PROC
; comparer la valeur en entrée avec 10
        CMP      r0,#0xa
```

```
; si le résultat de la comparaison est égal (EQual), copier le pointeur sur la chaîne; "it is
    ten" dans R0
        ADREQ    r0,|L0.16| ; "it is ten"
; si le résultat de la comparaison est non égal (Not EQual), copier le pointeur sur la chaîne;
    "it is not ten" dans R0
        ADRNE    r0,|L0.28| ; "it is not ten"
        BX       lr
        ENDP

|L0.16|
        DCB      "it is ten",0
|L0.28|
        DCB      "it is not ten",0
```

Sans intervention manuelle, les deux instructions ADREQ et ADRNE ne peuvent être exécutées lors de la même exécution.

Keil avec optimisation pour le mode Thumb à besoin d'utiliser des instructions de saut conditionnel, puisqu'il n'y a pas d'instruction qui supporte le flag conditionnel.

Listing 1.125: avec optimisation Keil 6/2013 (Mode Thumb)

```
f PROC
; comparer la valeur entrée avec 10
        CMP      r0,#0xa
; sauter en |L0.8| si égal (EQual )
        BEQ      |L0.8|
        ADR      r0,|L0.12| ; "it is not ten"
        BX       lr
|L0.8|
        ADR      r0,|L0.28| ; "it is ten"
        BX       lr
        ENDP

|L0.12|
        DCB      "it is not ten",0
|L0.28|
        DCB      "it is ten",0
```

ARM64

GCC (Linaro) 4.9 avec optimisation pour ARM64 utilise aussi des sauts conditionnels:

Listing 1.126: GCC (Linaro) 4.9 avec optimisation

```
f :
        cmp     x0, 10
        beq     .L3             ; branchement si égal
        adrp    x0, .LC1        ; "it is ten"
        add     x0, x0,  :lo12 :.LC1
        ret
.L3 :
        adrp    x0, .LC0        ; "it is not ten"
        add     x0, x0,  :lo12 :.LC0
        ret
.LC0 :
        .string "it is ten"
.LC1 :
        .string "it is not ten"
```

C'est parce qu'ARM64 n'a pas d'instruction de chargement simple avec le flag conditionnel comme ADRcc en ARM 32-bit ou CMOVcc en x86.

Il a toutefois l'instruction « Conditional SELect » (CSEL)[*ARM Architecture Reference Manual, ARMv8, for ARMv8-A architecture profile*, (2013)p390, C5.5], mais GCC 4.9 ne semble pas assez malin pour l'utiliser dans un tel morceau de code.

MIPS

Malheureusement, GCC 4.4.5 pour MIPS n'est pas très malin non plus:

Listing 1.127: GCC 4.4.5 avec optimisation (résultat en sortie de l'assembleur)

```
$LC0 :
        .ascii  "it is not ten\000"
$LC1 :
        .ascii  "it is ten\000"
f :
        li      $2,10                   # 0xa
; comparer $a0 et 10, sauter si égal:
        beq     $4,$2,$L2
        nop ; slot de délai de branchement

; charger l'adresse de la chaîne "it is not ten" dans $v0 et sortir:
        lui     $2,%hi($LC0)
        j       $31
        addiu   $2,$2,%lo($LC0)

$L2 :
; charger l'adresse de la chaîne "it is ten" dans $v0 et sortir:
        lui     $2,%hi($LC1)
        j       $31
        addiu   $2,$2,%lo($LC1)
```

Récrivons-le à l'aide d'`unif/else`

```
const char* f (int a)
{
        if (a==10)
                return "it is ten";
        else
                return "it is not ten";
};
```

Curieusement, GCC 4.8 avec l'optimisation a pû utiliser `CMOVcc` dans ce cas:

Listing 1.128: GCC 4.8 avec optimisation

```
.LC0 :
        .string "it is ten"
.LC1 :
        .string "it is not ten"
f :
.LFB0 :
; comparer la valeur en entrée avec 10
        cmp     DWORD PTR [esp+4], 10
        mov     edx, OFFSET FLAT :.LC1 ; "it is not ten"
        mov     eax, OFFSET FLAT :.LC0 ; "it is ten"
; si le résultat de la comparaison est Not Equal, copier la valeur de EDX dans EAX
; sinon, ne rien faire
        cmovne  eax, edx
        ret
```

Keil avec optimisation génère un code identique à liste.1.124.

Mais MSVC 2012 avec optimisation n'est pas (encore) si bon.

Conclusion

Pourquoi est-ce que les compilateurs qui optimisent essayent de se débarrasser des sauts conditionnels? Voir à ce propos: 2.10.1 on page 470.

1.14.4 Trouver les valeurs minimale et maximale

32-bit

```
int my_max(int a, int b)
{
        if (a>b)
                return a;
        else
                return b;
};

int my_min(int a, int b)
{
        if (a<b)
                return a;
        else
                return b;
};
```

Listing 1.129: MSVC 2013 sans optimisation

```
_a$ = 8
_b$ = 12
_my_min PROC
        push    ebp
        mov     ebp, esp
        mov     eax, DWORD PTR _a$[ebp]
; comparer A et B :
        cmp     eax, DWORD PTR _b$[ebp]
; sauter si A est supérieur ou égal à B:
        jge     SHORT $LN2@my_min
; recharger A dans EAX si autrement et sauter à la sortie
        mov     eax, DWORD PTR _a$[ebp]
        jmp     SHORT $LN3@my_min
        jmp     SHORT $LN3@my_min ; ce JMP est redondant
$LN2@my_min :
; renvoyer B
        mov     eax, DWORD PTR _b$[ebp]
$LN3@my_min :
        pop     ebp
        ret     0
_my_min ENDP

_a$ = 8
_b$ = 12
_my_max PROC
        push    ebp
        mov     ebp, esp
        mov     eax, DWORD PTR _a$[ebp]
; comparer A et B :
        cmp     eax, DWORD PTR _b$[ebp]
; sauter si A est inférieur ou égal à B:
        jle     SHORT $LN2@my_max
; recharger A dans EAX si autrement et sauter à la sortie
        mov     eax, DWORD PTR _a$[ebp]
        jmp     SHORT $LN3@my_max
        jmp     SHORT $LN3@my_max ; ce JMP est redondant
$LN2@my_max :
; renvoyer B
        mov     eax, DWORD PTR _b$[ebp]
$LN3@my_max :
        pop     ebp
        ret     0
_my_max ENDP
```

Ces deux fonctions ne diffèrent que de l'instruction de saut conditionnel: JGE (« Jump if Greater or Equal » saut si supérieur ou égal) est utilisée dans la première et JLE (« Jump if Less or Equal » saut si inférieur ou égal) dans la seconde.

Il y a une instruction JMP en trop dans chaque fonction, que MSVC a probablement mise par erreur.

Sans branchement

Le mode Thumb d'ARM nous rappelle le code x86:

Listing 1.130: avec optimisation Keil 6/2013 (Mode Thumb)

```
my_max PROC
; R0=A
; R1=B
; comparer A et B :
        CMP      r0,r1
; branchement si A est supérieur à B:
        BGT      |L0.6|
; autrement (A<=B) renvoyer R1 (B) :
        MOVS     r0,r1
|L0.6|
; retourner
        BX       lr
        ENDP

my_min PROC
; R0=A
; R1=B
; comparer A et B :
        CMP      r0,r1
; branchement si A est inférieur à B:
        BLT      |L0.14|
; autrement (A>=B) renvoyer R1 (B) :
        MOVS     r0,r1
|L0.14|
; retourner
        BX       lr
        ENDP
```

Les fonctions diffèrent au niveau de l'instruction de branchement: BGT et BLT. Il est possible d'utiliser le suffixe conditionnel en mode ARM, donc le code est plus court.

MOVcc n'est exécutée que si la condition est remplie:

Listing 1.131: avec optimisation Keil 6/2013 (Mode ARM)

```
my_max PROC
; R0=A
; R1=B
; comparer A et B :
        CMP      r0,r1
; renvoyer B au lieu de A en copiant B dans R0
; cette instruction ne s'exécutera que si A<=B (en effet, LE Less or Equal, inférieur ou égal)
; si l'instruction n'est pas exécutée (dans le cas où A>B), A est toujours dans le registre R0
        MOVLE    r0,r1
        BX       lr
        ENDP

my_min PROC
; R0=A
; R1=B
; comparer A et B :
        CMP      r0,r1
; renvoyer B au lieu de A en copiant B dans R0
; cette instruction ne s'exécutera que si A>=B (GE Greater or Equal, supérieur ou égal)
; si l'instruction n'est pas exécutée (dans le cas où A<B), A est toujours dans le registre R0
        MOVGE    r0,r1
        BX       lr
        ENDP
```

GCC 4.8.1 avec optimisation et MSVC 2013 avec optimisation peuvent utiliser l'instruction CMOVcc, qui est analogue à MOVcc en ARM:

Listing 1.132: MSVC 2013 avec optimisation

```
my_max :
        mov     edx, DWORD PTR [esp+4]
        mov     eax, DWORD PTR [esp+8]
; EDX=A
; EAX=B
; comparer A et B :
        cmp     edx, eax
; si A>=B, charger la valeur A dans EAX
; l'instruction ne fait rien autrement (si A<B)
        cmovge  eax, edx
        ret

my_min :
        mov     edx, DWORD PTR [esp+4]
        mov     eax, DWORD PTR [esp+8]
; EDX=A
; EAX=B
; comparer A et B :
        cmp     edx, eax
; si A<=B, charger la valeur A dans EAX
; l'instruction ne fait rien autrement (si A>B)
        cmovle  eax, edx
        ret
```

64-bit

```
#include <stdint.h>

int64_t my_max(int64_t a, int64_t b)
{
        if (a>b)
                return a;
        else
                return b;
};

int64_t my_min(int64_t a, int64_t b)
{
        if (a<b)
                return a;
        else
                return b;
};
```

Il y a beaucoup de code inutile qui embrouille, mais il est compréhensible:

Listing 1.133: GCC 4.9.1 ARM64 sans optimisation

```
my_max :
        sub     sp, sp, #16
        str     x0, [sp,8]
        str     x1, [sp]
        ldr     x1, [sp,8]
        ldr     x0, [sp]
        cmp     x1, x0
        ble     .L2
        ldr     x0, [sp,8]
        b       .L3
.L2 :
        ldr     x0, [sp]
.L3 :
        add     sp, sp, 16
        ret

my_min :
        sub     sp, sp, #16
        str     x0, [sp,8]
```

```
        str     x1, [sp]
        ldr     x1, [sp,8]
        ldr     x0, [sp]
        cmp     x1, x0
        bge     .L5
        ldr     x0, [sp,8]
        b       .L6
.L5 :
        ldr     x0, [sp]
.L6 :
        add     sp, sp, 16
        ret
```

Sans branchement

Il n'y a pas besoin de lire les arguments dans la pile, puisqu'ils sont déjà dans les registres:

Listing 1.134: GCC 4.9.1 x64 avec optimisation

```
my_max :
; RDI=A
; RSI=B
; comparer A et B :
        cmp     rdi, rsi
; préparer B pour le renvoyer dans RAX:
        mov     rax, rsi
; si A>=B, mettre A (RDI) dans RAX pour le renvoyer.
; cette instruction ne fait rien autrement (si A<B)
        cmovge  rax, rdi
        ret

my_min :
; RDI=A
; RSI=B
; comparer A et B :
        cmp     rdi, rsi
; préparer B pour le renvoyer dans RAX:
        mov     rax, rsi
; si A<=B, mettre A (RDI) dans RAX pour le renvoyer.
; cette instruction ne fait rien autrement (si A>B)
        cmovle  rax, rdi
        ret
```

MSVC 2013 fait presque la même chose.

ARM64 possède l'instruction CSEL, qui fonctionne comme `MOVcc` en ARM ou `CMOVcc` en x86, seul le nom diffère: « Conditional SELect ».

Listing 1.135: GCC 4.9.1 ARM64 avec optimisation

```
my_max :
; X0=A
; X1=B
; comparer A et B :
        cmp     x0, x1
; copier X0 (A) dans X0 si X0>=X1 ou A>=B (Greater or Equal, supérieur ou égal)
; copier X1 (B) dans X0 si A<B
        csel    x0, x0, x1, ge
        ret

my_min :
; X0=A
; X1=B
; comparer A et B :
        cmp     x0, x1
; copier X0 (A) dans X0 si X0<=X1 ou A<=B (Less or Equal, inférieur ou égal)
; copier X1 (B) dans X0 si A>B
        csel    x0, x0, x1, le
        ret
```

MIPS

Malheureusement, GCC 4.4.5 pour MIPS n'est pas si performant:

Listing 1.136: GCC 4.4.5 avec optimisation (IDA)

```
my_max :
; mettre $v1 à 1 si $a1<$a0, ou l'effacer autrement (si $a1>$a0):
                slt     $v1, $a1, $a0
; sauter, si $v1 est 0 (ou $a1>$a0) :
                beqz    $v1, locret_10
; ceci est le slot de délai de branchement
; préparer $a1 dans $v0 si le branchement est pris:
                move    $v0, $a1
; le branchment n'est pas pris, préparer $a0 dans $v0:
                move    $v0, $a0

locret_10 :
                jr      $ra
                or      $at, $zero ; slot de délai de branchement, NOP

; la fonction min() est la même, mais les opérandes dans l'instruction SLT sont échangés:
my_min :
                slt     $v1, $a0, $a1
                beqz    $v1, locret_28
                move    $v0, $a1
                move    $v0, $a0

locret_28 :
                jr      $ra
                or      $at, $zero ; slot de délai de branchement, NOP
```

N'oubliez pas le slot de délai de branchement (*branch delay slots*): le premier MOVE est exécuté *avant* BEQZ, le second MOVE n'est exécuté que si la branche n'a pas été prise.

1.14.5 Conclusion

x86

Voici le squelette générique d'un saut conditionnel:

Listing 1.137: x86

```
CMP registre, registre/valeur
Jcc true ; cc=condition code, code de condition
false :
... le code qui sera exécuté si le résultat de la comparaison est faux (false) ...
JMP exit
true :
... le code qui sera exécuté si le résultat de la comparaison est vrai (true) ...
exit :
```

ARM

Listing 1.138: ARM

```
CMP registre, registre/valeur
Bcc true ; cc=condition code
false :
... le code qui sera exécuté si le résultat de la comparaison est faux (false) ...
JMP exit
true :
... le code qui sera exécuté si le résultat de la comparaison est vrai (true) ...
exit :
```

MIPS

Listing 1.139: Check si zéro (Branch if EQual Zero)

```
BEQZ REG, label
...
```

Listing 1.140: Check si plus petit que zéro (Branch if Less Than Zero) en utilisant une pseudo instruction

```
BLTZ REG, label
...
```

Listing 1.141: Check si les valeurs sont égales (Branch if EQual)

```
BEQ REG1, REG2, label
...
```

Listing 1.142: Check si les valeurs ne sont pas égales (Branch if Not Equal)

```
BNE REG1, REG2, label
...
```

Listing 1.143: Check REG2 plus petit que REG3 (signé)

```
SLT REG1, REG2, REG3
BEQ REG1, label
...
```

Listing 1.144: Check REG2 plus petit que REG3 (non signé)

```
SLTU REG1, REG2, REG3
BEQ REG1, label
...
```

Sans branchement

Si le corps d'instruction conditionnelle est très petit, l'instruction de déplacement conditionnel peut être utilisée: `MOVcc` en ARM (en mode ARM), `CSEL` en ARM64, `CMOVcc` en x86.

ARM

Il est possible d'utiliser les suffixes conditionnels pour certaines instructions ARM:

Listing 1.145: ARM (Mode ARM)

```
CMP registre, registre/valeur
instr1_cc ; cette instruction sera exécutée si le code conditionnel est vrai (true)
instr2_cc ; cette autre instruction sera exécutée si cet autre code conditionnel est
    vrai (true)
... etc.
```

Bien sûr, il n'y a pas de limite au nombre d'instructions avec un suffixe de code conditionnel, tant que les flags du CPU ne sont pas modifiés par l'une d'entre elles.

Le mode Thumb possède l'instruction `IT`, permettant d'ajouter le suffixe conditionnel pour les quatre instructions suivantes. Lire à ce propos: 1.19.7 on page 264.

Listing 1.146: ARM (Mode Thumb)

```
CMP registre, registre/valeur
ITEEE EQ ; met ces suffixes : if-then-else-else-else
instr1   ; instruction exécutée si la condition est vraie
instr2   ; instruction exécutée si la condition est fausse
instr3   ; instruction exécutée si la condition est fausse
instr4   ; instruction exécutée si la condition est fausse
```

1.14.6 Exercice

(ARM64) Essayez de récrire le code pour liste.1.126 en supprimant toutes les instructions de saut conditionnel et en utilisant l'instruction CSEL.

1.15 switch()/case/default

1.15.1 Petit nombre de cas

```
#include <stdio.h>

void f (int a)
{
    switch (a)
    {
    case 0: printf ("zero\n"); break;
    case 1: printf ("one\n"); break;
    case 2: printf ("two\n"); break;
    default : printf ("something unknown\n"); break;
    };
};

int main()
{
    f (2); // test
};
```

x86

MSVC sans optimisation

Résultat (MSVC 2010):

Listing 1.147: MSVC 2010

```
tv64 = -4 ; size = 4
_a$ = 8   ; size = 4
_f    PROC
    push   ebp
    mov    ebp, esp
    push   ecx
    mov    eax, DWORD PTR _a$[ebp]
    mov    DWORD PTR tv64[ebp], eax
    cmp    DWORD PTR tv64[ebp], 0
    je     SHORT $LN4@f
    cmp    DWORD PTR tv64[ebp], 1
    je     SHORT $LN3@f
    cmp    DWORD PTR tv64[ebp], 2
    je     SHORT $LN2@f
    jmp    SHORT $LN1@f
$LN4@f :
    push   OFFSET $SG739 ; 'zero', 0aH, 00H
    call   _printf
    add    esp, 4
    jmp    SHORT $LN7@f
$LN3@f :
    push   OFFSET $SG741 ; 'one', 0aH, 00H
    call   _printf
    add    esp, 4
    jmp    SHORT $LN7@f
$LN2@f :
    push   OFFSET $SG743 ; 'two', 0aH, 00H
    call   _printf
    add    esp, 4
```

```
    jmp    SHORT $LN7@f
$LN1@f :
    push   OFFSET $SG745 ; 'something unknown', 0aH, 00H
    call   _printf
    add    esp, 4
$LN7@f :
    mov    esp, ebp
    pop    ebp
    ret    0
_f    ENDP
```

Notre fonction avec quelques cas dans switch() est en fait analogue à cette construction:

```
void f (int a)
{
    if (a==0)
        printf ("zero\n");
    else if (a==1)
        printf ("one\n");
    else if (a==2)
        printf ("two\n");
    else
        printf ("something unknown\n");
};
```

Si nous utilisons switch() avec quelques cas, il est impossible de savoir si il y avait un vrai switch() dans le code source, ou un ensemble de directives if().

Ceci indique que switch() est comme un sucre syntaxique pour un grand nombre de if() imbriqués.

Il n'y a rien de particulièrement nouveau pour nous dans le code généré, à l'exception que le compilateur déplace la variable d'entrée a dans une variable locale temporaire `tv64` [93].

Si nous compilons ceci avec GCC 4.4.1, nous obtenons presque le même résultat, même avec le niveau d'optimisation le plus élevé (`-03` option).

MSVC avec optimisation

Maintenant compilons dans MSVC avec l'optimisation (`/Ox`): `cl 1.c /Fa1.asm /Ox`

Listing 1.148: MSVC

```
_a$ = 8 ; size = 4
_f    PROC
    mov    eax, DWORD PTR _a$[esp-4]
    sub    eax, 0
    je     SHORT $LN4@f
    sub    eax, 1
    je     SHORT $LN3@f
    sub    eax, 1
    je     SHORT $LN2@f
    mov    DWORD PTR _a$[esp-4], OFFSET $SG791 ; 'something unknown', 0aH, 00H
    jmp    _printf
$LN2@f :
    mov    DWORD PTR _a$[esp-4], OFFSET $SG789 ; 'two', 0aH, 00H
    jmp    _printf
$LN3@f :
    mov    DWORD PTR _a$[esp-4], OFFSET $SG787 ; 'one', 0aH, 00H
    jmp    _printf
$LN4@f :
    mov    DWORD PTR _a$[esp-4], OFFSET $SG785 ; 'zero', 0aH, 00H
    jmp    _printf
_f    ENDP
```

Ici, nous voyons quelques hacks moches.

Premièrement: la valeurs de a est mise dans EAX et 0 en est soustrait. Ça semble absurde, mais cela est fait pour vérifier si la valeur dans EAX est 0. Si oui, le flag ZF est mis (e.g. soustraire de 0 est 0) et le

[93] Les variables locales sur la pile sont préfixées avec `tv`—c'est ainsi que MSVC appelle les variables internes dont il a besoin.

premier saut conditionnel JE (*Jump if Equal* saut si égal ou synonyme JZ —*Jump if Zero* saut si zéro) va être effectué et le déroulement du programme passera au label $LN4@f, où le message 'zero' est affiché. Si le premier saut n'est pas effectué, 1 est soustrait de la valeur d'entrée et si à une étape le résultat est 0, le saut correspondant sera effectué.

Et si aucun saut n'est exécuté, l'exécution passera au printf() avec comme argument la chaîne 'something unknown'.

Deuxièmement: nous voyons quelque chose d'inhabituel pour nous: un pointeur sur une chaîne est mis dans la variable a et ensuite printf() est appelé non pas par CALL, mais par JMP. Il y a une explication simple à cela: l'appelant pousse une valeur sur la pile et appelle notre fonction via CALL. CALL lui même pousse l'adresse de retour (RA) sur la pile et fait un saut inconditionnel à l'adresse de notre fonction. Notre fonction, à tout moment de son exécution (car elle ne contient pas d'instruction qui modifie le pointeur de pile) à le schéma suivant pour sa pile:

- ESP—pointe sur RA
- ESP+4—pointe sur la variable a

D'un autre côté, lorsque nous appelons printf() ici nous avons exactement la même disposition de pile, excepté pour le premier argument de printf(), qui doit pointer sur la chaîne. Et c'est ce que fait notre code.

Il remplace le premier argument de la fonction par l'adresse de la chaîne et saute à printf(), comme si nous n'avions pas appelé notre fonction f(), mais directement printf(). printf() affiche la chaîne sur la sortie standard et ensuite exécute l'instruction RET qui POPs RA de la pile et l'exécution est renvoyée non pas à f() mais plutôt à l'appelant de f(), ignorant la fin de la fonction f().

Tout ceci est possible car printf() est appelée, dans tous les cas, tout à la fin de la fonction f(). Dans un certain sens, c'est similaire à la fonction longjmp()[94]. Et bien sûr, c'est fait dans un but de vitesse d'exécution.

Un cas similaire avec le compilateur ARM est décrit dans la section « printf() avec plusieurs arguments », ici (1.8.2 on page 54).

[94] Wikipédia

OllyDbg

Comme cet exemple est compliqué, traçons-le dans OllyDbg.

OllyDbg peut détecter des constructions avec switch(), et ajoute des commentaires utiles. EAX contient 2 au début, c'est la valeur du paramètre de la fonction:

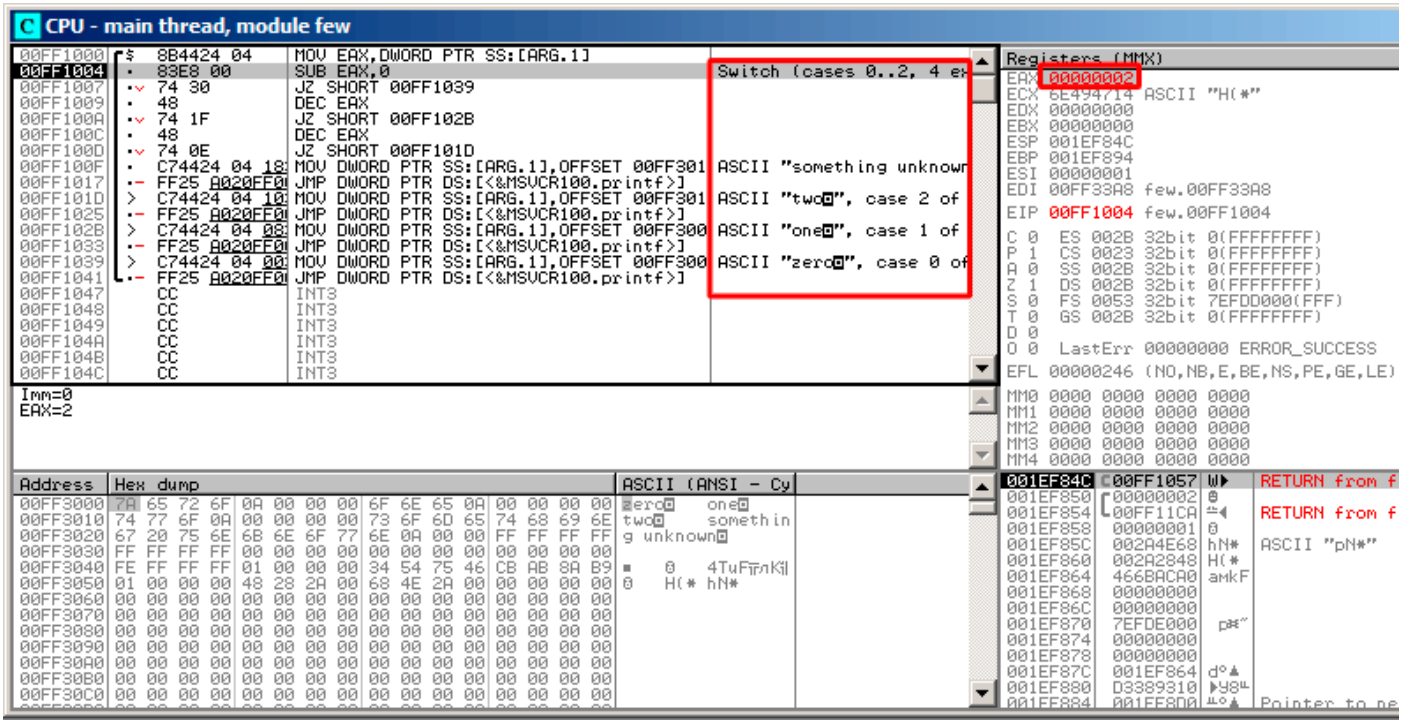


Fig. 1.42: OllyDbg : EAX contient maintenant le premier (et unique) argument de la fonction

0 est soustrait de 2 dans EAX. Bien sûr, EAX contient toujours 2. Mais le flag ZF est maintenant à 0, indiquant que le résultat est différent de zéro:

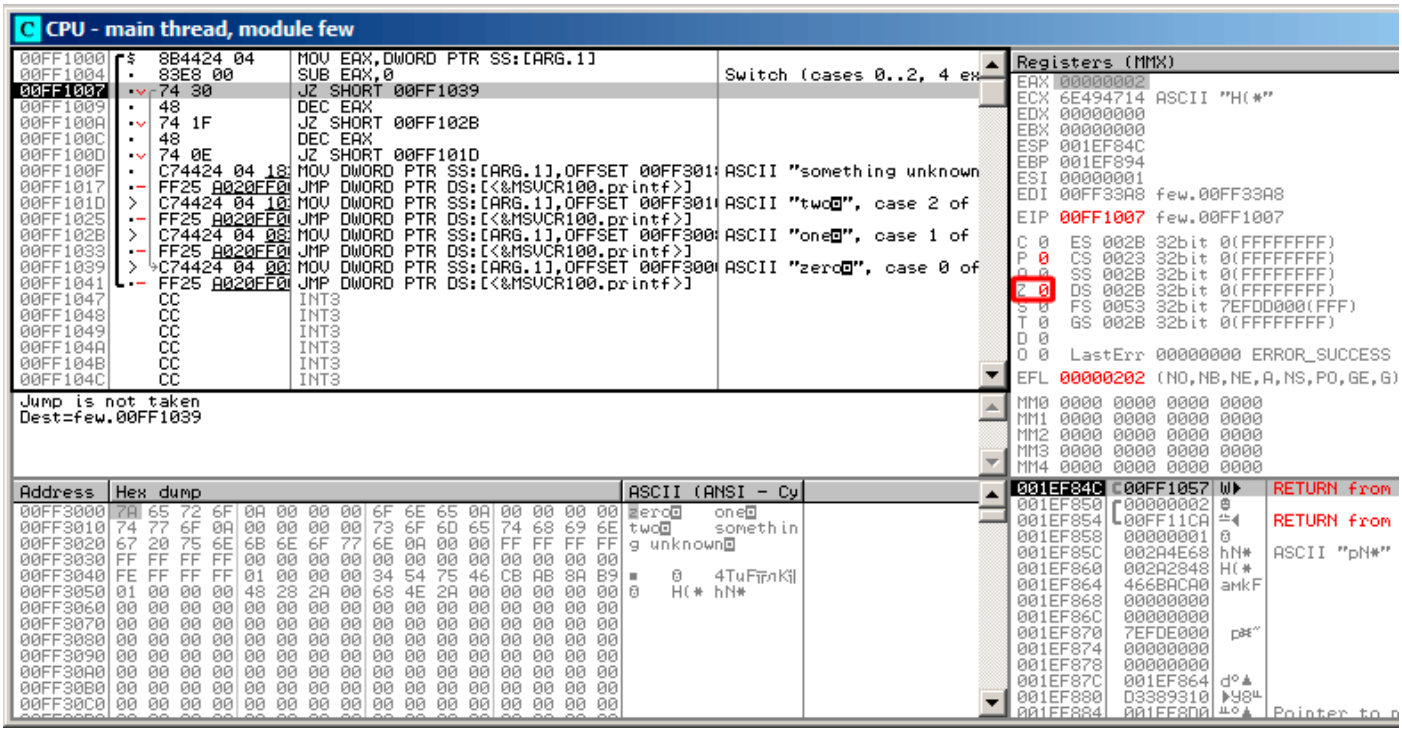


Fig. 1.43: OllyDbg : SUB exécuté

DEC est exécuté et EAX contient maintenant 1. Mais 1 est différent de zéro, donc le flag ZF est toujours à 0:

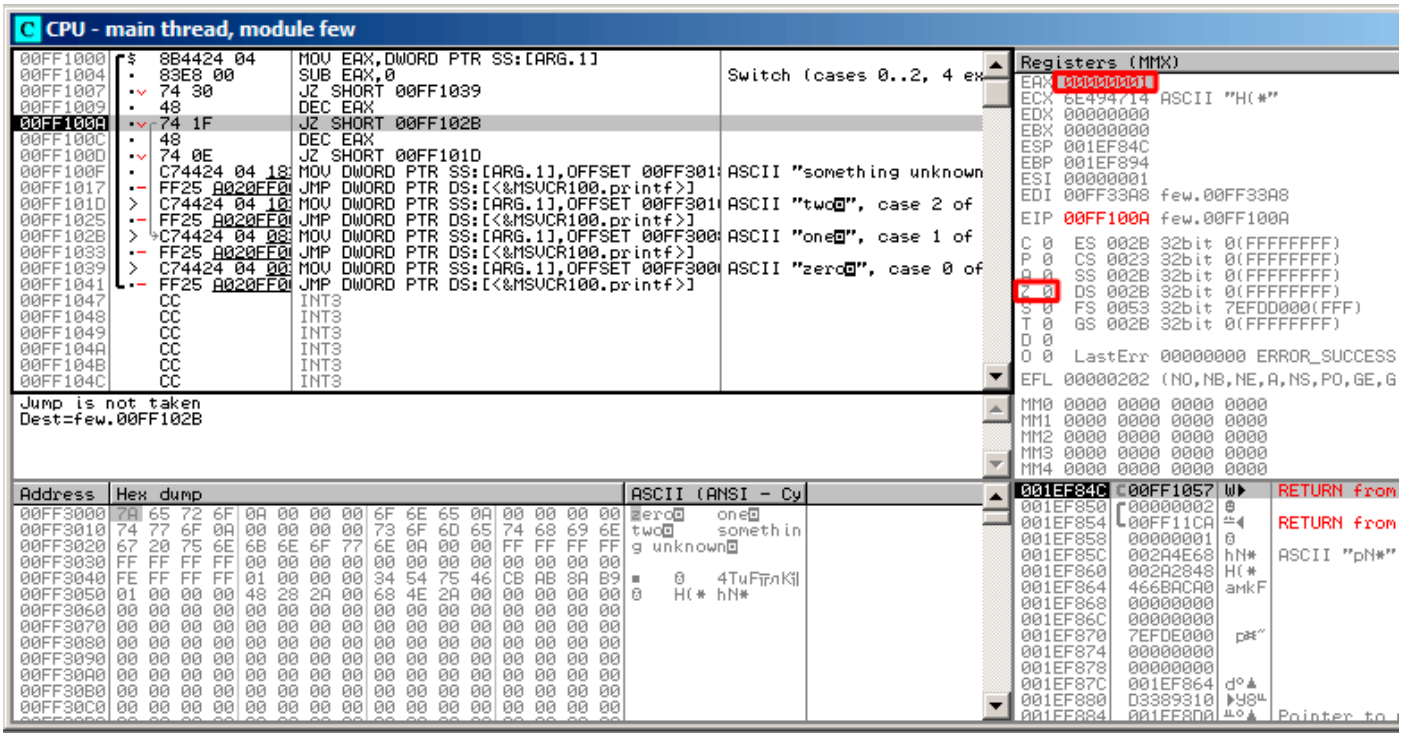


Fig. 1.44: OllyDbg : premier DEC exécuté

Le DEC suivant est exécuté. EAX contient maintenant 0 et le flag ZF est mis, car le résultat devient zéro:

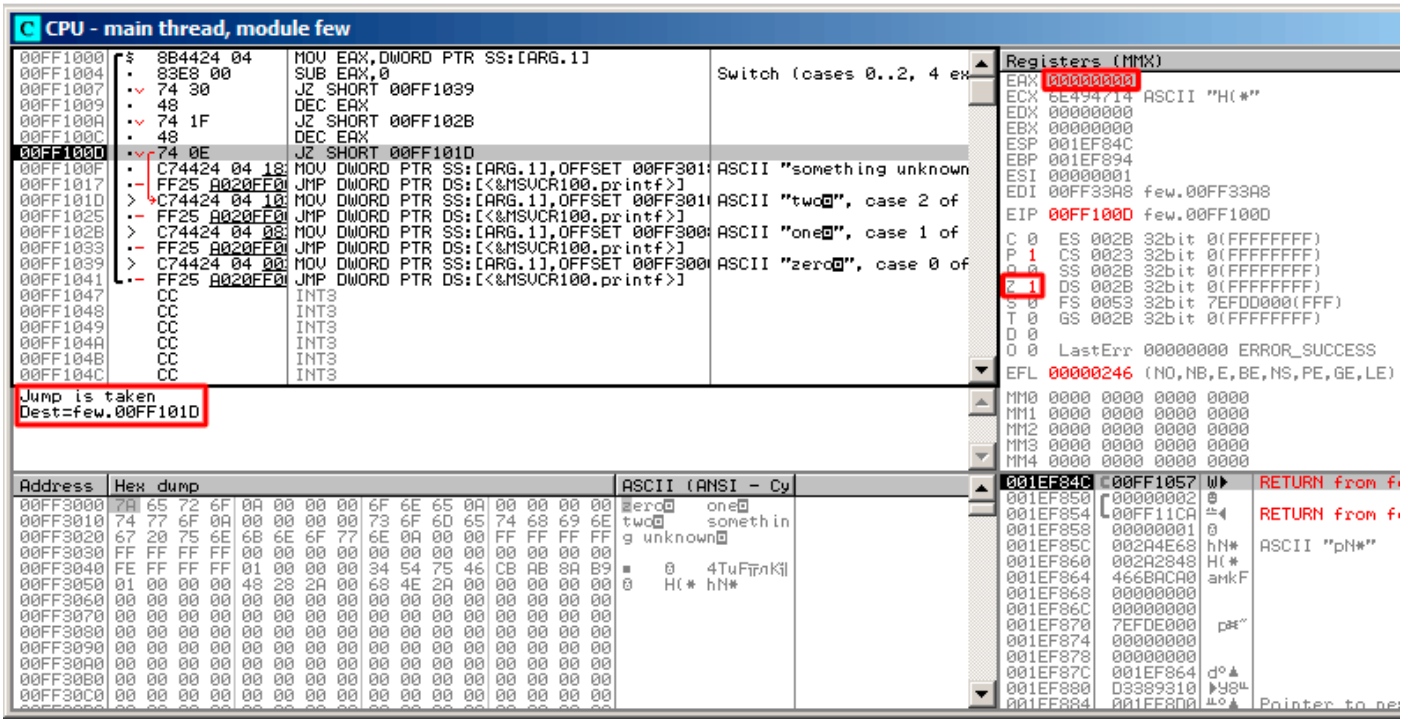


Fig. 1.45: OllyDbg : second DEC exécuté

OllyDbg montre que le saut va être effectué (*Jump is taken*).

Un pointeur sur la chaîne « two » est maintenant écrit sur la pile:

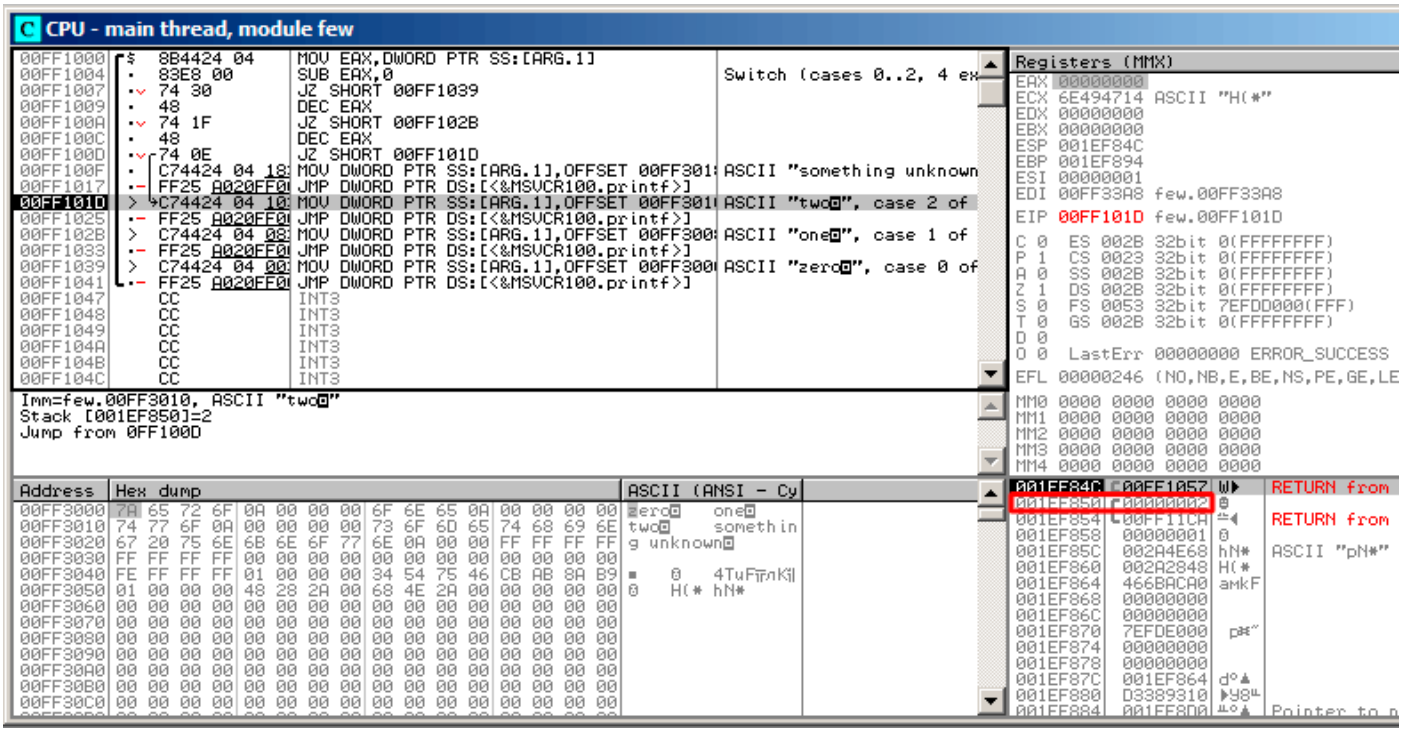


Fig. 1.46: OllyDbg : pointeur sur la chaîne qui va être écrite à la place du premier argument

Veuillez noter: l'argument de la fonction courante est 2, et 2 est maintenant sur la pile, à l'adresse `0x001EF850`.

MOV écrit le pointeur sur la chaîne à l'adresse 0x001EF850 (voir la fenêtre de la pile). Puis, le saut est effectué. Ceci est la première instruction de la fonction printf() dans MSVCR100.DLL (Cet exemple a été compilé avec le switch /MD):

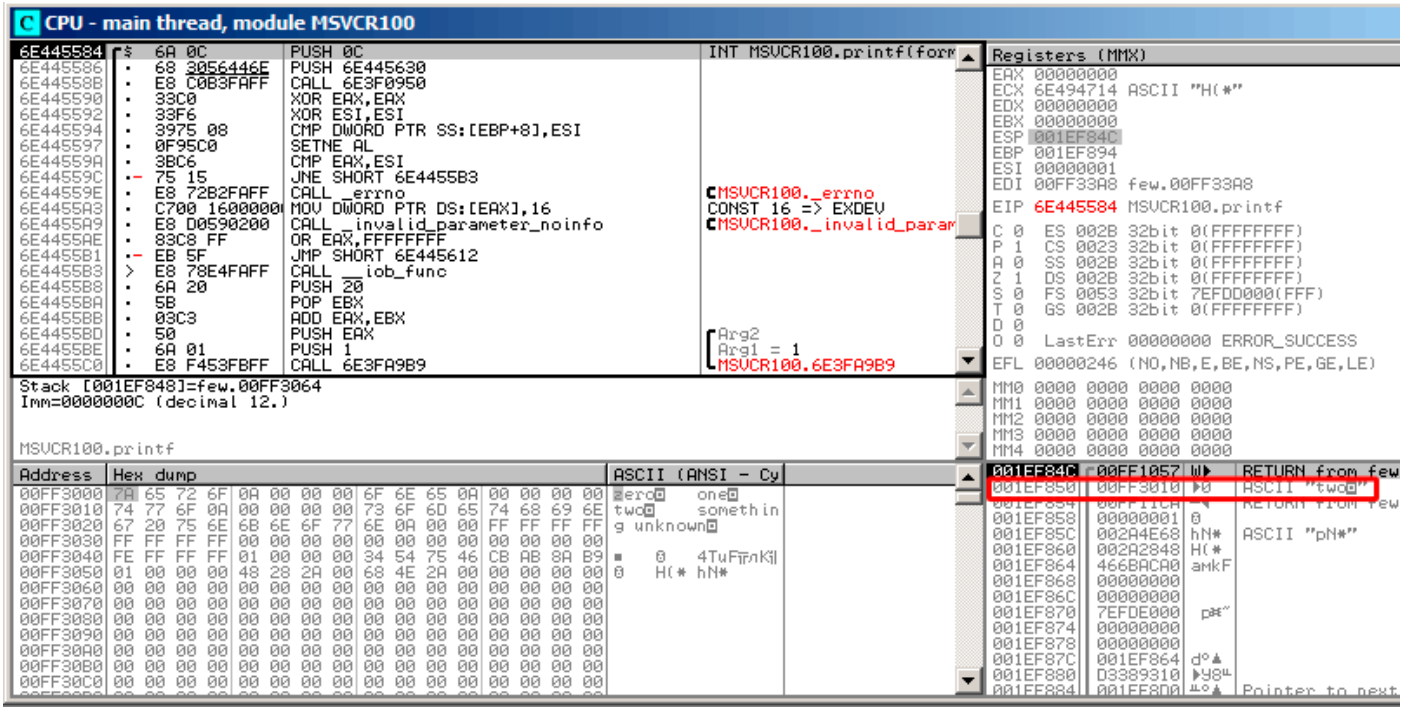


Fig. 1.47: OllyDbg : première instruction de printf() dans MSVCR100.DLL

Maintenant printf() traite la chaîne à l'adresse 0x00FF3010 comme c'est son seul argument et l'affiche.

Ceci est la dernière instruction de `printf()` :

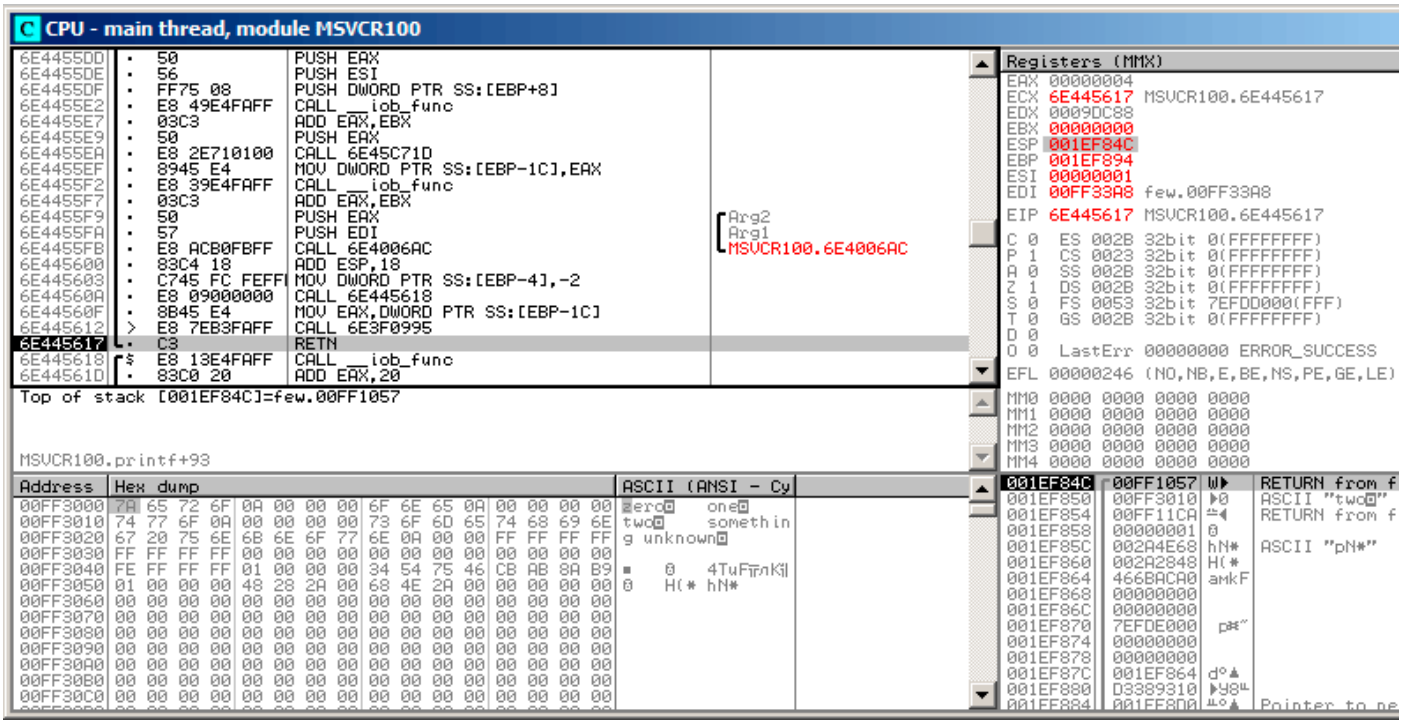


Fig. 1.48: OllyDbg : dernière instruction de `printf()` dans MSVCR100.DLL

La chaîne « two » vient juste d'être affichée dans la fenêtre console.

Maintenant, appuyez sur F7 ou F8 (enjamber) et le retour ne se fait pas sur `f()`, mais sur `main()` :

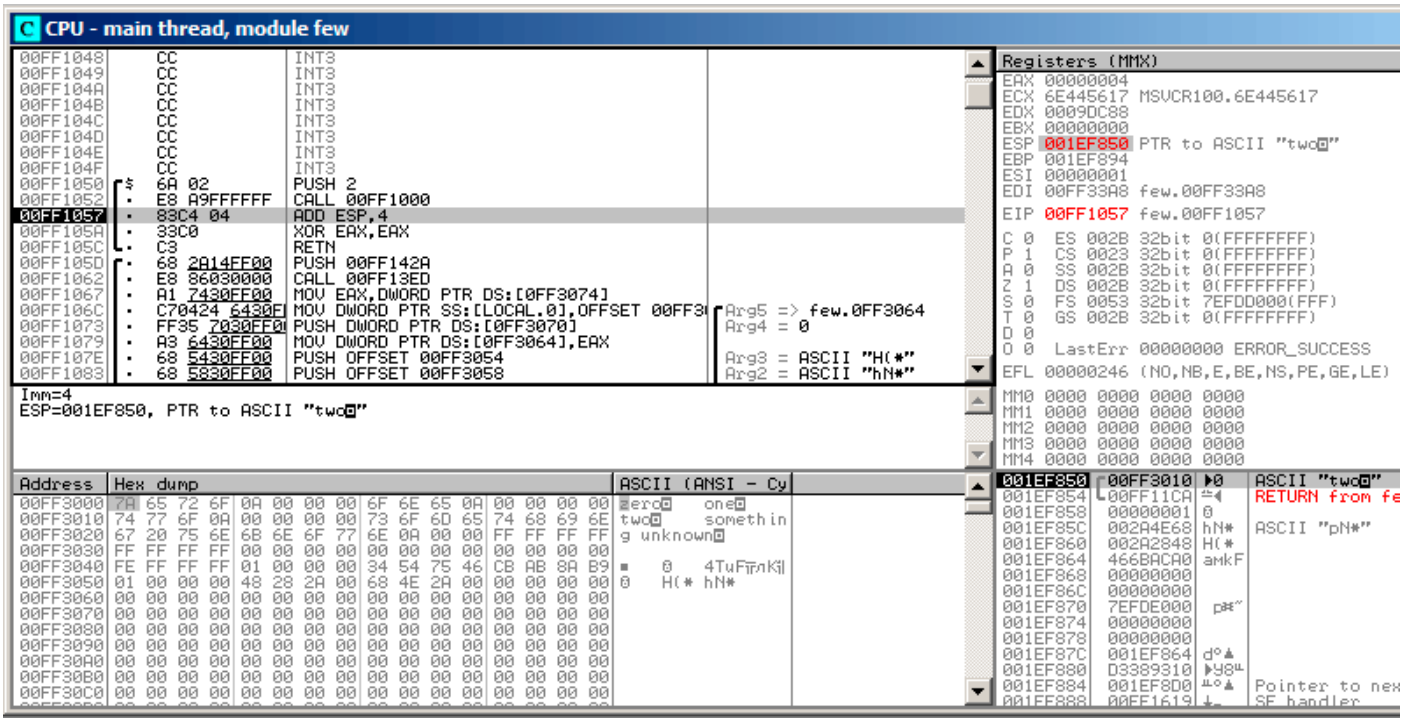


Fig. 1.49: OllyDbg : retourne à `main()`

Oui, le saut a été direct, depuis les entrailles de `printf()` vers `main()`. Car RA dans la pile pointe non pas quelque part dans `f()`, mais en fait sur `main()`. Et `CALL 0x00FF1000` a été l'instruction qui a appelé `f()`.

ARM: avec optimisation Keil 6/2013 (Mode ARM)

```
.text :0000014C             f1 :
.text :0000014C 00 00 50 E3   CMP     R0, #0
.text :00000150 13 0E 8F 02   ADREQ   R0, aZero ; "zero\n"
.text :00000154 05 00 00 0A   BEQ     loc_170
.text :00000158 01 00 50 E3   CMP     R0, #1
.text :0000015C 4B 0F 8F 02   ADREQ   R0, aOne  ; "one\n"
.text :00000160 02 00 00 0A   BEQ     loc_170
.text :00000164 02 00 50 E3   CMP     R0, #2
.text :00000168 4A 0F 8F 12   ADRNE   R0, aSomethingUnkno ; "something unknown\n"
.text :0000016C 4E 0F 8F 02   ADREQ   R0, aTwo  ; "two\n"
.text :00000170
.text :00000170             loc_170 :  ; CODE XREF : f1+8
.text :00000170                       ; f1+14
.text :00000170 78 18 00 EA   B       __2printf
```

A nouveau, en investiguant ce code, nous ne pouvons pas dire si il y avait un switch() dans le code source d'origine ou juste un ensemble de déclarations if().

En tout cas, nous voyons ici des instructions conditionnelles (comme ADREQ (*Equal*)) qui ne sont exécutées que si $R0 = 0$, et qui chargent ensuite l'adresse de la chaîne *«zero\n»* dans R0. L'instruction suivante BEQ redirige le flux d'exécution en `loc_170`, si $R0 = 0$.

Le lecteur attentif peut se demander si BEQ s'exécute correctement puisque ADREQ a déjà mis une autre valeur dans le registre R0.

Oui, elle s'exécutera correctement, car BEQ vérifie les flags mis par l'instruction CMP et ADREQ ne modifie aucun flag.

Les instructions restantes nous sont déjà familières. Il y a seulement un appel à `printf()`, à la fin, et nous avons déjà examiné cette astuce ici (1.8.2 on page 54). A la fin, il y a trois chemins vers `printf()`.

La dernière instruction, `CMP R0, #2`, est nécessaire pour vérifier si $a = 2$.

Si ce n'est pas vrai, alors ADRNE charge un pointeur sur la chaîne *«something unknown \n»* dans R0, puisque a a déjà été comparée pour savoir s'elle est égale à 0 ou 1, et nous sommes sûrs que la variable a n'est pas égale à l'un de ces nombres, à ce point. Et si $R0 = 2$, un pointeur sur la chaîne *«two\n»* sera chargé par ADREQ dans R0.

ARM: avec optimisation Keil 6/2013 (Mode Thumb)

```
.text :000000D4              f1 :
.text :000000D4 10 B5        PUSH    {R4,LR}
.text :000000D6 00 28        CMP     R0, #0
.text :000000D8 05 D0        BEQ     zero_case
.text :000000DA 01 28        CMP     R0, #1
.text :000000DC 05 D0        BEQ     one_case
.text :000000DE 02 28        CMP     R0, #2
.text :000000E0 05 D0        BEQ     two_case
.text :000000E2 91 A0        ADR     R0, aSomethingUnkno ; "something unknown\n"
.text :000000E4 04 E0        B       default_case

.text :000000E6              zero_case :  ; CODE XREF : f1+4
.text :000000E6 95 A0        ADR     R0, aZero ; "zero\n"
.text :000000E8 02 E0        B       default_case

.text :000000EA              one_case :  ; CODE XREF : f1+8
.text :000000EA 96 A0        ADR     R0, aOne ; "one\n"
.text :000000EC 00 E0        B       default_case

.text :000000EE              two_case :  ; CODE XREF : f1+C
.text :000000EE 97 A0        ADR     R0, aTwo ; "two\n"
.text :000000F0              default_case ; CODE XREF : f1+10
.text :000000F0                                      ; f1+14
.text :000000F0 06 F0 7E F8  BL      __2printf
.text :000000F4 10 BD        POP     {R4,PC}
```

Comme il y déjà été dit, il n'est pas possible d'ajouter un prédicat conditionnel à la plupart des instructions en mode Thumb, donc ce dernier est quelque peu similaire au code CISC-style x86, facilement compréhensible.

ARM64: GCC (Linaro) 4.9 sans optimisation

```
.LC12 :
        .string "zero"
.LC13 :
        .string "one"
.LC14 :
        .string "two"
.LC15 :
        .string "something unknown"
f12 :
        stp     x29, x30, [sp, -32]!
        add     x29, sp, 0
        str     w0, [x29,28]
        ldr     w0, [x29,28]
        cmp     w0, 1
        beq     .L34
        cmp     w0, 2
        beq     .L35
        cmp     w0, wzr
        bne     .L38            ; sauter au label par défaut
        adrp    x0, .LC12       ; "zero"
        add     x0, x0,  :lo12 :.LC12
        bl      puts
        b       .L32
.L34 :
        adrp    x0, .LC13       ; "one"
        add     x0, x0,  :lo12 :.LC13
        bl      puts
```

```
        b       .L32
.L35 :
        adrp    x0, .LC14       ; "two"
        add     x0, x0,  :lo12 :.LC14
        bl      puts
        b       .L32
.L38 :
        adrp    x0, .LC15       ; "something unknown"
        add     x0, x0,  :lo12 :.LC15
        bl      puts
        nop
.L32 :
        ldp     x29, x30, [sp], 32
        ret
```

Le type de la valeur d'entrée est *int*, par conséquent le registre `W0` est utilisé pour garder la valeur au lieu du registre complet `X0`.

Les pointeurs de chaîne sont passés à `puts()` en utilisant la paire d'instructions `ADRP/ADD` comme expliqué dans l'exemple « Hello, world! » : 1.5.4 on page 24.

ARM64: GCC (Linaro) 4.9 avec optimisation

```
f12 :
        cmp     w0, 1
        beq     .L31
        cmp     w0, 2
        beq     .L32
        cbz     w0, .L35
; cas par défaut
        adrp    x0, .LC15       ; "something unknown"
        add     x0, x0,  :lo12 :.LC15
        b       puts
.L35 :
        adrp    x0, .LC12       ; "zero"
        add     x0, x0,  :lo12 :.LC12
        b       puts
.L32 :
        adrp    x0, .LC14       ; "two"
        add     x0, x0,  :lo12 :.LC14
        b       puts
.L31 :
        adrp    x0, .LC13       ; "one"
        add     x0, x0,  :lo12 :.LC13
        b       puts
```

Ce morceau de code est mieux optimisé. L'instruction `CBZ` (*Compare and Branch on Zero* comparer et sauter si zéro) effectue un saut si `W0` vaut zéro. Il y a alors un saut direct à `puts()` au lieu de l'appeler, comme cela a été expliqué avant: 1.15.1 on page 154.

MIPS

Listing 1.149: GCC 4.4.5 avec optimisation (IDA)

```
f :
                lui     $gp, (__gnu_local_gp >> 16)
; est-ce 1?
                li      $v0, 1
                beq     $a0, $v0, loc_60
                la      $gp, (__gnu_local_gp & 0xFFFF) ; slot de délai de branchement
; est-ce 2?
                li      $v0, 2
                beq     $a0, $v0, loc_4C
                or      $at, $zero ; slot de délai de branchement, NOP
; jump, if not equal to 0:
                bnez    $a0, loc_38
```

```
                or      $at, $zero ; slot de délai de branchement, NOP
; cas zéro :
                lui     $a0, ($LC0 >> 16)  # "zero"
                lw      $t9, (puts & 0xFFFF)($gp)
                or      $at, $zero ; slot de délai de branchement, NOP
                jr      $t9 ; slot de délai de branchement, NOP
                la      $a0, ($LC0 & 0xFFFF)  # "zero" ; slot de délai de branchement

loc_38 :                                # CODE XREF : f+1C
                lui     $a0, ($LC3 >> 16)  # "something unknown"
                lw      $t9, (puts & 0xFFFF)($gp)
                or      $at, $zero ; slot de délai de branchement, NOP
                jr      $t9
                la      $a0, ($LC3 & 0xFFFF)  # "something unknown" ; slot de délai de
    branchement

loc_4C :                                # CODE XREF : f+14
                lui     $a0, ($LC2 >> 16)  # "two"
                lw      $t9, (puts & 0xFFFF)($gp)
                or      $at, $zero ; slot de délai de branchement, NOP
                jr      $t9
                la      $a0, ($LC2 & 0xFFFF)  # "two" ; slot de délai de branchement

loc_60 :                                # CODE XREF : f+8
                lui     $a0, ($LC1 >> 16)  # "one"
                lw      $t9, (puts & 0xFFFF)($gp)
                or      $at, $zero ; slot de délai de branchement, NOP
                jr      $t9
                la      $a0, ($LC1 & 0xFFFF)  # "one" ; slot de délai de branchement
```

La fonction se termine toujours en appelant `puts()`, donc nous voyons un saut à `puts()` (JR : « Jump Register ») au lieu de « jump and link ». Nous avons parlé de ceci avant: 1.15.1 on page 154.

Nous voyons aussi souvent l'instruction NOP après LW. Ceci est le slot de délai de chargement (« load delay slot »): un autre slot de délai (*delay slot*) en MIPS.

Une instruction suivant LW peut s'exécuter pendant que LW charge une valeur depuis la mémoire.

Toutefois, l'instruction suivante ne doit pas utiliser le résultat de LW.

Les CPU MIPS modernes ont la capacité d'attendre si l'instruction suivante utilise le résultat de LW, donc ceci est un peu démodé, mais GCC ajoute toujours des NOPs pour les anciens CPU MIPS. En général, ça peut être ignoré.

Conclusion

Un *switch()* avec peu de cas est indistinguable d'une construction avec *if/else*, par exemple: liste.1.15.1.

1.15.2 De nombreux cas

Si une déclaration `switch()` contient beaucoup de cas, il n'est pas très pratique pour le compilateur de générer un trop gros code avec de nombreuses instructions JE/JNE.

```
#include <stdio.h>

void f (int a)
{
    switch (a)
    {
    case 0: printf ("zero\n"); break;
    case 1: printf ("one\n"); break;
    case 2: printf ("two\n"); break;
    case 3: printf ("three\n"); break;
    case 4: printf ("four\n"); break;
    default : printf ("something unknown\n"); break;
    };
};
```

```
int main()
{
    f (2); // test
};
```

x86

MSVC sans optimisation

Nous obtenons (MSVC 2010):

Listing 1.150: MSVC 2010

```
tv64 = -4  ; size = 4
_a$ = 8    ; size = 4
_f    PROC
    push   ebp
    mov    ebp, esp
    push   ecx
    mov    eax, DWORD PTR _a$[ebp]
    mov    DWORD PTR tv64[ebp], eax
    cmp    DWORD PTR tv64[ebp], 4
    ja     SHORT $LN1@f
    mov    ecx, DWORD PTR tv64[ebp]
    jmp    DWORD PTR $LN11@f[ecx*4]
$LN6@f :
    push   OFFSET $SG739 ; 'zero', 0aH, 00H
    call   _printf
    add    esp, 4
    jmp    SHORT $LN9@f
$LN5@f :
    push   OFFSET $SG741 ; 'one', 0aH, 00H
    call   _printf
    add    esp, 4
    jmp    SHORT $LN9@f
$LN4@f :
    push   OFFSET $SG743 ; 'two', 0aH, 00H
    call   _printf
    add    esp, 4
    jmp    SHORT $LN9@f
$LN3@f :
    push   OFFSET $SG745 ; 'three', 0aH, 00H
    call   _printf
    add    esp, 4
    jmp    SHORT $LN9@f
$LN2@f :
    push   OFFSET $SG747 ; 'four', 0aH, 00H
    call   _printf
    add    esp, 4
    jmp    SHORT $LN9@f
$LN1@f :
    push   OFFSET $SG749 ; 'something unknown', 0aH, 00H
    call   _printf
    add    esp, 4
$LN9@f :
    mov    esp, ebp
    pop    ebp
    ret    0
    npad    2 ; aligner le label suivant
$LN11@f :
    DD    $LN6@f ; 0
    DD    $LN5@f ; 1
    DD    $LN4@f ; 2
    DD    $LN3@f ; 3
    DD    $LN2@f ; 4
_f    ENDP
```

Ce que nous voyons ici est un ensemble d'appels à `printf()` avec des arguments variés. Ils ont tous, non seulement des adresses dans la mémoire du processus, mais aussi des labels symboliques internes assignés par le compilateur. Tous ces labels ont aussi mentionnés dans la table interne `$LN11@f`.

Au début de la fonctions, si a est supérieur à 4, l'exécution est passée au labal `$LN1@f`, où `printf()` est appelé avec l'argument `'something unknown'`.

Mais si la valeur de a est inférieure ou égale à 4, elle est alors multipliée par 4 et ajoutée à l'adresse de la table `$LN11@f`. C'est ainsi qu'une adresse à l'intérieur de la table est construite, pointant exactement sur l'élément dont nous avons besoin. Par exemple, supposons que a soit égale à 2. $2 * 4 = 8$ (tous les éléments de la table sont adressés dans un processus 32-bit, c'est pourquoi les éléments ont une taille de 4 octets). L'adresse de la table `$LN11@f` + 8 est celle de l'élément de la table où le label `$LN4@f` est stocké. `JMP` prend l'adresse de `$LN4@f` dans la table et y saute.

Cette table est quelquefois appelée *jumptable* (table de saut) ou *branch table* (table de branchement)[95].

Le `printf()` correspondant est appelé avec l'argument `'two'`.
Littéralement, l'instruction `jmp DWORD PTR $LN11@f[ecx*4]` signifie *sauter au DWORD qui est stocké à l'adresse* `$LN11@f + ecx * 4`.

npad (**??** on page ??) est une macro du langage d'assemblage qui aligne le label suivant de telle sorte qu'il soit stocké à une adresse alignée sur une limite de 4 octets (ou 16 octets). C'est très adapté pour le processeur puisqu'il est capable d'aller chercher des valeurs 32-bit dans la mémoire à travers le bus mémoire, la mémoire cache, etc., de façons beaucoup plus efficace si c'est aligné.

[95] L'ensemble de la méthode était appelé *computed GOTO* (GOTO calculés) dans les premières versions de ForTran: Wikipédia. Pas très pertinent de nos jours, mais quel terme!

OllyDbg

Essayons cet exemple dans OllyDbg. La valeur d'entrée de la fonction (2) est chargée dans EAX :

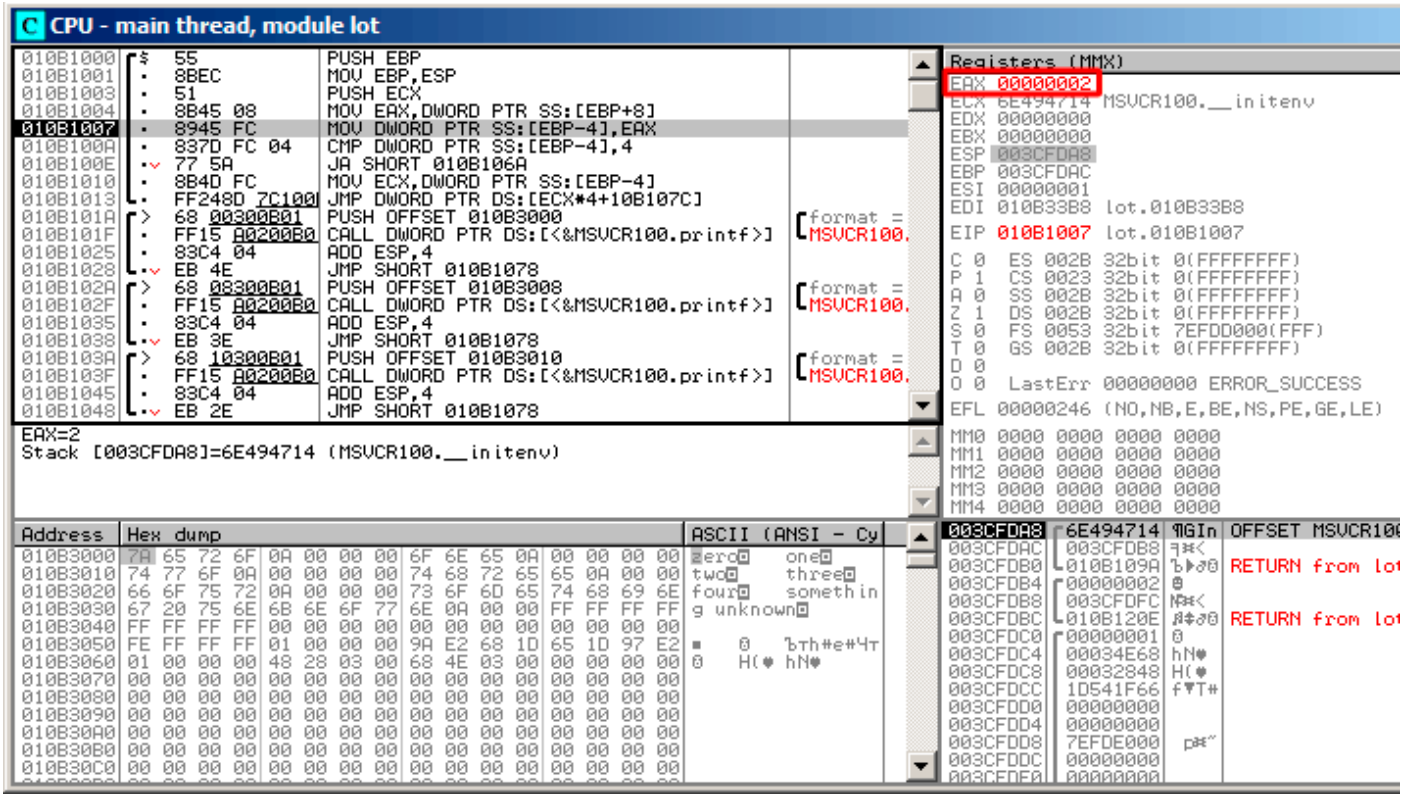


Fig. 1.50: OllyDbg : la valeur d'entrée de la fonction est chargée dans EAX

La valeur entrée est testée, est-elle plus grande que 4? Si non, le saut par « défaut » n'est pas pris:

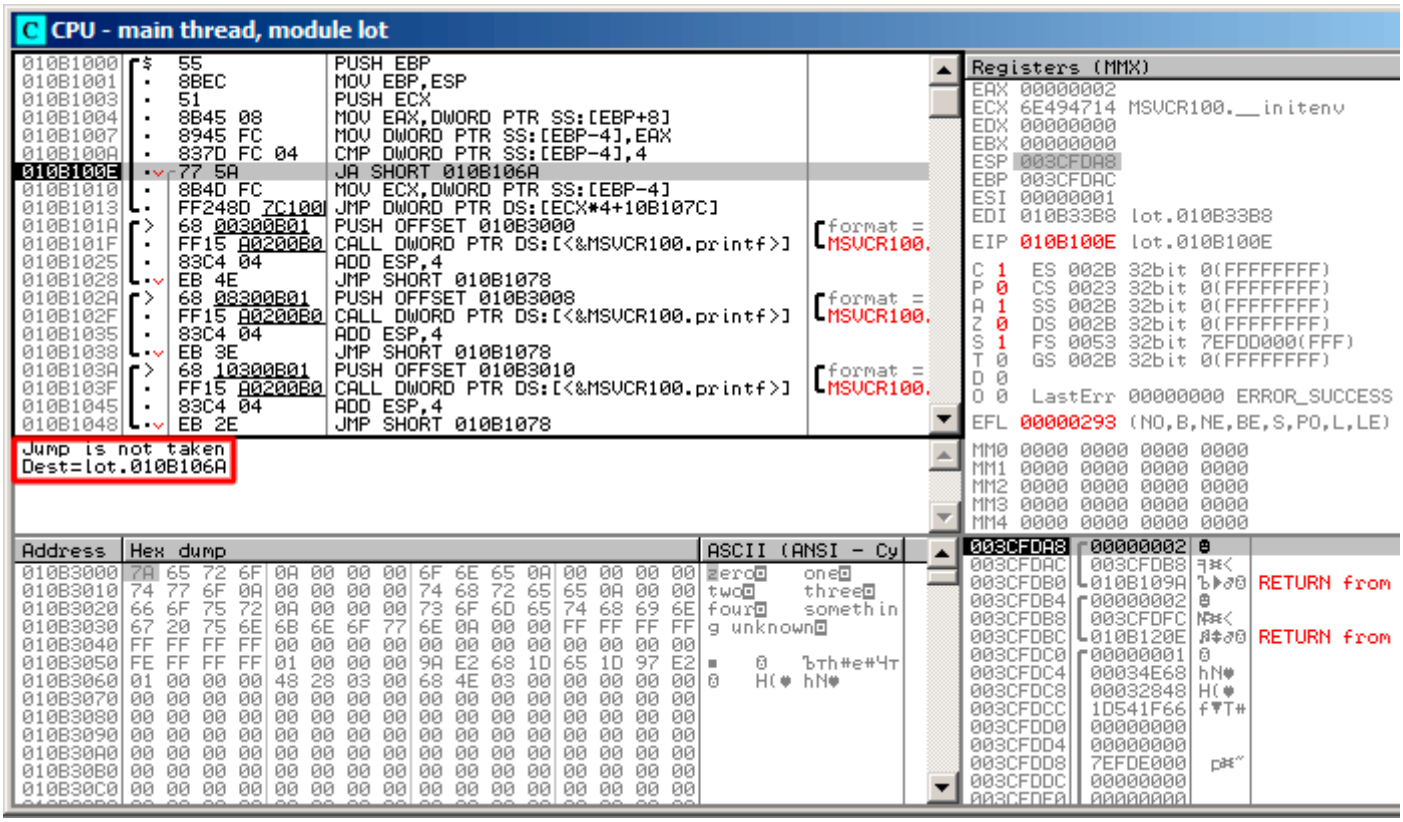


Fig. 1.51: OllyDbg : 2 n'est pas plus grand que 4: le saut n'est pas pris

Ici, nous voyons une table des sauts:

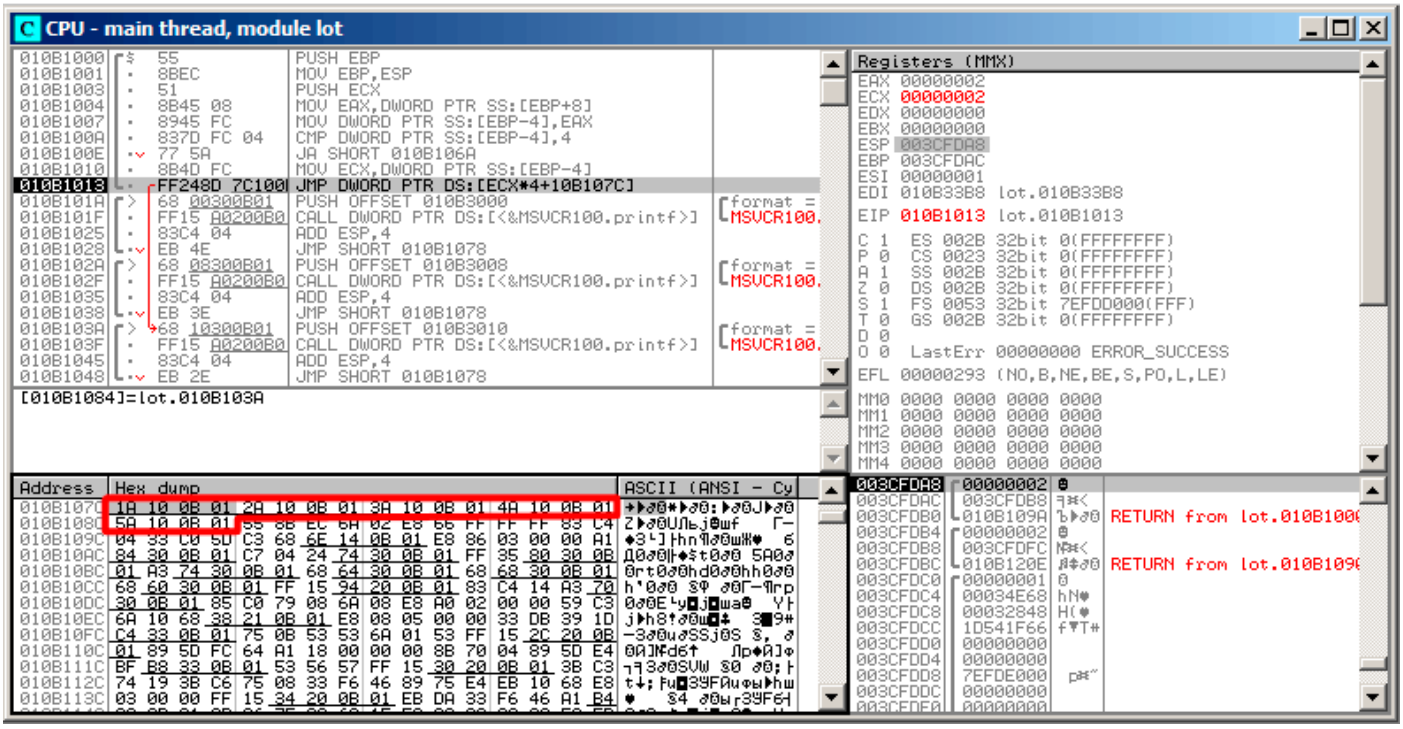


Fig. 1.52: OllyDbg : calcul de l'adresse de destination en utilisant la table des sauts

Ici, nous avons cliqué « Follow in Dump » → « Address constant », donc nous voyons maintenant la *jump-table* dans la fenêtre des données. Il y a 5 valeurs 32-bit[96]. ECX contient maintenant 2, donc le troisième élément (peut être indexé par 2[97]) de la table va être utilisé. Il est également possible de cliquer sur « Follow in Dump » → « Memory address » et OllyDbg va montrer l'élément adressé par l'instruction JMP. Il s'agit de 0x010B103A.

[96] Elles sont soulignées par OllyDbg car ce sont aussi des FIXUPs: 4.5.2 on page 570, nous y reviendrons plus tard

[97] À propos des index de tableaux, lire aussi: **??**

Après le saut, nous sommes en 0x010B103A : le code qui affiche « two » va être exécuté:

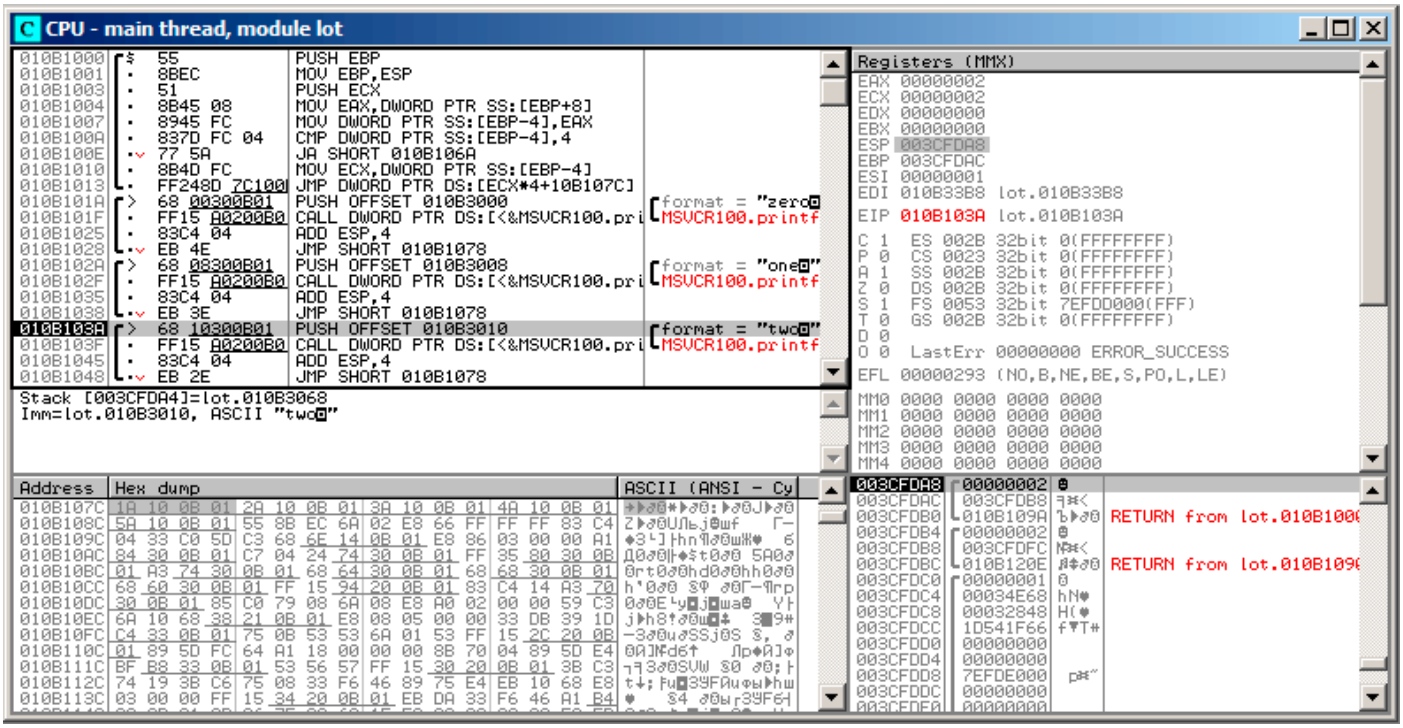


Fig. 1.53: OllyDbg : maintenant nous sommes au *cas:* label

GCC sans optimisation

Voyons ce que GCC 4.4.1 génère:

Listing 1.151: GCC 4.4.1

```
        public f
f       proc near  ; CODE XREF : main+10

var_18 = dword ptr -18h
arg_0  = dword ptr  8

        push    ebp
        mov     ebp, esp
        sub     esp, 18h
        cmp     [ebp+arg_0], 4
        ja      short loc_8048444
        mov     eax, [ebp+arg_0]
        shl     eax, 2
        mov     eax, ds :off_804855C[eax]
        jmp     eax

loc_80483FE :  ; DATA XREF : .rodata :off_804855C
        mov     [esp+18h+var_18], offset aZero  ; "zero"
        call    _puts
        jmp     short locret_8048450

loc_804840C :  ; DATA XREF : .rodata :08048560
        mov     [esp+18h+var_18], offset aOne  ; "one"
        call    _puts
        jmp     short locret_8048450

loc_804841A :  ; DATA XREF : .rodata :08048564
        mov     [esp+18h+var_18], offset aTwo  ; "two"
        call    _puts
        jmp     short locret_8048450

loc_8048428 :  ; DATA XREF : .rodata :08048568
```

```
        mov     [esp+18h+var_18], offset aThree ; "three"
        call    _puts
        jmp     short locret_8048450

loc_8048436 : ; DATA XREF : .rodata :0804856C
        mov     [esp+18h+var_18], offset aFour ; "four"
        call    _puts
        jmp     short locret_8048450

loc_8048444 : ; CODE XREF : f+A
        mov     [esp+18h+var_18], offset aSomethingUnkno ; "something unknown"
        call    _puts

locret_8048450 : ; CODE XREF : f+26
                 ; f+34...
        leave
        retn
f       endp

off_804855C dd offset loc_80483FE   ; DATA XREF : f+12
            dd offset loc_804840C
            dd offset loc_804841A
            dd offset loc_8048428
            dd offset loc_8048436
```

C'est presque la même chose, avec une petite nuance: l'argument `arg_0` est multiplié par 4 en décalant de 2 bits vers la gauche (c'est presque comme multiplier par 4) (1.18.2 on page 218). Ensuite l'adresse du label est prise depuis le tableau `off_804855C`, stockée dans EAX, et ensuite `JMP EAX` effectue le saut réel.

ARM: avec optimisation Keil 6/2013 (Mode ARM)

Listing 1.152: avec optimisation Keil 6/2013 (Mode ARM)

```
00000174             f2
00000174 05 00 50 E3     CMP     R0, #5           ; switch 5 cases
00000178 00 F1 8F 30     ADDCC   PC, PC, R0,LSL#2 ; switch jump
0000017C 0E 00 00 EA     B       default_case     ; jumptable 00000178 default case

00000180
00000180             loc_180 ; CODE XREF : f2+4
00000180 03 00 00 EA     B       zero_case        ; jumptable 00000178 case 0

00000184
00000184             loc_184 ; CODE XREF : f2+4
00000184 04 00 00 EA     B       one_case         ; jumptable 00000178 case 1

00000188
00000188             loc_188 ; CODE XREF : f2+4
00000188 05 00 00 EA     B       two_case         ; jumptable 00000178 case 2

0000018C
0000018C             loc_18C ; CODE XREF : f2+4
0000018C 06 00 00 EA     B       three_case       ; jumptable 00000178 case 3

00000190
00000190             loc_190 ; CODE XREF : f2+4
00000190 07 00 00 EA     B       four_case        ; jumptable 00000178 case 4

00000194
00000194             zero_case ; CODE XREF : f2+4
00000194                       ; f2 :loc_180
00000194 EC 00 8F E2     ADR     R0, aZero        ; jumptable 00000178 case 0
00000198 06 00 00 EA     B       loc_1B8

0000019C
0000019C             one_case ; CODE XREF : f2+4
0000019C                      ; f2 :loc_184
```

```
0000019C EC 00 8F E2     ADR     R0, aOne        ; jumptable 00000178 case 1
000001A0 04 00 00 EA     B       loc_1B8

000001A4
000001A4             two_case ; CODE XREF : f2+4
000001A4                      ; f2 :loc_188
000001A4 01 0C 8F E2     ADR     R0, aTwo        ; jumptable 00000178 case 2
000001A8 02 00 00 EA     B       loc_1B8

000001AC
000001AC             three_case ; CODE XREF : f2+4
000001AC                        ; f2 :loc_18C
000001AC 01 0C 8F E2     ADR     R0, aThree      ; jumptable 00000178 case 3
000001B0 00 00 00 EA     B       loc_1B8

000001B4
000001B4             four_case ; CODE XREF : f2+4
000001B4                       ; f2 :loc_190
000001B4 01 0C 8F E2     ADR     R0, aFour       ; jumptable 00000178 case 4
000001B8
000001B8             loc_1B8   ; CODE XREF : f2+24
000001B8                       ; f2+2C
000001B8 66 18 00 EA     B       __2printf

000001BC
000001BC             default_case ; CODE XREF : f2+4
000001BC                          ; f2+8
000001BC D4 00 8F E2     ADR     R0, aSomethingUnkno ; jumptable 00000178 default case
000001C0 FC FF FF EA     B       loc_1B8
```

Ce code utilise les caractéristiques du mode ARM dans lequel toutes les instructions ont une taille fixe de 4 octets.

Gardons à l'esprit que la valeur maximale de a est 4 et que toute autre valeur supérieure provoquera l'affichage de la chaîne *«something unknown\n»*

La première instruction `CMP R0, #5` compare la valeur entrée dans a avec 5.

[98] L'instruction suivante, `ADDCC PC, PC, R0,LSL#2`, est exécutée si et seulement si $R0 < 5$ (*CC=Carry clear / Less than* retenue vide, inférieur à). Par conséquent, si `ADDCC` n'est pas exécutée (c'est le cas $R0 \geq 5$), un saut au label *default_case* se produit.

Mais si $R0 < 5$ et que `ADDCC` est exécuté, voici ce qui se produit:

La valeur dans `R0` est multipliée par 4. En fait, le suffixe de l'instruction `LSL#2` signifie « décalage à gauche de 2 bits ». Mais comme nous le verrons plus tard (1.18.2 on page 217) dans la section « Décalages », décaler de 2 bits vers la gauche est équivalent à multiplier par 4.

Puis, nous ajoutons $R0 * 4$ à la valeur courante du PC, et sautons à l'une des instructions `B` (*Branch*) situées plus bas.

Au moment de l'exécution de `ADDCC`, la valeur du PC est en avance de 8 octets (`0x180`) sur l'adresse à laquelle l'instruction `ADDCC` se trouve (`0x178`), ou, autrement dit, en avance de 2 instructions.

C'est ainsi que le pipeline des processeurs ARM fonctionne: lorsque `ADDCC` est exécutée, le processeur, à ce moment, commence à préparer les instructions après la suivante, c'est pourquoi PC pointe ici. Cela doit être mémorisé.

Si $a = 0$, elle sera ajoutée à la valeur de PC, et la valeur courante de PC sera écrite dans PC (qui est 8 octets en avant) et un saut au label *loc_180* sera effectué, qui est 8 octets en avant du point où l'instruction se trouve.

Si $a = 1$, alors $PC + 8 + a * 4 = PC + 8 + 1 * 4 = PC + 12 = 0x184$ sera écrit dans PC, qui est l'adresse du label *loc_184*.

A chaque fois que l'on ajoute 1 à a, le PC résultant est incrémenté de 4.

4 est la taille des instructions en mode ARM, et donc, la longueur de chaque instruction `B` desquelles il y a 5 à la suite.

Chacune de ces cinq instructions `B` passe le contrôle plus loin, à ce qui a été programmé dans le *switch()*.

[98] ADD—addition

Le chargement du pointeur sur la chaîne correspondante se produit ici, etc.

ARM: avec optimisation Keil 6/2013 (Mode Thumb)

Listing 1.153: avec optimisation Keil 6/2013 (Mode Thumb)

```
000000F6                        EXPORT f2
000000F6                    f2
000000F6 10 B5                  PUSH    {R4,LR}
000000F8 03 00                  MOVS    R3, R0
000000FA 06 F0 69 F8            BL      __ARM_common_switch8_thumb ; switch 6 cases

000000FE 05                     DCB 5
000000FF 04 06 08 0A 0C 10      DCB 4, 6, 8, 0xA, 0xC, 0x10 ; jump table for switch statement
00000105 00                     ALIGN 2
00000106
00000106                    zero_case ; CODE XREF : f2+4
00000106 8D A0                  ADR     R0, aZero ; jumptable 000000FA case 0
00000108 06 E0                  B       loc_118

0000010A
0000010A                    one_case ; CODE XREF : f2+4
0000010A 8E A0                  ADR     R0, aOne ; jumptable 000000FA case 1
0000010C 04 E0                  B       loc_118

0000010E
0000010E                    two_case ; CODE XREF : f2+4
0000010E 8F A0                  ADR     R0, aTwo ; jumptable 000000FA case 2
00000110 02 E0                  B       loc_118

00000112
00000112                    three_case ; CODE XREF : f2+4
00000112 90 A0                  ADR     R0, aThree ; jumptable 000000FA case 3
00000114 00 E0                  B       loc_118

00000116
00000116                    four_case ; CODE XREF : f2+4
00000116 91 A0                  ADR     R0, aFour ; jumptable 000000FA case 4
00000118
00000118                    loc_118 ; CODE XREF : f2+12
00000118                            ; f2+16
00000118 06 F0 6A F8            BL      __2printf
0000011C 10 BD                  POP     {R4,PC}

0000011E
0000011E                    default_case ; CODE XREF : f2+4
0000011E 82 A0                  ADR     R0, aSomethingUnkno ; jumptable 000000FA default case
00000120 FA E7                  B       loc_118


000061D0                        EXPORT __ARM_common_switch8_thumb
000061D0                    __ARM_common_switch8_thumb ; CODE XREF : example6_f2+4
000061D0 78 47                  BX      PC

000061D2 00 00                  ALIGN 4
000061D2                    ; End of function __ARM_common_switch8_thumb
000061D2
000061D4                    __32__ARM_common_switch8_thumb ; CODE XREF : ↙
    ↳ __ARM_common_switch8_thumb
000061D4 01 C0 5E E5            LDRB    R12, [LR,#-1]
000061D8 0C 00 53 E1            CMP     R3, R12
000061DC 0C 30 DE 27            LDRCSB  R3, [LR,R12]
000061E0 03 30 DE 37            LDRCCB  R3, [LR,R3]
000061E4 83 C0 8E E0            ADD     R12, LR, R3,LSL#1
000061E8 1C FF 2F E1            BX      R12
000061E8                    ; End of function __32__ARM_common_switch8_thumb
```

On ne peut pas être sûr que toutes ces instructions en mode Thumb et Thumb-2 ont la même taille. On peut même dire que les instructions dans ces modes ont une longueur variable, tout comme en x86.

Donc, une table spéciale est ajoutée, qui contient des informations sur le nombre de cas (sans inclure celui par défaut), et un offset pour chaque label auquel le contrôle doit être passé dans chaque cas.

Une fonction spéciale est présente ici qui s'occupe de la table et du passage du contrôle, appelée *__ARM_common_switch8_thumb*. Elle commence avec `BX PC`, dont la fonction est de passer le mode du processeur en ARM. Ensuite, vous voyez la fonction pour le traitement de la table.

C'est trop avancé pour être détaillé ici, donc passons cela.

Il est intéressant de noter que la fonction utilise le registre LR comme un pointeur sur la table.

En effet, après l'appel de cette fonction, LR contient l'adresse après l'instruction `BL __ARM_common_switch8_thumb`, où la table commence.

Il est intéressant de noter que le code est généré comme une fonction indépendante afin de la ré-utiliser, donc le compilateur ne générera pas le même code pour chaque déclaration switch().

IDA l'a correctement identifié comme une fonction de service et une table, et a ajouté un commentaire au label comme `jumptable 000000FA case 0`.

MIPS

Listing 1.154: GCC 4.4.5 avec optimisation (IDA)

```
f :
                lui     $gp, (__gnu_local_gp >> 16)
; sauter en loc_24 si la valeur entrée est plus petite que 5:
                sltiu   $v0, $a0, 5
                bnez    $v0, loc_24
                la      $gp, (__gnu_local_gp & 0xFFFF) ; slot de délai de branchement
; la valeur entrée est supérieur ou égale à 5.
; afficher "something unknown" et terminer :
                lui     $a0, ($LC5 >> 16)  # "something unknown"
                lw      $t9, (puts & 0xFFFF)($gp)
                or      $at, $zero ; NOP
                jr      $t9
                la      $a0, ($LC5 & 0xFFFF)  # "something unknown"
                                              ; slot de délai de branchement

loc_24 :                                # CODE XREF : f+8
; charger l'adresse de la table de branchement/saut
; LA est une pseudo instruction, il s'agit en fait de LUI et ADDIU :
                la      $v0, off_120
; multiplier la valeur entrés par 4:
                sll     $a0, 2
; ajouter la valeur multipliée et l'adresse de la table de saut:
                addu    $a0, $v0, $a0
; charger l'élément de la table de saut :
                lw      $v0, 0($a0)
                or      $at, $zero ; NOP
; sauter à l'adresse que nous avons dans la table de saut:
                jr      $v0
                or      $at, $zero ; slot de délai de branchement, NOP

sub_44 :                                # DATA XREF : .rodata :0000012C
; afficher "three" et terminer
                lui     $a0, ($LC3 >> 16)  # "three"
                lw      $t9, (puts & 0xFFFF)($gp)
                or      $at, $zero ; NOP
                jr      $t9
                la      $a0, ($LC3 & 0xFFFF)  # "three" ; slot de délai de branchement

sub_58 :                                # DATA XREF : .rodata :00000130
; afficher "four" et terminer
                lui     $a0, ($LC4 >> 16)  # "four"
                lw      $t9, (puts & 0xFFFF)($gp)
                or      $at, $zero ; NOP
                jr      $t9
                la      $a0, ($LC4 & 0xFFFF)  # "four" ; slot de délai de branchement
```

```
sub_6C :                                # DATA XREF : .rodata :off_120
; afficher "zero" et terminer
                lui     $a0, ($LC0 >> 16)  # "zero"
                lw      $t9, (puts & 0xFFFF)($gp)
                or      $at, $zero ; NOP
                jr      $t9
                la      $a0, ($LC0 & 0xFFFF)  # "zero" ; slot de délai de branchement

sub_80 :                                # DATA XREF : .rodata :00000124
; afficher "one" et terminer
                lui     $a0, ($LC1 >> 16)  # "one"
                lw      $t9, (puts & 0xFFFF)($gp)
                or      $at, $zero ; NOP
                jr      $t9
                la      $a0, ($LC1 & 0xFFFF)  # "one" ; slot de délai de branchement

sub_94 :                                # DATA XREF : .rodata :00000128
; afficher "two" et terminer
                lui     $a0, ($LC2 >> 16)  # "two"
                lw      $t9, (puts & 0xFFFF)($gp)
                or      $at, $zero ; NOP
                jr      $t9
                la      $a0, ($LC2 & 0xFFFF)  # "two" ; slot de délai de branchement

; peut être mis dans une section .rodata:
off_120 :        .word sub_6C
                .word sub_80
                .word sub_94
                .word sub_44
                .word sub_58
```

La nouvelle instruction pour nous est SLTIU (« Set on Less Than Immediate Unsigned » Mettre si inférieur à la valeur immédiate non signée).

Ceci est la même que SLTU (« Set on Less Than Unsigned »), mais « I » signifie « immediate », i.e., un nombre doit être spécifié dans l'instruction elle-même.

BNEZ est « Branch if Not Equal to Zero ».

Le code est très proche de l'autre ISAs. SLL (« Shift Word Left Logical ») effectue une multiplication par 4.

MIPS est un CPU 32-bit après tout, donc toutes les adresses de la *jumtable* sont 32-bits.

Conclusion

Squelette grossier d'un *switch()* :

Listing 1.155: x86

```
MOV REG, input
CMP REG, 4 ; nombre maximal de cas
JA default
SHL REG, 2 ; trouver l'élément dans la table. décaler de 3 bits en x64.
MOV REG, jump_table[REG]
JMP REG

case1 :
    ; faire quelque chose
    JMP exit
case2 :
    ; faire quelque chose
    JMP exit
case3 :
    ; faire quelque chose
    JMP exit
case4 :
    ; faire quelque chose
    JMP exit
case5 :
    ; faire quelque chose
```

```
    JMP exit

default :

    ...

exit :

    ....

jump_table dd case1
           dd case2
           dd case3
           dd case4
           dd case5
```

Le saut à une adresse de la table de saut peut aussi être implémenté en utilisant cette instruction: `JMP jump_table[REG*4]`. Ou `JMP jump_table[REG*8]` en x64.

Une table de saut est juste un tableau de pointeurs, comme celle décrite plus loin: 1.20.5 on page 288.

1.15.3 Lorsqu'il y a quelques déclarations *case* dans un bloc

Voici une construction très répandue: quelques déclarations *case* pour un seul bloc:

```
#include <stdio.h>

void f(int a)
{
        switch (a)
        {
        case 1:
        case 2:
        case 7:
        case 10:
                printf ("1, 2, 7, 10\n");
                break;
        case 3:
        case 4:
        case 5:
        case 6:
                printf ("3, 4, 5\n");
                break;
        case 8:
        case 9:
        case 20:
        case 21:
                printf ("8, 9, 21\n");
                break;
        case 22:
                printf ("22\n");
                break;
        default :
                printf ("default\n");
                break;
        };
};

int main()
{
        f(4);
};
```

C'est souvent du gaspillage de générer un bloc pour chaque cas possible, c'est pourquoi ce qui se fait d'habitude, c'est de générer un bloc et une sorte de répartiteur.

MSVC

Listing 1.156: MSVC 2010 avec optimisation

```
1  $SG2798 DB      '1, 2, 7, 10', 0aH, 00H
2  $SG2800 DB      '3, 4, 5', 0aH, 00H
3  $SG2802 DB      '8, 9, 21', 0aH, 00H
4  $SG2804 DB      '22', 0aH, 00H
5  $SG2806 DB      'default', 0aH, 00H
6
7  _a$ = 8
8  _f      PROC
9          mov     eax, DWORD PTR _a$[esp-4]
10         dec     eax
11         cmp     eax, 21
12         ja      SHORT $LN1@f
13         movzx   eax, BYTE PTR $LN10@f[eax]
14         jmp     DWORD PTR $LN11@f[eax*4]
15 $LN5@f :
16         mov     DWORD PTR _a$[esp-4], OFFSET $SG2798 ; '1, 2, 7, 10'
17         jmp     DWORD PTR __imp__printf
18 $LN4@f :
19         mov     DWORD PTR _a$[esp-4], OFFSET $SG2800 ; '3, 4, 5'
20         jmp     DWORD PTR __imp__printf
21 $LN3@f :
22         mov     DWORD PTR _a$[esp-4], OFFSET $SG2802 ; '8, 9, 21'
23         jmp     DWORD PTR __imp__printf
24 $LN2@f :
25         mov     DWORD PTR _a$[esp-4], OFFSET $SG2804 ; '22'
26         jmp     DWORD PTR __imp__printf
27 $LN1@f :
28         mov     DWORD PTR _a$[esp-4], OFFSET $SG2806 ; 'default'
29         jmp     DWORD PTR __imp__printf
30         npad    2 ; aligner la table $LN11@f sur une limite de 16-octet
31 $LN11@f :
32         DD      $LN5@f ; afficher '1, 2, 7, 10'
33         DD      $LN4@f ; afficher '3, 4, 5'
34         DD      $LN3@f ; afficher '8, 9, 21'
35         DD      $LN2@f ; afficher '22'
36         DD      $LN1@f ; afficher 'default'
37 $LN10@f :
38         DB      0 ; a=1
39         DB      0 ; a=2
40         DB      1 ; a=3
41         DB      1 ; a=4
42         DB      1 ; a=5
43         DB      1 ; a=6
44         DB      0 ; a=7
45         DB      2 ; a=8
46         DB      2 ; a=9
47         DB      0 ; a=10
48         DB      4 ; a=11
49         DB      4 ; a=12
50         DB      4 ; a=13
51         DB      4 ; a=14
52         DB      4 ; a=15
53         DB      4 ; a=16
54         DB      4 ; a=17
55         DB      4 ; a=18
56         DB      4 ; a=19
57         DB      2 ; a=20
58         DB      2 ; a=21
59         DB      3 ; a=22
60 _f      ENDP
```

Nous voyons deux tables ici: la première (`$LN10@f`) est une table d'index, et la seconde (`$LN11@f`) est un tableau de pointeurs sur les blocs.

Tout d'abord, la valeur entrée est utilisée comme un index dans la table d'index (ligne 13).

Voici un petit récapitulatif pour les valeurs dans la table: 0 est le premier bloc *case* (pour les valeurs 1, 2, 7, 10), 1 est le second (pour les valeurs 3, 4, 5), 2 est le troisième (pour les valeurs 8, 9, 21), 3 est le quatrième (pour la valeur 22), 4 est pour le bloc par défaut.

Ici, nous obtenons un index pour la seconde table de pointeurs sur du code et nous y sautons (ligne 14).

Il est intéressant de remarquer qu'il n'y a pas de cas pour une valeur d'entrée de 0.

C'est pourquoi nous voyons l'instruction DEC à la ligne 10, et la table commence à $a = 1$, car il n'y a pas besoin d'allouer un élément dans la table pour $a = 0$.

C'est un pattern très répandu.

Donc, pourquoi est-ce que c'est économique ? Pourquoi est-ce qu'il n'est pas possible de faire comme avant (1.15.2 on page 172), avec une seule table consistant en des pointeurs vers les blocs? La raison est que les index des éléments de la table sont 8-bit, donc c'est plus compact.

GCC

GCC génère du code de la façon dont nous avons déjà discuté (1.15.2 on page 172), en utilisant juste une table de pointeurs.

ARM64: GCC 4.9.1 avec optimisation

Il n'y a pas de code à exécuter si la valeur entrée est 0, c'est pourquoi GCC essaye de rendre la table des sauts plus compacte et donc il commence avec la valeur d'entrée 1.

GCC 4.9.1 pour ARM64 utilise un truc encore plus astucieux. Il est capable d'encoder tous les offsets en octets 8-bit.

Rappelons-nous que toutes les instructions ARM64 ont une taille de 4 octets.

GCC utilise le fait que tous les offsets de mon petit exemple sont tous proche l'un de l'autre. Donc la table des sauts consiste en de simple octets.

Listing 1.157: avec optimisation GCC 4.9.1 ARM64

```
f14 :
; valeur entrée dans W0
        sub     w0, w0, #1
        cmp     w0, 21
; branchement si inférieur ou égal (non signé) :
        bls     .L9
.L2 :
; afficher "default":
        adrp    x0, .LC4
        add     x0, x0,  :lo12 :.LC4
        b       puts
.L9 :
; charger l'adresse de la table des sauts dans X1 :
        adrp    x1, .L4
        add     x1, x1,  :lo12 :.L4
; W0=input_value-1
; charger un octet depuis la table :
        ldrb    w0, [x1,w0,uxtw]
; charger l'adresse du label .Lrtx4 :
        adr     x1, .Lrtx4
; multiplier l'élément de la table par 4 (en décalant de 2 bits à gauche) et; ajouter (ou
    soustraire) à l'adresse de .Lrtx4 :
        add     x0, x1, w0, sxtb #2
; sauter à l'adresse calculée:
        br      x0
; ce label pointe dans le segment de code (text) :
.Lrtx4 :
        .section        .rodata
; tout ce qui se trouve après la déclaration ".section" est alloué dans le segment de données
; en lecture seule (rodata) :
.L4 :
        .byte   (.L3 - .Lrtx4) / 4      ; case 1
        .byte   (.L3 - .Lrtx4) / 4      ; case 2
```

```
        .byte   (.L5 - .Lrtx4) / 4      ; case 3
        .byte   (.L5 - .Lrtx4) / 4      ; case 4
        .byte   (.L5 - .Lrtx4) / 4      ; case 5
        .byte   (.L5 - .Lrtx4) / 4      ; case 6
        .byte   (.L3 - .Lrtx4) / 4      ; case 7
        .byte   (.L6 - .Lrtx4) / 4      ; case 8
        .byte   (.L6 - .Lrtx4) / 4      ; case 9
        .byte   (.L3 - .Lrtx4) / 4      ; case 10
        .byte   (.L2 - .Lrtx4) / 4      ; case 11
        .byte   (.L2 - .Lrtx4) / 4      ; case 12
        .byte   (.L2 - .Lrtx4) / 4      ; case 13
        .byte   (.L2 - .Lrtx4) / 4      ; case 14
        .byte   (.L2 - .Lrtx4) / 4      ; case 15
        .byte   (.L2 - .Lrtx4) / 4      ; case 16
        .byte   (.L2 - .Lrtx4) / 4      ; case 17
        .byte   (.L2 - .Lrtx4) / 4      ; case 18
        .byte   (.L2 - .Lrtx4) / 4      ; case 19
        .byte   (.L6 - .Lrtx4) / 4      ; case 20
        .byte   (.L6 - .Lrtx4) / 4      ; case 21
        .byte   (.L7 - .Lrtx4) / 4      ; case 22
        .text
; tout ce qui se trouve après la déclaration ".text" est alloué dans le segment de code
   (text) :
.L7 :
; afficher "22"
        adrp    x0, .LC3
        add     x0, x0,  :lo12 :.LC3
        b       puts
.L6 :
; afficher "8, 9, 21"
        adrp    x0, .LC2
        add     x0, x0,  :lo12 :.LC2
        b       puts
.L5 :
; afficher "3, 4, 5"
        adrp    x0, .LC1
        add     x0, x0,  :lo12 :.LC1
        b       puts
.L3 :
; afficher "1, 2, 7, 10"
        adrp    x0, .LC0
        add     x0, x0,  :lo12 :.LC0
        b       puts
.LC0 :
        .string "1, 2, 7, 10"
.LC1 :
        .string "3, 4, 5"
.LC2 :
        .string "8, 9, 21"
.LC3 :
        .string "22"
.LC4 :
        .string "default"
```

Compilons cet exemple en un fichier objet et ouvrons-le dans IDA. Voici la table des sauts:

Listing 1.158: jumptable in IDA

```
.rodata :0000000000000064                AREA .rodata, DATA, READONLY
.rodata :0000000000000064                ; ORG 0x64
.rodata :0000000000000064 $d             DCB    9    ; case 1
.rodata :0000000000000065                DCB    9    ; case 2
.rodata :0000000000000066                DCB    6    ; case 3
.rodata :0000000000000067                DCB    6    ; case 4
.rodata :0000000000000068                DCB    6    ; case 5
.rodata :0000000000000069                DCB    6    ; case 6
.rodata :000000000000006A                DCB    9    ; case 7
.rodata :000000000000006B                DCB    3    ; case 8
.rodata :000000000000006C                DCB    3    ; case 9
.rodata :000000000000006D                DCB    9    ; case 10
.rodata :000000000000006E                DCB 0xF7    ; case 11
```

```
.rodata :000000000000006F           DCB 0xF7    ; case 12
.rodata :0000000000000070           DCB 0xF7    ; case 13
.rodata :0000000000000071           DCB 0xF7    ; case 14
.rodata :0000000000000072           DCB 0xF7    ; case 15
.rodata :0000000000000073           DCB 0xF7    ; case 16
.rodata :0000000000000074           DCB 0xF7    ; case 17
.rodata :0000000000000075           DCB 0xF7    ; case 18
.rodata :0000000000000076           DCB 0xF7    ; case 19
.rodata :0000000000000077           DCB    3    ; case 20
.rodata :0000000000000078           DCB    3    ; case 21
.rodata :0000000000000079           DCB    0    ; case 22
.rodata :000000000000007B ; .rodata ends
```

Donc dans le cas de 1, 9 est multiplié par 4 et ajouté à l'adresse du label `Lrtx4`.

Dans le cas de 22, 0 est multiplié par 4, ce qui donne 0.

Juste après le label `Lrtx4` se trouve le label L7, où se trouve le code qui affiche « 22 ».

Il n'y a pas de table des sauts dans le segment de code, elle est allouée dans la section .rodata (il n'y a pas de raison de l'allouer dans le segment de code).

Il y a aussi des octets négatifs (0xF7), ils sont utilisés pour sauter en arrière dans le code qui affiche la chaîne « default » (en `.L2`).

1.15.4 Fall-through

Un autre usage très répandu de l'opérateur `switch()` est ce qu'on appelle un « fallthrough » (passer à travers). Voici un exemple simple[99] :

```
1  bool is_whitespace(char c) {
2          switch (c) {
3                  case ' ' : // fallthrough
4                  case '\t' : // fallthrough
5                  case '\r' : // fallthrough
6                  case '\n' :
7                          return true;
8                  default : // not whitespace
9                          return false;
10         }
11 }
```

Légèrement plus difficile, tiré du noyau Linux[100] :

```
1  char nco1, nco2;
2
3  void f(int if_freq_khz)
4  {
5
6          switch (if_freq_khz) {
7                  default :
8                          printf("IF=%d KHz is not supportted, 3250 assumed\n", if_freq_khz);
9                          /* fallthrough */
10                 case 3250: /* 3.25Mhz */
11                         nco1 = 0x34;
12                         nco2 = 0x00;
13                         break;
14                 case 3500: /* 3.50Mhz */
15                         nco1 = 0x38;
16                         nco2 = 0x00;
17                         break;
18                 case 4000: /* 4.00Mhz */
19                         nco1 = 0x40;
20                         nco2 = 0x00;
21                         break;
22                 case 5000: /* 5.00Mhz */
```

[99] Copié/collé depuis https://github.com/azonalon/prgraas/blob/master/prog1lib/lecture_examples/is_whitespace.c
[100] Copié/collé depuis https://github.com/torvalds/linux/blob/master/drivers/media/dvb-frontends/lgdt3306a.c

```
                        nco1 = 0x50;
                        nco2 = 0x00;
                        break;
                case 5380: /* 5.38Mhz */
                        nco1 = 0x56;
                        nco2 = 0x14;
                        break;
        }
};
```

Listing 1.159: GCC 5.4.0 x86 avec optimisation

```
.LC0 :
        .string "IF=%d KHz is not supportted, 3250 assumed\n"
f :
        sub     esp, 12
        mov     eax, DWORD PTR [esp+16]
        cmp     eax, 4000
        je      .L3
        jg      .L4
        cmp     eax, 3250
        je      .L5
        cmp     eax, 3500
        jne     .L2
        mov     BYTE PTR nco1, 56
        mov     BYTE PTR nco2, 0
        add     esp, 12
        ret
.L4 :
        cmp     eax, 5000
        je      .L7
        cmp     eax, 5380
        jne     .L2
        mov     BYTE PTR nco1, 86
        mov     BYTE PTR nco2, 20
        add     esp, 12
        ret
.L2 :
        sub     esp, 8
        push    eax
        push    OFFSET FLAT :.LC0
        call    printf
        add     esp, 16
.L5 :
        mov     BYTE PTR nco1, 52
        mov     BYTE PTR nco2, 0
        add     esp, 12
        ret
.L3 :
        mov     BYTE PTR nco1, 64
        mov     BYTE PTR nco2, 0
        add     esp, 12
        ret
.L7 :
        mov     BYTE PTR nco1, 80
        mov     BYTE PTR nco2, 0
        add     esp, 12
        ret
```

Nous atteignons le label `.L5` si la fonction a reçue le nombre 3250 en entrée. Mais nous pouvons atteindre ce label d'une autre façon: nous voyons qu'il n'y a pas de saut entre l'appel à `printf()` et le label `.L5`.

Nous comprenons maintenant pourquoi la déclaration *switch()* est parfois une source de bug: un *break* oublié va transformer notre déclaration *switch()* en un *fallthrough*, et plusieurs blocs seront exécutés au lieu d'un seul.

1.15.5 Exercices

Exercice #1

Il est possible de modifier l'exemple en C de 1.15.2 on page 166 de telle sorte que le compilateur produise un code plus concis, mais qui fonctionne toujours pareil.

1.16 Boucles

1.16.1 Exemple simple

x86

Il y a une instruction LOOP spéciale en x86 qui teste le contenu du registre ECX et si il est différent de 0, le décrémente et continue l'exécution au label de l'opérande LOOP. Probablement que cette instruction n'est pas très pratique, et il n'y a aucun compilateur moderne qui la génère automatiquement. Donc, si vous la rencontrez dans du code, il est probable qu'il s'agisse de code assembleur écrit manuellement.

En C/C++ les boucles sont en général construites avec une déclaration `for()`, `while()` ou `do/while()`.

Commençons avec `for()`.

Cette déclaration définit l'initialisation de la boucle (met le compteur à sa valeur initiale), la condition de boucle (est-ce que le compteur est plus grand qu'une limite?), qu'est-ce qui est fait à chaque itération (incrémenter/décrémenter) et bien sûr le corps de la boucle.

```
for (initialisation; condition; à chaque itération)
{
    corps_de_la_boucle;
}
```

Le code généré consiste également en quatre parties.

Commençons avec un exemple simple:

```
#include <stdio.h>

void printing_function(int i)
{
        printf ("f(%d)\n", i);
};

int main()
{
        int i;

        for (i=2; i<10; i++)
                printing_function(i);

        return 0;
};
```

Résultat (MSVC 2010):

Listing 1.160: MSVC 2010

```
_i$ = -4
_main    PROC
    push   ebp
    mov    ebp, esp
    push   ecx
    mov    DWORD PTR _i$[ebp], 2    ; initialiser la boucle
    jmp    SHORT $LN3@main
$LN2@main :
    mov    eax, DWORD PTR _i$[ebp]  ; ici se trouve ce que nous faisons après chaque itération :
    add    eax, 1                   ; ajouter 1 à la valeur de (i)
    mov    DWORD PTR _i$[ebp], eax
```

```
$LN3@main :
    cmp    DWORD PTR _i$[ebp], 10  ; cette condition est testée avant chaque itération
    jge    SHORT $LN1@main         ; si (i) est supérieur ou égal à 10, la boucle se termine
    mov    ecx, DWORD PTR _i$[ebp] ; corps de la boucle : appel de printing_function(i)
    push   ecx
    call   _printing_function
    add    esp, 4
    jmp    SHORT $LN2@main         ; saut au début de la boucle
$LN1@main :                         ; fin de la boucle
    xor    eax, eax
    mov    esp, ebp
    pop    ebp
    ret    0
_main    ENDP
```

Comme nous le voyons, rien de spécial.

GCC 4.4.1 génère presque le même code, avec une différence subtile:

Listing 1.161: GCC 4.4.1

```
main            proc near

var_20          = dword ptr -20h
var_4           = dword ptr -4

                push    ebp
                mov     ebp, esp
                and     esp, 0FFFFFFF0h
                sub     esp, 20h
                mov     [esp+20h+var_4], 2  ; initialiser (i)
                jmp     short loc_8048476

loc_8048465 :
                mov     eax, [esp+20h+var_4]
                mov     [esp+20h+var_20], eax
                call    printing_function
                add     [esp+20h+var_4], 1  ; incrémenter (i)

loc_8048476 :
                cmp     [esp+20h+var_4], 9
                jle     short loc_8048465   ; si i<=9, continuer la boucle
                mov     eax, 0
                leave
                retn
main            endp
```

Maintenant, regardons ce que nous obtenons avec l'optimisation (/0x):

Listing 1.162: avec optimisation MSVC

```
_main    PROC
    push   esi
    mov    esi, 2
$LL3@main :
    push   esi
    call   _printing_function
    inc    esi
    add    esp, 4
    cmp    esi, 10     ; 0000000aH
    jl     SHORT $LL3@main
    xor    eax, eax
    pop    esi
    ret    0
_main    ENDP
```

Ce qui se passe alors, c'est que l'espace pour la variable i n'est plus alloué sur la pile locale, mais utilise un registre individuel pour cela, ESI. Ceci est possible pour ce genre de petites fonctions, où il n'y a pas beaucoup de variables locales.

Il est très important que la fonction f() ne modifie pas la valeur de ESI. Notre compilateur en est sûr ici. Et si le compilateur décide d'utiliser le registre ESI aussi dans la fonction f(), sa valeur devra être sauvegardée lors du prologue de la fonction et restaurée lors de son épilogue, presque comme dans notre listing: notez les PUSH ESI/POP ESI au début et à la fin de la fonction.

Essayons GCC 4.4.1 avec l'optimisation la plus performante (option -03):

Listing 1.163: GCC 4.4.1 avec optimisation

```
main            proc near

var_10          = dword ptr -10h

                push    ebp
                mov     ebp, esp
                and     esp, 0FFFFFFF0h
                sub     esp, 10h
                mov     [esp+10h+var_10], 2
                call    printing_function
                mov     [esp+10h+var_10], 3
                call    printing_function
                mov     [esp+10h+var_10], 4
                call    printing_function
                mov     [esp+10h+var_10], 5
                call    printing_function
                mov     [esp+10h+var_10], 6
                call    printing_function
                mov     [esp+10h+var_10], 7
                call    printing_function
                mov     [esp+10h+var_10], 8
                call    printing_function
                mov     [esp+10h+var_10], 9
                call    printing_function
                xor     eax, eax
                leave
                retn
main            endp
```

Hé, GCC a juste déroulé notre boucle.

Le déroulement de boucle est un avantage lorsqu'il n'y a pas beaucoup d'itérations et que nous pouvons économiser du temps d'exécution en supprimant les instructions de gestion de la boucle. D'un autre côté, le code est étonnement plus gros.

Dérouler des grandes boucles n'est pas recommandé de nos jours, car les grosses fonctions ont une plus grande empreinte sur le cache[101].

Ok, augmentons la valeur maximale de la variable i à 100 et essayons à nouveau. GCC donne:

Listing 1.164: GCC

```
                public main
main            proc near

var_20          = dword ptr -20h

                push    ebp
                mov     ebp, esp
                and     esp, 0FFFFFFF0h
                push    ebx
                mov     ebx, 2    ; i=2
                sub     esp, 1Ch

; aligner le label loc_80484D0 (début du corps de la boucle) sur une limite de 16-octet :
                nop

loc_80484D0 :
; passer (i) comme premier argument à printing_function() :
                mov     [esp+20h+var_20], ebx
```

[101] Un très bon article à ce sujet: [Ulrich Drepper, *What Every Programmer Should Know About Memory*, (2007)][102]. D'autres recommandations sur l'expansion des boucles d'Intel sont ici: [[*Intel® 64 and IA-32 Architectures Optimization Reference Manual*, (2014)]3.4.1.7].

```
                add     ebx, 1    ; i++
                call    printing_function
                cmp     ebx, 64h  ; i==100?
                jnz     short loc_80484D0 ; si non, continuer
                add     esp, 1Ch
                xor     eax, eax  ; renvoyer 0
                pop     ebx
                mov     esp, ebp
                pop     ebp
                retn
main            endp
```

C'est assez similaire à ce que MSVC 2010 génère avec l'optimisation (/0x), avec l'exception que le registre EBX est utilisé pour la variable i.

GCC est sûr que ce registre ne sera pas modifié à l'intérieur de la fonction `f()`, et si il l'était, il serait sauvé dans le prologue de la fonction et restauré dans l'épilogue, tout comme dans la fonction `main()`.

x86: OllyDbg

Compilons notre exemple dans MSVC 2010 avec les options `/Ox` et `/Ob0`, puis chargeons le dans OllyDbg.

Il semble qu'OllyDbg soit capable de détecter des boucles simples et les affiche entre parenthèses, par commodité.

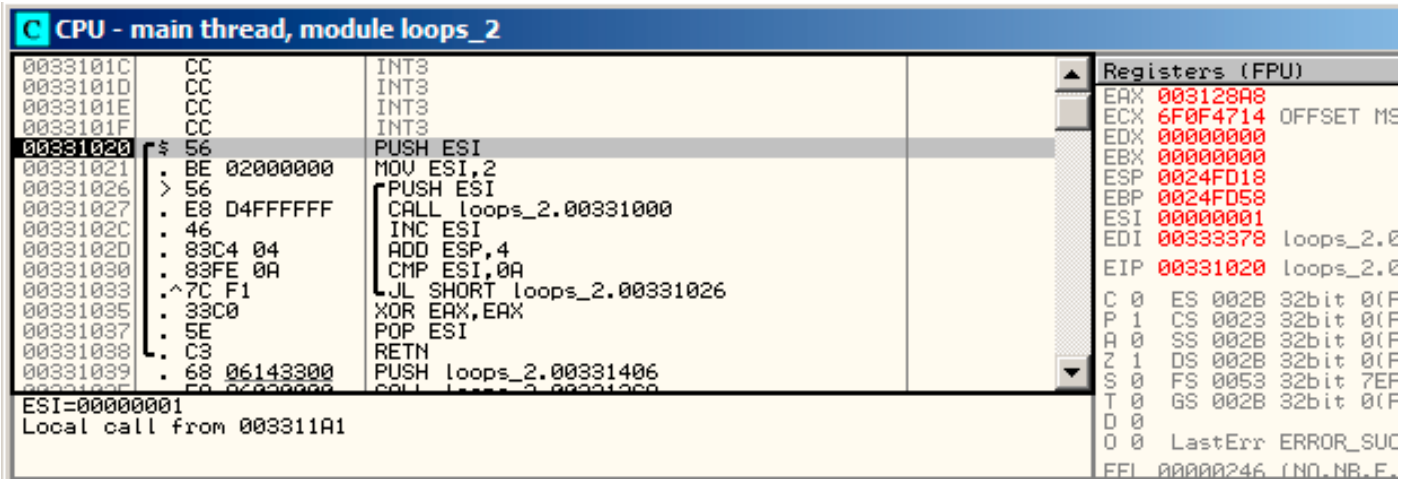


Fig. 1.54: OllyDbg : début de `main()`

En traçant (F8 — enjamber) nous voyons ESI s'incrémenter. Ici, par exemple, $ESI = i = 6$:

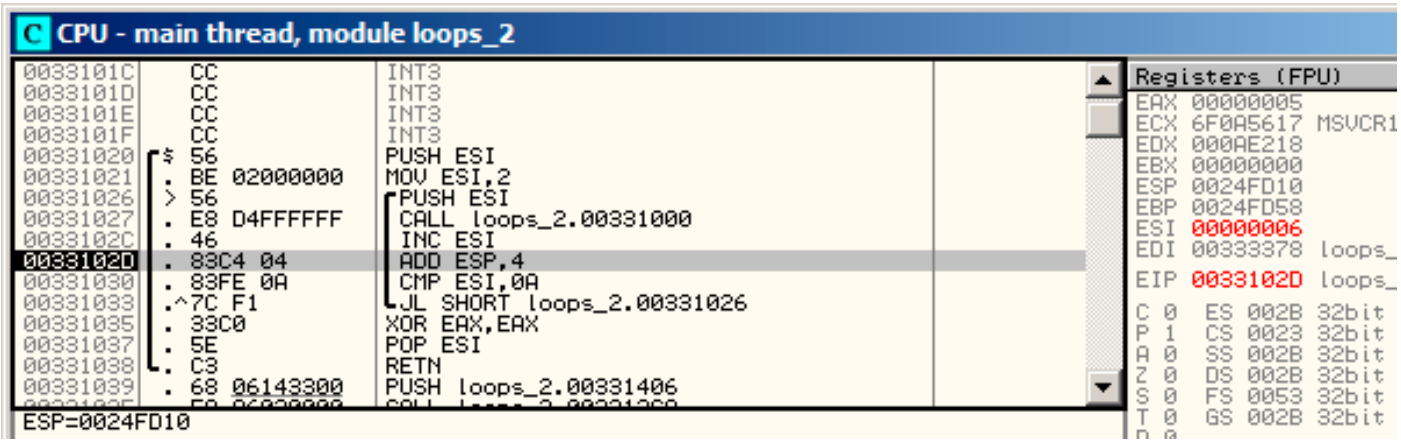


Fig. 1.55: OllyDbg : le corps de la boucle vient de s'exécuter avec $i = 6$

9 est la dernière valeur de la boucle. C'est pourquoi JL ne s'exécute pas après l'incrémentation, et que le fonction se termine.

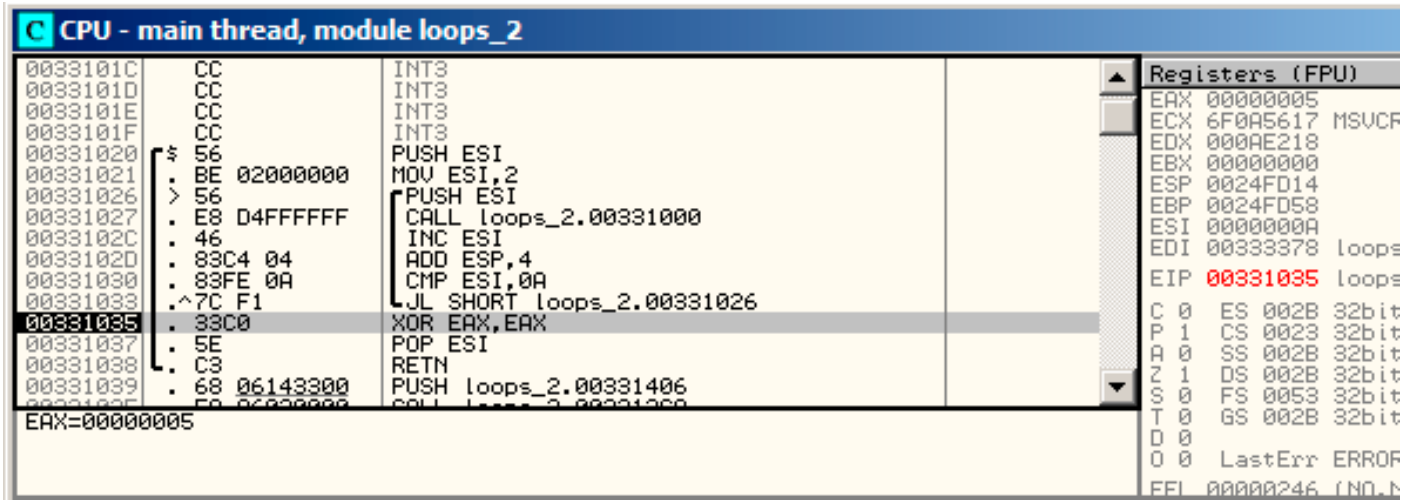


Fig. 1.56: OllyDbg : $ESI = 10$, fin de la boucle

x86: tracer

Comme nous venons de le voir, il n'est pas très commode de tracer manuellement dans le débogueur. C'est pourquoi nous allons essayer tracer.

Nous ouvrons dans IDA l'exemple compilé, trouvons l'adresse de l'instruction PUSH ESI (qui passe le seul argument à f()), qui est 0x401026 dans ce cas et nous lançons le tracer :

```
tracer.exe -l :loops_2.exe bpx=loops_2.exe!0x00401026
```

BPX met juste un point d'arrêt à l'adresse et tracer va alors afficher l'état des registres.

Voici ce que l'on voit dans tracer.log :

```
PID=12884|New process loops_2.exe
(0) loops_2.exe!0x401026
EAX=0x00a328c8 EBX=0x00000000 ECX=0x6f0f4714 EDX=0x00000000
ESI=0x00000002 EDI=0x00333378 EBP=0x0024fbfc ESP=0x0024fbb8
EIP=0x00331026
FLAGS=PF ZF IF
(0) loops_2.exe!0x401026
EAX=0x00000005 EBX=0x00000000 ECX=0x6f0a5617 EDX=0x000ee188
ESI=0x00000003 EDI=0x00333378 EBP=0x0024fbfc ESP=0x0024fbb8
EIP=0x00331026
FLAGS=CF PF AF SF IF
(0) loops_2.exe!0x401026
EAX=0x00000005 EBX=0x00000000 ECX=0x6f0a5617 EDX=0x000ee188
ESI=0x00000004 EDI=0x00333378 EBP=0x0024fbfc ESP=0x0024fbb8
EIP=0x00331026
FLAGS=CF PF AF SF IF
(0) loops_2.exe!0x401026
EAX=0x00000005 EBX=0x00000000 ECX=0x6f0a5617 EDX=0x000ee188
ESI=0x00000005 EDI=0x00333378 EBP=0x0024fbfc ESP=0x0024fbb8
EIP=0x00331026
FLAGS=CF AF SF IF
(0) loops_2.exe!0x401026
EAX=0x00000005 EBX=0x00000000 ECX=0x6f0a5617 EDX=0x000ee188
ESI=0x00000006 EDI=0x00333378 EBP=0x0024fbfc ESP=0x0024fbb8
EIP=0x00331026
FLAGS=CF PF AF SF IF
(0) loops_2.exe!0x401026
EAX=0x00000005 EBX=0x00000000 ECX=0x6f0a5617 EDX=0x000ee188
ESI=0x00000007 EDI=0x00333378 EBP=0x0024fbfc ESP=0x0024fbb8
EIP=0x00331026
FLAGS=CF AF SF IF
(0) loops_2.exe!0x401026
EAX=0x00000005 EBX=0x00000000 ECX=0x6f0a5617 EDX=0x000ee188
ESI=0x00000008 EDI=0x00333378 EBP=0x0024fbfc ESP=0x0024fbb8
```

```
EIP=0x00331026
FLAGS=CF AF SF IF
(0) loops_2.exe!0x401026
EAX=0x00000005 EBX=0x00000000 ECX=0x6f0a5617 EDX=0x000ee188
ESI=0x00000009 EDI=0x00333378 EBP=0x0024fbfc ESP=0x0024fbb8
EIP=0x00331026
FLAGS=CF PF AF SF IF
PID=12884|Process loops_2.exe exited. ExitCode=0 (0x0)
```

Nous voyons comment la valeur du registre ESI change de 2 à 9.

Encore plus que ça, tracer peut collecter les valeurs des registres pour toutes les adresses dans la fonction. C'est appelé *trace* ici. Chaque instruction est tracée, toutes les valeurs intéressantes des registres sont enregistrées.

Ensuite, un script IDA .idc est généré, qui ajoute des commentaires. Donc, dans IDA, nous avons appris que l'adresse de la fonction `main()` est `0x00401020` et nous lançons:

```
tracer.exe -l :loops_2.exe bpf=loops_2.exe!0x00401020,trace :cc
```

BPF signifie mettre un point d'arrêt sur la fonction.

Comme résultat, nous obtenons les scripts `loops_2.exe.idc` et `loops_2.exe_clear.idc`.

Nous chargeons loops_2.exe.idc dans IDA et voyons:

```
.text:00401020
.text:00401020 ; =============== S U B R O U T I N E =======================================
.text:00401020
.text:00401020
.text:00401020 ; int __cdecl main(int argc, const char **argv, const char **envp)
.text:00401020 _main           proc near               ; CODE XREF: ___tmainCRTStartup+11D↓p
.text:00401020
.text:00401020 argc            = dword ptr  4
.text:00401020 argv            = dword ptr  8
.text:00401020 envp            = dword ptr  0Ch
.text:00401020
.text:00401020                 push    esi             ; ESI=1
.text:00401021                 mov     esi, 2
.text:00401026
.text:00401026 loc_401026:                             ; CODE XREF: _main+13↓j
.text:00401026                 push    esi             ; ESI=2..9
.text:00401027                 call    sub_401000      ; tracing nested maximum level (1) reached,
.text:0040102C                 inc     esi             ; ESI=2..9
.text:0040102D                 add     esp, 4          ; ESP=0x38fcbc
.text:00401030                 cmp     esi, 0Ah        ; ESI=3..0xa
.text:00401033                 jl      short loc_401026 ; SF=false,true OF=false
.text:00401035                 xor     eax, eax
.text:00401037                 pop     esi
.text:00401038                 retn                    ; EAX=0
.text:00401038 _main           endp
```

Fig. 1.57: IDA avec le script .idc chargé

Nous voyons que ESI varie de 2 à 9 au début du corps de boucle, mais de 3 à 0xA (10) après l'incrément. Nous voyons aussi que main() se termine avec 0 dans EAX.

tracer génère également loops_2.exe.txt, qui contient des informations sur le nombre de fois qu'une instruction a été exécutée et les valeurs du registre:

Listing 1.165: loops_2.exe.txt

```
0x401020 (.text+0x20), e=       1 [PUSH ESI] ESI=1
0x401021 (.text+0x21), e=       1 [MOV ESI, 2]
0x401026 (.text+0x26), e=       8 [PUSH ESI] ESI=2..9
0x401027 (.text+0x27), e=       8 [CALL 8D1000h] tracing nested maximum level (1) reached, ↙
    ↳ skipping this CALL 8D1000h=0x8d1000
0x40102c (.text+0x2c), e=       8 [INC ESI] ESI=2..9
0x40102d (.text+0x2d), e=       8 [ADD ESP, 4] ESP=0x38fcbc
0x401030 (.text+0x30), e=       8 [CMP ESI, 0Ah] ESI=3..0xa
0x401033 (.text+0x33), e=       8 [JL 8D1026h] SF=false,true OF=false
0x401035 (.text+0x35), e=       1 [XOR EAX, EAX]
0x401037 (.text+0x37), e=       1 [POP ESI]
0x401038 (.text+0x38), e=       1 [RETN] EAX=0
```

Nous pouvons utiliser grep ici.

ARM

sans optimisation Keil 6/2013 (Mode ARM)

```
main
        STMFD   SP!, {R4,LR}
        MOV     R4, #2
        B       loc_368
loc_35C  ; CODE XREF : main+1C
        MOV     R0, R4
        BL      printing_function
        ADD     R4, R4, #1

loc_368  ; CODE XREF : main+8
```

```
        CMP     R4, #0xA
        BLT     loc_35C
        MOV     R0, #0
        LDMFD   SP!, {R4,PC}
```

Le compteur de boucle i est stocké dans le registre R4. L'instruction `MOV R4, #2` initialise i. Les instructions `MOV R0, R4` et `BL printing_function` composent le corps de la boucle, la première instruction préparant l'argument pour la fonction `f()` et la seconde l'appelant. L'instruction `ADD R4, R4, #1` ajoute 1 à la variable i à chaque itération. `CMP R4, #0xA` compare i avec `0xA` (10). L'instruction suivante, `BLT` (*Branch Less Than*) saute si i est inférieur à 10. Autrement, 0 est écrit dans `R0` (puisque notre fonction renvoie 0) et l'exécution de la fonction se termine.

avec optimisation Keil 6/2013 (Mode Thumb)

```
_main
                PUSH    {R4,LR}
                MOVS    R4, #2

loc_132                                 ; CODE XREF : _main+E
                MOVS    R0, R4
                BL      printing_function
                ADDS    R4, R4, #1
                CMP     R4, #0xA
                BLT     loc_132
                MOVS    R0, #0
                POP     {R4,PC}
```

Pratiquement la même chose.

avec optimisation Xcode 4.6.3 (LLVM) (Mode Thumb-2)

```
_main
        PUSH            {R4,R7,LR}
        MOVW            R4, #0x1124 ; "%d\n"
        MOVS            R1, #2
        MOVT.W          R4, #0
        ADD             R7, SP, #4
        ADD             R4, PC
        MOV             R0, R4
        BLX             _printf
        MOV             R0, R4
        MOVS            R1, #3
        BLX             _printf
        MOV             R0, R4
        MOVS            R1, #4
        BLX             _printf
        MOV             R0, R4
        MOVS            R1, #5
        BLX             _printf
        MOV             R0, R4
        MOVS            R1, #6
        BLX             _printf
        MOV             R0, R4
        MOVS            R1, #7
        BLX             _printf
        MOV             R0, R4
        MOVS            R1, #8
        BLX             _printf
        MOV             R0, R4
        MOVS            R1, #9
        BLX             _printf
        MOVS            R0, #0
        POP             {R4,R7,PC}
```

En fait, il y avait ceci dans ma fonction `f()` :

```
void printing_function(int i)
{
    printf ("%d\n", i);
};
```

Donc, non seulement LLVM *déroule* la boucle, mais aussi *inline* ma fonction très simple et insère son corps 8 fois au lieu de l'appeler.

Ceci est possible lorsque la fonction est très simple (comme la mienne) et lorsqu'elle n'est pas trop appelée (comme ici).

ARM64: GCC 4.9.1 avec optimisation

Listing 1.166: GCC 4.9.1 avec optimisation

```
printing_function :
; préparer le second argument de printf() :
        mov     w1, w0
; charger l'adresse de la chaîne "f(%d)\n"
        adrp    x0, .LC0
        add     x0, x0,  :lo12 :.LC0
; seulement sauter ici au lieu de sauter avec lien et retour :
        b       printf
main :
; sauver FP et LR dans la pile locale :
        stp     x29, x30, [sp, -32]!
; préparer une structure de pile :
        add     x29, sp, 0
; sauver le contenu du registre X19 dans la pile locale :
        str     x19, [sp,16]
; nous allons utiliser le registre W19 comme compteur.
; lui assigner une valeur initiale de 2:
        mov     w19, 2
.L3 :
; préparer le premier argument de printing_function() :
        mov     w0, w19
; incrémenter le registre compteur.
        add     w19, w19, 1
; ici W0 contient toujours la valeur du compteur avant incrémentation.
        bl      printing_function
; est-ce terminé ?
        cmp     w19, 10
; non, sauter au début du corps de boucle :
        bne     .L3
; renvoyer 0
        mov     w0, 0
; restaurer le contenu du registre X19 :
        ldr     x19, [sp,16]
; restaurer les valeurs de FP et LR :
        ldp     x29, x30, [sp], 32
        ret
.LC0 :
        .string "f(%d)\n"
```

ARM64: GCC 4.9.1 sans optimisation

Listing 1.167: GCC 4.9.1 -fno-inline sans optimisation

```
printing_function :
; préparer le second argument de printf() :
        mov     w1, w0
; charger l'adresse de la chaîne "f(%d)\n"
        adrp    x0, .LC0
        add     x0, x0,  :lo12 :.LC0
; seulement sauter ici au lieu de sauter avec lien et retour :
```

```
        b       printf
main :
; sauver FP et LR dans la pile locale :
        stp     x29, x30, [sp, -32]!
; préparer une structure de pile :
        add     x29, sp, 0
; sauver le contenu du registre X19 dans la pile locale :
        str     x19, [sp,16]
; nous allons utiliser le registre W19 comme compteur.
; lui assigner une valeur initiale de 2:
        mov     w19, 2
.L3 :
; préparer le premier argument de printing_function() :
        mov     w0, w19
; incrémenter le registre compteur.
        add     w19, w19, 1
; ici W0 contient toujours la valeur du compteur avant incrémentation.
        bl      printing_function
; est-ce terminé?
        cmp     w19, 10
; non, sauter au début du corps de boucle :
        bne     .L3
; renvoyer 0
        mov     w0, 0
; restaurer le contenu du registre X19 :
        ldr     x19, [sp,16]
; restaurer les valeurs de FP et LR :
        ldp     x29, x30, [sp], 32
        ret
.LC0 :
        .string "f(%d)\n"
```

MIPS

Listing 1.168: GCC 4.4.5 sans optimisation (IDA)

```
main :

; IDA ne connaît pas le nom des variables dans la pile locale
; Nous pouvons leurs en donner un manuellement :
i               = -0x10
saved_FP        = -8
saved_RA        = -4

; prologue de la fonction :
                addiu   $sp, -0x28
                sw      $ra, 0x28+saved_RA($sp)
                sw      $fp, 0x28+saved_FP($sp)
                move    $fp, $sp
; initialiser le compteur à 2 et stocker cette valeur dans la pile locale
                li      $v0, 2
                sw      $v0, 0x28+i($fp)
; pseudo-instruction. "BEQ $ZERO, $ZERO, loc_9C" c'est en fait :
                b       loc_9C
                or      $at, $zero ; slot de délai de branchement, NOP

loc_80 :                                # CODE XREF : main+48
; charger la valeur du compteur depuis la pile locale et appeler printing_function() :
                lw      $a0, 0x28+i($fp)
                jal     printing_function
                or      $at, $zero ; slot de délai de branchement, NOP
; charger le compteur, l'incrémenter, et le stocker de nouveau :
                lw      $v0, 0x28+i($fp)
                or      $at, $zero ; NOP
                addiu   $v0, 1
                sw      $v0, 0x28+i($fp)

loc_9C :                                # CODE XREF : main+18
```

```
; tester le compteur, est-ce 10?
                lw      $v0, 0x28+i($fp)
                or      $at, $zero ; NOP
                slti    $v0, 0xA
; si il est inférieur à 10, sauter en loc_80 (début du corps de la boucle) :
                bnez    $v0, loc_80
                or      $at, $zero ; slot de délai de branchement, NOP
; fin, renvoyer 0:
                move    $v0, $zero
; épilogue de la fonction :
                move    $sp, $fp
                lw      $ra, 0x28+saved_RA($sp)
                lw      $fp, 0x28+saved_FP($sp)
                addiu   $sp, 0x28
                jr      $ra
                or      $at, $zero ; slot de délai de branchement, NOP
```

L'instruction qui est nouvelle pour nous est B. C'est la pseudo instruction (BEQ).

Encore une chose

Dans le code généré, nous pouvons voir: après avoir initialisé i, le corps de la boucle n'est pas exécuté, car la condition sur i est d'abord vérifiée, et c'est seulement après cela que le corps de la boucle peut être exécuté. Et cela est correct.

Ceci car si la condition de boucle n'est pas remplie au début, le corps de la boucle ne doit pas être exécuté. Ceci est possible dans le cas suivant:

```
for (i=0; i<nombre_total_d_element_à_traiter; i++)
    corps_de_la_boucle;
```

Si *nombre_total_d_element_à_traiter* est 0, le corps de la boucle ne sera pas exécuté du tout.

C'est pourquoi la condition est testée avant l'exécution.

Toutefois, un compilateur qui optimise pourrait échanger le corps de la boucle et la condition, si il est certain que la situation que nous venons de décrire n'est pas possible (comme dans le cas de notre exemple simple, et en utilisant des compilateurs comme Keil, Xcode (LLVM) et MSVC avec le flag d'optimisation.

1.16.2 Routine de copie de blocs de mémoire

Les routines réelles de copie de mémoire copient 4 ou 8 octets à chaque itération, utilisent SIMD[103], la vectorisation, etc. Mais dans un but didactique, cet exemple est le plus simple possible.

```
#include <stdio.h>

void my_memcpy (unsigned char* dst, unsigned char* src, size_t cnt)
{
        size_t i;
        for (i=0; i<cnt; i++)
                dst[i]=src[i];
};
```

Implémentation simple

Listing 1.169: GCC 4.9 x64 optimisé pour la taille (-Os)

```
my_memcpy :
; RDI = adresse de destination
; RSI = adresse source
; RDX = taille de bloc

; initialiser le compteur (i) à 0
```

[103] Single Instruction, Multiple Data

```
        xor     eax, eax
.L2 :
; tous les octets sont-ils copiés ? alors sortir :
        cmp     rax, rdx
        je      .L5
; charger l'octet en RSI+i :
        mov     cl, BYTE PTR [rsi+rax]
; stocker l'octet en RDI+i :
        mov     BYTE PTR [rdi+rax], cl
        inc     rax ; i++
        jmp     .L2
.L5 :
        ret
```

Listing 1.170: GCC 4.9 ARM64 optimisé pour la taille (-Os)

```
my_memcpy :
; X0 = adresse de destination
; X1 = adresse source
; X2 = taille de bloc

; initialiser le compteur (i) à 0
        mov     x3, 0
.L2 :
; tous les octets sont-ils copiés ? alors sortir :
        cmp     x3, x2
        beq     .L5
; charger l'octet en X1+i :
        ldrb    w4, [x1,x3]
; stocker l'octet en X0+i :
        strb    w4, [x0,x3]
        add     x3, x3, 1 ; i++
        b       .L2
.L5 :
        ret
```

Listing 1.171: avec optimisation Keil 6/2013 (Mode Thumb)

```
my_memcpy PROC
; R0 = adresse de destination
; R1 = adresse source
; R2 = taille de bloc

        PUSH    {r4,lr}
; initialiser le compteur (i) à 0
        MOVS    r3,#0
; la condition est testée à la fin de la fonction, donc y sauter :
        B       |L0.12|
|L0.6|
; charger l'octet en R1+i :
        LDRB    r4,[r1,r3]
; stocker l'octet en R0+i :
        STRB    r4,[r0,r3]
; i++
        ADDS    r3,r3,#1
|L0.12|
; i<taille ?
        CMP     r3,r2
; sauter au début de la boucle si c'est le cas :
        BCC     |L0.6|
        POP     {r4,pc}
        ENDP
```

ARM en mode ARM

Keil en mode ARM tire pleinement avantage des suffixes conditionnels:

Listing 1.172: avec optimisation Keil 6/2013 (Mode ARM)

```
my_memcpy PROC
; R0 = adresse de destination
; R1 = adresse source
; R2 = taille de bloc

; initialiser le compteur (i) à 0
        MOV      r3,#0
|L0.4|
; tous les octets sont-ils copiés?
        CMP      r3,r2
; le bloc suivant est exécuté seulement si la condition less than est remplie,
; i.e., if R2<R3 ou i<taille.
; charger l'octet en R1+i :
        LDRBCC   r12,[r1,r3]
; stocker l'octet en R0+i :
        STRBCC   r12,[r0,r3]
; i++
        ADDCC    r3,r3,#1
; la dernière instruction du bloc conditionnel.
; sauter au début de la boucle si i<taille
; ne rien faire autrement (i.e., si i>=taille)
        BCC      |L0.4|
; retourner
        BX       lr
        ENDP
```

C'est pourquoi il y a seulement une instruction de branchement au lieu de 2.

MIPS

Listing 1.173: GCC 4.4.5 optimisé pour la taille (-Os) (IDA)

```
my_memcpy :
; sauter à la partie test de la boucle :
                b       loc_14
; initialiser le compteur (i) à 0
; il se trouvera toujours dans $v0 :
                move    $v0, $zero ; slot de délai de branchement

loc_8 :                                 # CODE XREF : my_memcpy+1C
; charger l'octet non-signé à l'adresse $t0 dans $v1 :
                lbu     $v1, 0($t0)
; incrémenter le compteur (i) :
                addiu   $v0, 1
; stocker l'octet en $a3
                sb      $v1, 0($a3)

loc_14 :                                # CODE XREF : my_memcpy
; tester si le compteur (i) dans $v0 est toujours inférieur au 3ème argument de la fonction ("↙
    ↳ cnt" dans $a2) :
                sltu    $v1, $v0, $a2
; former l'adresse de l'octet dans le bloc source :
                addu    $t0, $a1, $v0
; $t0 = $a1+$v0 = src+i
; sauter au corps de la boucle  si le compteur est toujours inférieur à "cnt":
                bnez    $v1, loc_8
; former l'adresse de l'octet dans le bloc de destination ($a3 = $a0+$v0 = dst+i) :
                addu    $a3, $a0, $v0 ; slot de délai de branchement
; terminer si BNEZ n'a pas exécuté de saut :
                jr      $ra
                or      $at, $zero ; slot de délai de branchement, NOP
```

Nous avons ici deux nouvelles instructions: LBU (« Load Byte Unsigned » charger un octet non signé) et SB (« Store Byte » stocker un octet).

Tout comme en ARM, tous les registres MIPS ont une taille de 32-bit, il n'y en a pas d'un octet de large comme en x86.

Donc, lorsque l'on travaille avec des octets seuls, nous devons utiliser un registre de 32-bit pour chacun d'entre eux.

LBU charge un octet et met les autres bits à zéro (« Unsigned »).

En revanche, l'instruction LB (« Load Byte ») étend le signe de l'octet chargé sur 32-bit.

SB écrit simplement un octet depuis les 8 bits de poids faible d'un registre dans la mémoire.

Vectorisation

GCC avec optimisation peut faire beaucoup mieux avec cet exemple: 1.29.1 on page 415.

1.16.3 Vérification de condition

Il est important de garder à l'esprit que dans une boucle *for()*, la condition est vérifiée préalablement à l'itération du corps de la boucle et non pas après. Cela étant il est souvent plus pratique pour le compilateur de placer les instructions qui effectuent le test après le corps de la boucle. Il arrive aussi qu'il rajoute des vérifications au début du corps de la boucle.

Par exemple:

```
#include <stdio.h>

void f(int start, int finish)
{
        for (; start<finish; start++)
                printf ("%d\n", start);
};
```

GCC 5.4.0 x64 en mode optimisé:

```
f :
; check condition (1) :
        cmp     edi, esi
        jge     .L9
        push    rbp
        push    rbx
        mov     ebp, esi
        mov     ebx, edi
        sub     rsp, 8
.L5 :
        mov     edx, ebx
        xor     eax, eax
        mov     esi, OFFSET FLAT :.LC0 ; '%d\n"
        mov     edi, 1
        add     ebx, 1
        call    __printf_chk
; check condition (2) :
        cmp     ebp, ebx
        jne     .L5
        add     rsp, 8
        pop     rbx
        pop     rbp
.L9 :
        rep ret
```

Nous constatons la présence de deux vérifications.

Le code décompilé produit par Hex-Rays (dans sa version 2.2.0) est celui-ci:

```
void __cdecl f(unsigned int start, unsigned int finish)
{
  unsigned int v2; // ebx@2
  __int64 v3; // rdx@3

  if ( (signed int)start < (signed int)finish )
  {
```

```
    v2 = start;
    do
    {
      v3 = v2++;
      _printf_chk(1LL, "%d\n", v3);
    }
    while ( finish != v2 );
  }
}
```

Dans le cas présent, il ne fait aucun doute que la structure *do/while()* peut être remplacée par une construction *for()*, et que le premier contrôle peut être supprimé.

1.16.4 Conclusion

Squelette grossier d'une boucle de 2 à 9 inclus:

Listing 1.174: x86

```
    mov [counter], 2 ; initialisation
    jmp check
body :
    ; corps de la boucle
    ; faire quelque chose ici
    ; utiliser la variable compteur dans la pile locale
    add [counter], 1 ; incrémenter
check :
    cmp [counter], 9
    jle body
```

L'opération d'incrémentation peut être représentée par 3 instructions dans du code non optimisé:

Listing 1.175: x86

```
    MOV [counter], 2 ; initialisation
    JMP check
body :
    ; corps de la boucle
    ; faire quelque chose ici
    ; utiliser la variable compteur dans la pile locale
    MOV REG, [counter] ; incrémenter
    INC REG
    MOV [counter], REG
check :
    CMP [counter], 9
    JLE body
```

Si le corps de la boucle est court, un registre entier peut être dédié à la variable compteur:

Listing 1.176: x86

```
    MOV EBX, 2 ; initialisation
    JMP check
body :
    ; corps de la boucle
    ; faire quelque chose ici
    ; utiliser le compteur dans EBX, mais ne pas le modifier!
    INC EBX ; incrémenter
check :
    CMP EBX, 9
    JLE body
```

Certaines parties de la boucle peuvent être générées dans un ordre différent par le compilateur:

Listing 1.177: x86

```
    MOV [counter], 2 ; initialisation
    JMP label_check
label_increment :
```

```
    ADD [counter], 1 ; incrémenter
label_check :
    CMP [counter], 10
    JGE exit
    ; corps de la boucle
    ; faire quelque cose ici
    ; utiliser la variable compteur dans la pile locale
    JMP label_increment
exit :
```

En général, la condition est testée *avant* le corps de la boucle, mais le compilateur peut la réarranger afin que la condition soit testée *après* le corps de la boucle.

Cela est fait lorsque le compilateur est certain que la condition est toujours *vraie* à la première itération, donc que le corps de la boucle doit être exécuté au moins une fois:

Listing 1.178: x86

```
    MOV REG, 2 ; initialisation
body :
    ; corps de la boucle
    ; faire quelque chose ici
    ; utiliser le compteur dans REG, mais ne pas le modifier!
    INC REG ; incrémenter
    CMP REG, 10
    JL body
```

En utilisant l'instruction LOOP. Ceci est rare, les compilateurs ne l'utilisent pas. Lorsque vous la voyez, c'est le signe que le morceau de code a été écrit à la main:

Listing 1.179: x86

```
    ; compter de 10 à 1
    MOV ECX, 10
body :
    ; corps de la boucle
    ; faire quelque chose ici
    ; utiliser le compteur dans ECX, mais ne pas le modifier!
    LOOP body
```

ARM.

Le registre R4 est dédié à la variable compteur dans cet exemple:

Listing 1.180: ARM

```
    MOV R4, 2 ; initialisation
    B check
body :
    ; corps de la boucle
    ; faire quelque chose ici
    ; utiliser le compteur dans R4, mais ne pas le modifier!
    ADD R4,R4, #1 ; incrémenter
check :
    CMP R4, #10
    BLT body
```

1.16.5 Exercices

- http://challenges.re/54
- http://challenges.re/55
- http://challenges.re/56
- http://challenges.re/57

1.17 Plus d'information sur les chaînes

1.17.1 strlen()

Parlons encore une fois des boucles. Souvent, la fonction `strlen()` [104] est implémentée en utilisant une déclaration `while()`. Voici comment cela est fait dans les bibliothèques standards de MSVC:

```
int my_strlen (const char * str)
{
        const char *eos = str;

        while( *eos++ ) ;

        return( eos - str - 1 );
}

int main()
{
        // test
        return my_strlen("hello!");
};
```

x86

MSVC sans optimisation

Compilons:

```
_eos$ = -4                      ; size = 4
_str$ = 8                       ; size = 4
_strlen PROC
    push    ebp
    mov     ebp, esp
    push    ecx
    mov     eax, DWORD PTR _str$[ebp]  ; copier le pointeur sur la chaîne "str"
    mov     DWORD PTR _eos$[ebp], eax  ; le copier dans la variable locale "eos"
$LN2@strlen_ :
    mov     ecx, DWORD PTR _eos$[ebp]  ; ECX=eos

    ; prendre un octet 8-bit depuis l'adresse dans ECX et le copier comme une valeur 32-bit ⤶
    ↳ dans EDX avec extension du signe

    movsx   edx, BYTE PTR [ecx]
    mov     eax, DWORD PTR _eos$[ebp]  ; EAX=eos
    add     eax, 1                     ; incrémenter EAX
    mov     DWORD PTR _eos$[ebp], eax  ; remettre EAX dans "eos"
    test    edx, edx                   ; est-ce que EDX est à zéro?
    je      SHORT $LN1@strlen_         ; oui, alors finir la boucle
    jmp     SHORT $LN2@strlen_         ; continuer la boucle
$LN1@strlen_ :

    ; ici nous calculons la différence entre deux pointeurs

    mov    eax, DWORD PTR _eos$[ebp]
    sub    eax, DWORD PTR _str$[ebp]
    sub    eax, 1                      ; soustraire 1 du résultat et sortir
    mov    esp, ebp
    pop    ebp
    ret    0
_strlen_ ENDP
```

Nous avons ici deux nouvelles instructions: MOVSX et TEST.

[104] compter les caractères d'une chaîne en langage C

La première—MOVSX—prend un octet depuis une adresse en mémoire et stocke la valeur dans un registre 32-bit. MOVSX signifie *MOV with Sign-Extend* (déplacement avec extension de signe). MOVSX met le reste des bits, du 8ème au 31ème, à 1 si l'octet source est *négatif* ou à 0 si il est *positif*.

Et voici pourquoi.

Par défaut, le type *char* est signé dans MSVC et GCC. Si nous avons deux valeurs dont l'une d'elle est un *char* et l'autre un *int*, (*int* est signé aussi), et si la première valeur contient -2 (codé en 0xFE) et que nous copions simplement cet octet dans le conteneur *int*, cela fait 0x000000FE, et ceci, pour le type *int* représente 254, mais pas -2. Dans un entier signé, -2 est codé en 0xFFFFFFFE. Donc, si nous devons transférer 0xFE depuis une variable de type *char* vers une de type *int*, nous devons identifier son signe et l'étendre. C'est ce qu'effectue MOVSX.

Lire à ce propos dans « *Représentations des nombres signés* » section (2.2 on page 456).

Il est difficile de dire si le compilateur doit stocker une variable *char* dans EDX, il pourrait simplement utiliser une partie 8-bit du registre (par exemple DL). Apparemment, l'allocateur de registre fonctionne comme ça.

Ensuite nous voyons TEST EDX, EDX. Vous pouvez en lire plus à propos de l'instruction TEST dans la section concernant les champs de bit (1.22 on page 306). Ici cette instruction teste simplement si la valeur dans EDX est égale à 0.

GCC sans optimisation

Essayons GCC 4.4.1:

```
                public strlen
strlen          proc near

eos             = dword ptr -4
arg_0           = dword ptr  8

                push    ebp
                mov     ebp, esp
                sub     esp, 10h
                mov     eax, [ebp+arg_0]
                mov     [ebp+eos], eax

loc_80483F0 :
                mov     eax, [ebp+eos]
                movzx   eax, byte ptr [eax]
                test    al, al
                setnz   al
                add     [ebp+eos], 1
                test    al, al
                jnz     short loc_80483F0
                mov     edx, [ebp+eos]
                mov     eax, [ebp+arg_0]
                mov     ecx, edx
                sub     ecx, eax
                mov     eax, ecx
                sub     eax, 1
                leave
                retn
strlen          endp
```

Le résultat est presque le même qu'avec MSVC, mais ici nous voyons MOVZX au lieu de MOVSX. MOVZX signifie *MOV with Zero-Extend* (déplacement avec extension à 0). Cette instruction copie une valeur 8-bit ou 16-bit dans un registre 32-bit et met les bits restant à 0. En fait, cette instructions n'est pratique que pour nous permettre de remplacer cette paire d'instructions:
`xor eax, eax / mov al, [...]`.

D'un autre côté, il est évident que le compilateur pourrait produire ce code:
`mov al, byte ptr [eax] / test al, al`—c'est presque le même, toutefois, les bits les plus haut du registre EAX vont contenir des valeurs aléatoires. Mais, admettons que c'est un inconvénient du compilateur—il ne peut pas produire du code plus compréhensible. À strictement parler, le compilateur n'est pas du tout obligé de générer du code compréhensible par les humains.

La nouvelle instruction suivante est SETNZ. Ici, si AL ne contient pas zéro, `test al, al` met le flag ZF à 0, mais SETNZ, si ZF==0 (*NZ* signifie *not zero*, non zéro) met AL à 1. En langage naturel, *si AL n'est pas zéro, sauter en loc_80483F0*. Le compilateur génère du code redondant, mais n'oublions pas qu'il n'est pas en mode optimisation.

MSVC avec optimisation

Maintenant, compilons tout cela avec MSVC 2012, avec le flag d'optimisation (/0x):

Listing 1.181: MSVC 2012 avec optimisation/Ob0

```
_str$ = 8                       ; size = 4
_strlen PROC
        mov     edx, DWORD PTR _str$[esp-4] ; EDX -> pointeur sur la chaîne
        mov     eax, edx                    ; déplacer dans EAX
$LL2@strlen :
        mov     cl, BYTE PTR [eax]          ; CL = *EAX
        inc     eax                         ; EAX++
        test    cl, cl                      ; CL==0?
        jne     SHORT $LL2@strlen           ; non, continuer la boucle
        sub     eax, edx                    ; calculer la différence entre les pointeurs
        dec     eax                         ; décrémenter EAX
        ret     0
_strlen ENDP
```

C'est plus simple maintenant. Inutile de préciser que le compilateur ne peut utiliser les registres aussi efficacement que dans une petite fonction, avec peu de variables locales.

INC/DEC—sont des instructions de incrémentation/décrémentation, en d'autres mots: ajouter ou soustraire 1 d'une/à une variable.

MSVC avec optimisation + OllyDbg

Nous pouvons essayer cet exemple (optimisé) dans OllyDbg. Voici la première itération:

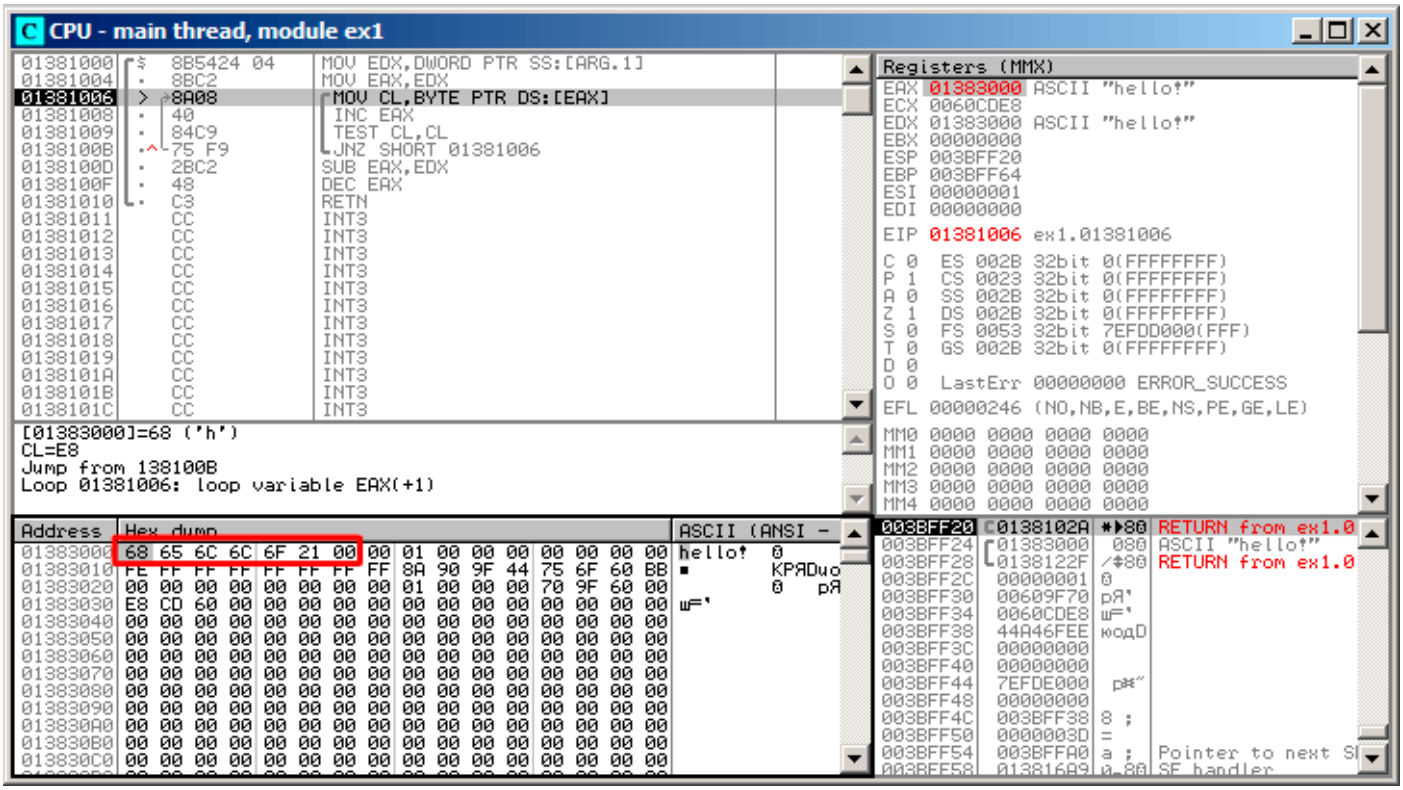


Fig. 1.58: OllyDbg : début de la première itération

Nous voyons qu'OllyDbg a trouvé une boucle et, par facilité, a mis ses instructions entre crochets. En cliquant sur le bouton droit sur EAX, nous pouvons choisir « Follow in Dump » et la fenêtre de la mémoire se déplace jusqu'à la bonne adresse. Ici, nous voyons la chaîne « hello! » en mémoire. Il y a au moins un zéro après cette dernière et ensuite des données aléatoires.

Si OllyDbg voit un registre contenant une adresse valide, qui pointe sur une chaîne, il montre cette chaîne.

Appuyons quelques fois sur F8 (enjamber), pour aller jusqu'au début du corps de la boucle:

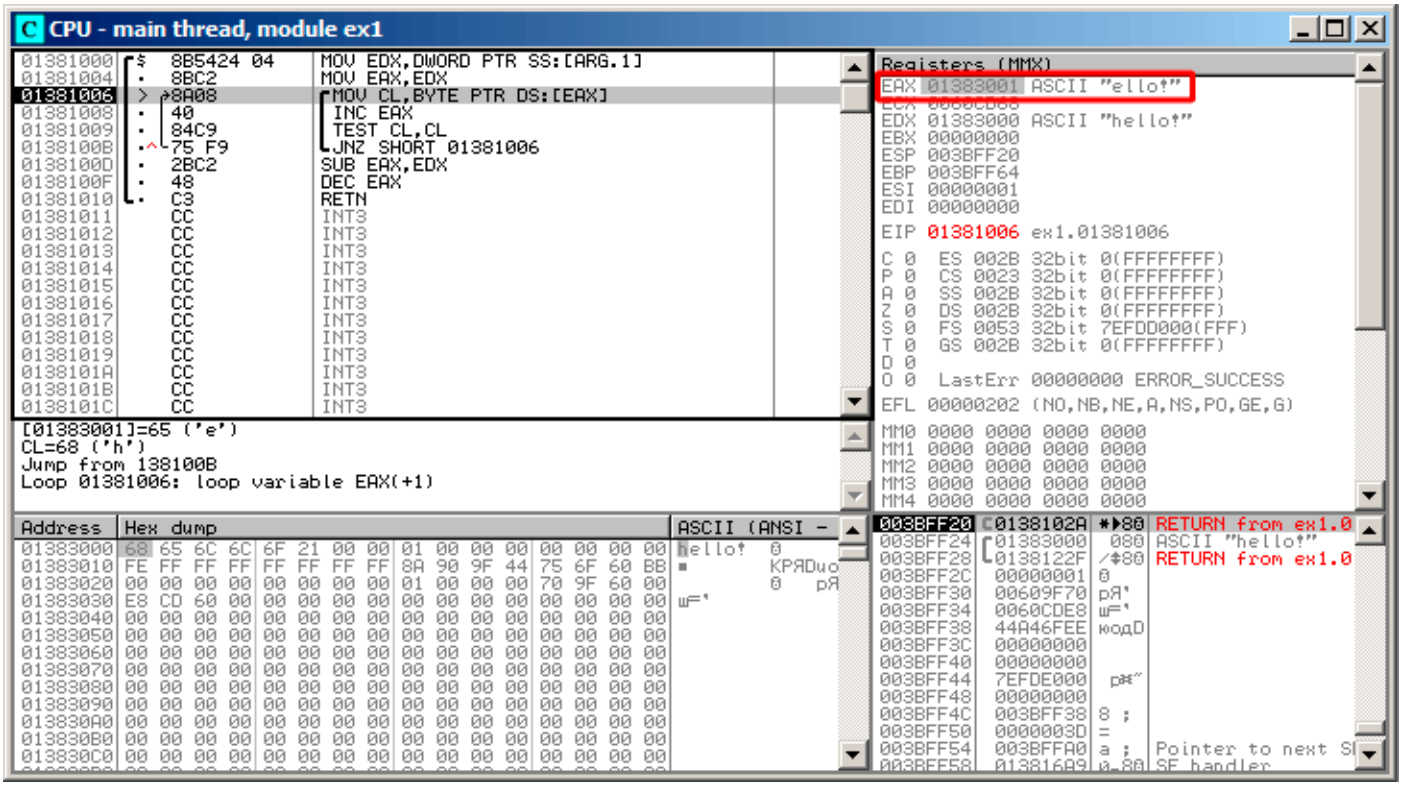


Fig. 1.59: OllyDbg : début de la seconde itération

Nous voyons qu'EAX contient l'adresse du second caractère de la chaîne.

Nous devons appuyons un certain nombre de fois sur F8 afin de sortir de la boucle:

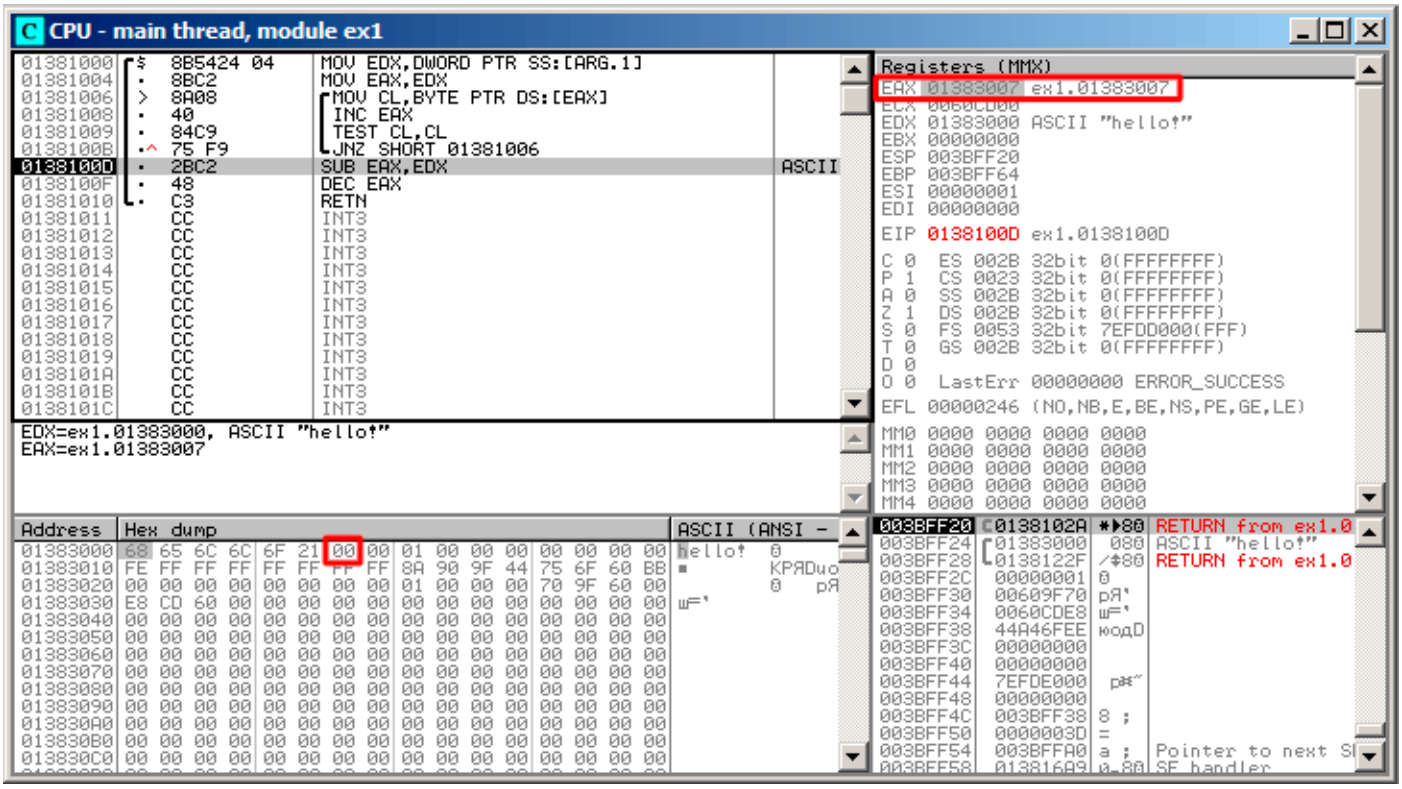


Fig. 1.60: OllyDbg : calcul de la différence entre les pointeurs

Nous voyons qu'EAX contient l'adresse de l'octet à zéro situé juste après la chaîne. Entre temps, EDX n'a pas changé, donc il pointe sur le début de la chaîne.

La différence entre ces deux valeurs est maintenant calculée.

L'instruction SUB vient juste d'être effectuée:

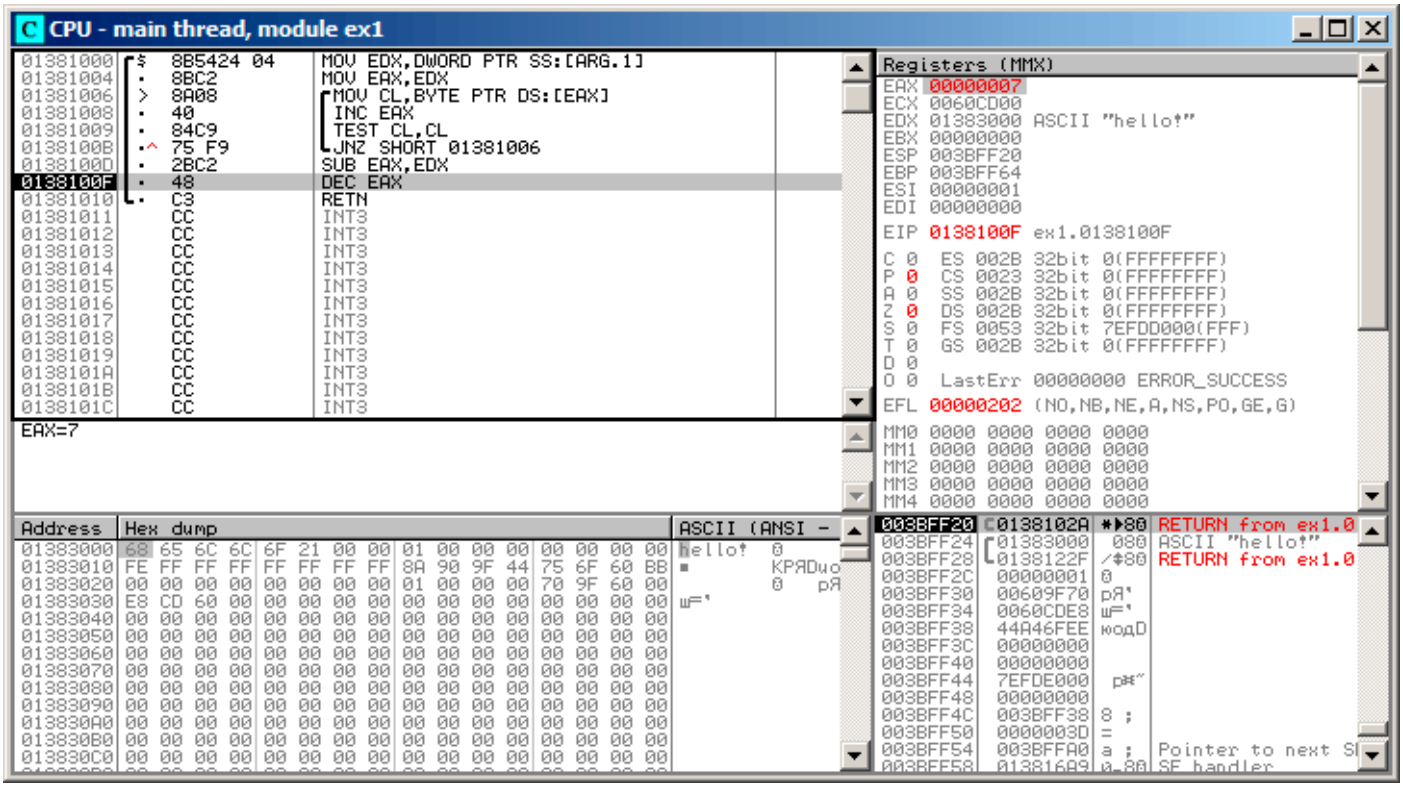


Fig. 1.61: OllyDbg : maintenant décrémenter EAX

La différence entre les deux pointeurs est maintenant dans le registre EAX—7. Effectivement, la longueur de la chaîne « hello! » est 6, mais avec l'octet à zéro inclus—7. Mais `strlen()` doit renvoyer le nombre de caractère non-zéro dans la chaîne. Donc la décrémentation est effectuée et ensuite la fonction sort.

GCC avec optimisation

Regardons ce que génère GCC 4.4.1 avec l'option d'optimisation `-03` :

```
                public strlen
strlen          proc near

arg_0           = dword ptr  8

                push    ebp
                mov     ebp, esp
                mov     ecx, [ebp+arg_0]
                mov     eax, ecx

loc_8048418 :
                movzx   edx, byte ptr [eax]
                add     eax, 1
                test    dl, dl
                jnz     short loc_8048418
                not     ecx
                add     eax, ecx
                pop     ebp
                retn
strlen          endp
```

`mov dl, byte ptr [eax]`. Ici GCC génère presque le même code que MSVC, à l'exception de la présence de `MOVZX`. Toutefois, ici, `MOVZX` pourrait être remplacé par
`mov dl, byte ptr [eax]`.

Peut-être est-il plus simple pour le générateur de code de GCC se se *rappeler* que le registre 32-bit EDX est alloué entièrement pour une variable *char* et il est sûr que les bits en partie haute ne contiennent pas de bruit indéfini.

Après cela, nous voyons une nouvelle instruction—NOT. Cette instruction inverse tout les bits de l'opérande.
Elle peut être vu comme un synonyme de l'instruction XOR ECX, 0ffffffffh. NOT et l'instruction suivante ADD calcule la différence entre les pointeurs et soustrait 1, d'une façon différente. Au début, ECX, où le pointeur sur *str* est stocké, est inversé et 1 en est soustrait.

Voir aussi: « Représentations des nombres signés » (2.2 on page 456).

En d'autres mots, à la fin de la fonction juste après le corps de la boucle, ces opérations sont exécutées:

```
ecx=str;
eax=eos;
ecx=(-ecx)-1;
eax=eax+ecx
return eax
```

... et ceci est effectivement équivalent à:

```
ecx=str;
eax=eos;
eax=eax-ecx;
eax=eax-1;
return eax
```

Pourquoi est-ce que GCC décide que cela est mieux? Difficile à deviner. Mais peut-être que les deux variantes sont également efficaces.

ARM

ARM 32-bit

sans optimisation Xcode 4.6.3 (LLVM) (Mode ARM)

Listing 1.182: sans optimisation Xcode 4.6.3 (LLVM) (Mode ARM)

```
_strlen

eos  = -8
str  = -4

     SUB    SP, SP, #8 ; allouer 8 octets pour les variables locales
     STR    R0, [SP,#8+str]
     LDR    R0, [SP,#8+str]
     STR    R0, [SP,#8+eos]

loc_2CB8 ; CODE XREF : _strlen+28
     LDR    R0, [SP,#8+eos]
     ADD    R1, R0, #1
     STR    R1, [SP,#8+eos]
     LDRSB  R0, [R0]
     CMP    R0, #0
     BEQ    loc_2CD4
     B      loc_2CB8
loc_2CD4 ; CODE XREF : _strlen+24
     LDR    R0, [SP,#8+eos]
     LDR    R1, [SP,#8+str]
     SUB    R0, R0, R1 ; R0=eos-str
     SUB    R0, R0, #1 ; R0=R0-1
     ADD    SP, SP, #8 ; libérer les 8 octets alloués
     BX     LR
```

LLVM sans optimisation génère beaucoup trop de code, toutefois, ici nous pouvons voir comment la fonction travaille avec les variables locales. Il y a seulement deux variables locales dans notre fonction: *eos* et *str*. Dans ce listing, généré par IDA, nous avons renommé manuellement *var_8* et *var_4* en *eos* et *str*.

La première instruction sauve simplement les valeurs d'entrée dans *str* et *eos*.

Le corps de la boucle démarre au label *loc_2CB8*.

Les trois première instructions du corps de la boucle (LDR, ADD, STR) chargent la valeur de *eos* dans R0. Puis la valeur est incrémentée et sauvée dans *eos*, qui se trouve sur la pile.

L'instruction suivante, `LDRSB R0, [R0]` (« Load Register Signed Byte »), charge un octet depuis la mémoire à l'adresse stockée dans RR0 et étend le signe à 32-bit[105]. Ceci est similaire à l'instruction MOVSX en x86.

Le compilateur traite cet octet comme signé, puisque le type *char* est signé selon la norme C. Il a déjà été écrit à propos de cela (1.17.1 on page 201) dans cette section, en relation avec le x86.

Il est à noter qu'il est impossible en ARM d'utiliser séparément la partie 8- ou 16-bit d'un registre 32-bit complet, comme c'est le cas en x86.

Apparemment, c'est parce que le x86 à une énorme histoire de rétro-compatibilité avec ses ancêtres, jusqu'au 8086 16-bit et même 8080 8-bit, mais ARM a été développé à partir de zéro comme un processeur RISC 32-bit.

Par conséquent, pour manipuler des octets séparés en ARM, on doit tout de même utiliser des registres 32-bit.

Donc, LDRSB charge des octets depuis la chaîne vers R0, un par un. Les instructions suivantes, CMP et BEQ vérifient si l'octet chargé est 0. Si il n'est pas à 0, le contrôle passe au début du corps de la boucle. Et si c'est 0, la boucle est terminée.

À la fin de la fonction, la différence entre *eos* et *str* est calculée, 1 en est soustrait, et la valeur résultante est renvoyée via R0.

N.B. Les registres n'ont pas été sauvés dans cette fonction.

C'est parce que dans la convention d'appel ARM, les registres R0-R3 sont des « registres scratch », destinés à passer les arguments, et il n'est pas requis de restaurer leur valeur en sortant de la fonction, puisque la fonction appelante ne va plus les utiliser. Par conséquent, ils peuvent être utilisés comme bien nous semble.

Il n'y a pas d'autres registres utilisés ici, c'est pourquoi nous n'avons rien à sauvegarder sur la pile.

Ainsi, le contrôle peut être rendu à la fonction appelante par un simple saut (BX), à l'adresse contenue dans le registre LR.

avec optimisation Xcode 4.6.3 (LLVM) (Mode Thumb)

Listing 1.183: avec optimisation Xcode 4.6.3 (LLVM) (Mode Thumb)

```
_strlen
            MOV       R1, R0

loc_2DF6
            LDRB.W    R2, [R1],#1
            CMP       R2, #0
            BNE       loc_2DF6
            MVNS      R0, R0
            ADD       R0, R1
            BX        LR
```

Comme le conclut LLVM avec l'optimisation, *eos* et *str* n'ont pas besoin d'espace dans la pile, et peuvent toujours être stockés dans les registres.

Avant le début du corps de la boucle, *str* est toujours dans R0, et *eos*—dans R1.

L'instruction `LDRB.W R2, [R1],#1` charge, dans R2, un octet de la mémoire à l'adresse stockée dans R1, en étendant le signe à une valeur 32-bit, mais pas seulement cela. #1 à la fin de l'instruction indique un

[105] Le compilateur Keil considère le type *char* comme signé, tout comme MSVC et GCC.

« Adressage post-indexé » (« Post-indexed addressing »), qui signifie que 1 doit être ajouté à R1 après avoir chargé l'octet. Pour en lire plus à ce propos: 1.32.2 on page 442.

Ensuite vous pouvez voir CMP et BNE[106] dans le corps de la boucle, ces instructions continuent de boucler jusqu'à ce que 0 soit trouvé dans la chaîne.

Les instructions MVNS[107] (inverse tous les bits, comme NOT en x86) et ADD calculent $eos - str - 1$. (1.17.1 on page 208). En fait, ces deux instructions calculent $R0 = str + eos$, qui est effectivement équivalent à ce qui est dans le code source, et la raison de ceci à déjà été expliquée ici (1.17.1 on page 208).

Apparemment, LLVM, tout comme GCC, conclu que ce code peut être plus court (ou plus rapide).

avec optimisation Keil 6/2013 (Mode ARM)

Listing 1.184: avec optimisation Keil 6/2013 (Mode ARM)

```
_strlen
                MOV     R1, R0

loc_2C8
                LDRB    R2, [R1],#1
                CMP     R2, #0
                SUBEQ   R0, R1, R0
                SUBEQ   R0, R0, #1
                BNE     loc_2C8
                BX      LR
```

Presque la même chose que ce que nous avions vu avant, à l'exception que l'expression $str - eos - 1$ peut être calculée non pas à la fin de la fonction, mais dans le corps de la boucle. Le suffixe -EQ, comme nous devrions nous en souvenir, implique que l'instruction ne s'exécute que si les opérandes de la dernière instruction CMP qui a été exécutée avant étaient égaux. Ainsi, si R0 contient 0, les deux instructions SUBEQ sont exécutées et le résultat est laissé dans le registre R0.

ARM64

GCC avec optimisation (Linaro) 4.9

```
my_strlen :
        mov     x1, x0
        ; X1 est maintenant un pointeur temporaire (eos), se comportant comme un curseur
.L58 :
  ; charger un octet de X1 dans W2, incrémenter X1 (post-index)
        ldrb    w2, [x1],1
        ; Compare and Branch if NonZero : comparer W2 avec 0, sauter en .L58 si il ne l'est pas
        cbnz    w2, .L58
        ; calculer la différence entre le pointeur initial dans X0 et l'adresse courante dans ↙
    ↳ X1
        sub     x0, x1, x0
        ; decrement lowest 32-bit of result
        sub     w0, w0, #1
        ret
```

L'algorithme est le même que dans 1.17.1 on page 203 : trouver un octet à zéro, calculer la différence antre les pointeurs et décrémenter le résultat de 1.size_t Quelques commentaires ont été ajouté par l'auteur de ce livre.

La seule différence notable est que cet exemple est un peu faux:
my_strlen() renvoie une valeur *int* 32-bit, tandis qu'elle devrait renvoyer un type size_t ou un autre type 64-bit.

La raison est que, théoriquement, strlen() peut-être appelée pour un énorme bloc de mémoire qui dépasse 4GB, donc elle doit être capable de renvoyer une valeur 64-bit sur une plate-forme 64-bit.

[106] (PowerPC, ARM) Branch if Not Equal
[107] MoVe Not

À cause de cette erreur, la dernière instruction SUB opère sur la partie 32-bit du registre, tandis que la pénultième instruction SUB travaille sur un registre 64-bit complet (elle calcule la différence entre les pointeurs).

C'est une erreur de l'auteur, il est mieux de la laisser ainsi, comme un exemple de ce à quoi ressemble le code dans un tel cas.

GCC sans optimisation (Linaro) 4.9

```
my_strlen :
; prologue de la fonction
        sub     sp, sp, #32
; le premier argument (str) va être stocké dans [sp,8]
        str     x0, [sp,8]
        ldr     x0, [sp,8]
; copier "str" dans la variable "eos"
        str     x0, [sp,24]
        nop
.L62 :
; eos++
        ldr     x0, [sp,24]  ; charger "eos" dans X0
        add     x1, x0, 1    ; incrémenter X0
        str     x1, [sp,24]  ; sauver X0 dans "eos"
; charger dans W0 un octet de la mémoire à l'adresse dans X0
        ldrb    w0, [x0]
; est-ce zéro? (WZR est le registre 32-bit qui contient toujours zéro)
        cmp     w0, wzr
; sauter si différent de zéro (Branch Not Equal)
        bne     .L62
; octet à zéro trouvé. calculer maintenant la différence
; charger "eos" dans X1
        ldr     x1, [sp,24]
; charger "str" dans X0
        ldr     x0, [sp,8]
; calculer la différence
        sub     x0, x1, x0
; décrémenter le résultat
        sub     w0, w0, #1
; épilogue de la fonction
        add     sp, sp, 32
        ret
```

C'est plus verbeux. Les variables sont beaucoup manipulées vers et depuis la mémoire (pile locale). Il y a la même erreur ici: l'opération de décrémentation se produit sur la partie 32-bit du registre.

MIPS

Listing 1.185: avec optimisation GCC 4.4.5 (IDA)

```
my_strlen :
; la variable "eos" sera toujours dans $v1 :
                move    $v1, $a0

loc_4 :
; charger l'octet à l'adresse dans "eos" dans $a1 :
                lb      $a1, 0($v1)
                or      $at, $zero ; slot de $délai$ de branchement, NOP
; si l'octet chargé n'est pas zéro, sauter en loc_4 :
                bnez    $a1, loc_4
; incrémenter "eos" de toutes façons :
                addiu   $v1, 1 ; slot de $délai$ de branchement
; boucle terminée. inverser variable "str" :
                nor     $v0, $zero, $a0
; $v0=-str-1
                jr      $ra
; valeur de retour = $v1 + $v0 = eos + ( -str-1 ) = eos - str - 1
                addu    $v0, $v1, $v0 ; slot de $délai$ de branchement
```

Il manque en MIPS une instruction NOT, mais il y a NOR qui correspond à l'opération OR + NOT.

Cette opération est largement utilisée en électronique digitale[108]. Par exemple, l'Apollo Guidance Computer (ordinateur de guidage Apollo) utilisé dans le programme Apollo, a été construit en utilisant seulement 5600 portes NOR: [Jens Eickhoff, *Onboard Computers, Onboard Software and Satellite Operations: An Introduction*, (2011)]. Mais l'élément NOT n'est pas très populaire en programmation informatique.

Donc, l'opération NOT est implémentée ici avec `NOR DST, $ZERO, SRC`.

D'après le chapitre sur les fondamentaux 2.2 on page 456 nous savons qu'une inversion des bits d'un nombre signé est la même chose que changer son signe et soustraire 1 du résultat.

Donc ce que NOT fait ici est de prendre la valeur de str et de la transformer en $-str-1$. L'opération d'addition qui suit prépare le résultat.

1.17.2 Limites de chaînes

Il est intéressant de noter comment les paramètres sont passés à la fonction win32 *GetOpenFileName()*. Afin de l'appeler, il faut définir une liste des extensions de fichier autorisées:

```
        OPENFILENAME *LPOPENFILENAME;
        ...
        char * filter = "Text files (*.txt)\0*.txt\0MS Word files (*.doc)\0*.doc\0\0";
        ...
        LPOPENFILENAME = (OPENFILENAME *)malloc(sizeof(OPENFILENAME));
        ...
        LPOPENFILENAME->lpstrFilter = filter;
        ...

        if(GetOpenFileName(LPOPENFILENAME))
        {
                ...
```

Ce qui se passe ici, c'est que la liste de chaînes est passée à *GetOpenFileName()*. Ce n'est pas un problème de l'analyser: à chaque fois que l'on rencontre un octet nul, c'est un élément. Quand on rencontre deux octets nul, c'est la fin de la liste. Si vous passez cette chaîne à `printf()`, elle traitera le premier élément comme une simple chaîne.

Donc, ceci est un chaîne, ou...? Il est plus juste de dire que c'est un buffer contenants plusieurs chaînes-C terminées par zéro, qui peut être stocké et traité comme un tout.

Un autre exemple est la fonction *strtok()*. Elle prend une chaîne et y écrit des octets nul. C'est ainsi qu'elle transforme la chaîne d'entrée en une sorte de buffer, qui contient plusieurs chaînes-C terminées par zéro.

1.18 Remplacement de certaines instructions arithmétiques par d'autres

Lors de la recherche d'optimisation, une instruction peut-être remplacée par une autre, ou même par un groupe d'instructions. Par exemple, ADD et SUB peuvent se remplacer: ligne 18 de liste. **??**.

Par exemple, l'instruction LEA est souvent utilisée pour des calculs arithmétiques simples: **??** on page ??.

1.18.1 Multiplication

Multiplication en utilisant l'addition

Voici un exemple simple:

```
unsigned int f(unsigned int a)
{
        return a*8;
};
```

[108] NOR est appelé « porte universelle »

La multiplication par 8 a été remplacée par 3 instructions d'addition, qui font la même chose. Il semble que l'optimiseur de MSVC a décidé que ce code peut être plus rapide.

Listing 1.186: MSVC 2010 avec optimisation

```
_TEXT   SEGMENT
_a$ = 8         ; size = 4
_f      PROC
; File c :\polygon\c\2.c
        mov     eax, DWORD PTR _a$[esp-4]
        add     eax, eax
        add     eax, eax
        add     eax, eax
        ret     0
_f      ENDP
_TEXT   ENDS
END
```

Multiplication en utilisant le décalage

Les instructions de multiplication et de division par un nombre qui est une puissance de 2 sont souvent remplacées par des instructions de décalage.

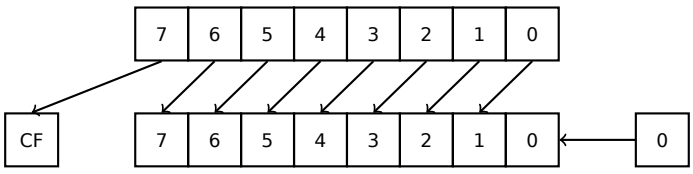
```
unsigned int f(unsigned int a)
{
        return a*4;
};
```

Listing 1.187: MSVC 2010 sans optimisation

```
_a$ = 8         ; size = 4
_f      PROC
        push    ebp
        mov     ebp, esp
        mov     eax, DWORD PTR _a$[ebp]
        shl     eax, 2
        pop     ebp
        ret     0
_f      ENDP
```

La multiplication par 4 consiste en un décalage du nombre de 2 bits vers la gauche et l'insertion de deux bits à zéro sur la droite (les deux derniers bits). C'est comme multiplier 3 par 100 —nous devons juste ajouter deux zéros sur la droite.

C'est ainsi que fonctionne l'instruction de décalage vers la gauche:



Les bits ajoutés à droite sont toujours des zéros.

Multiplication par 4 en ARM:

Listing 1.188: sans optimisation Keil 6/2013 (Mode ARM)

```
f PROC
        LSL      r0,r0,#2
        BX       lr
        ENDP
```

Multiplication par 4 en MIPS:

Listing 1.189: GCC 4.4.5 avec optimisation (IDA)

```
      jr      $ra
      sll     $v0, $a0, 2 ; branch delay slot
```

SLL signifie « Shift Left Logical » (décalage logique à gauche).

Multiplication en utilisant le décalage, la soustraction, et l'addition

Il est aussi possible de se passer des opérations de multiplication lorsque l'on multiplie par des nombres comme 7 ou 17, toujours en utilisant le décalage. Les mathématiques utilisées ici sont assez faciles.

32-bit

```
#include <stdint.h>

int f1(int a)
{
        return a*7;
};

int f2(int a)
{
        return a*28;
};

int f3(int a)
{
        return a*17;
};
```

x86

Listing 1.190: MSVC 2012 avec optimisation

```
; a*7
_a$ = 8
_f1     PROC
        mov     ecx, DWORD PTR _a$[esp-4]
; ECX=a
        lea     eax, DWORD PTR [ecx*8]
; EAX=ECX*8
        sub     eax, ecx
; EAX=EAX-ECX=ECX*8-ECX=ECX*7=a*7
        ret     0
_f1     ENDP

; a*28
_a$ = 8
_f2     PROC
        mov     ecx, DWORD PTR _a$[esp-4]
; ECX=a
        lea     eax, DWORD PTR [ecx*8]
; EAX=ECX*8
        sub     eax, ecx
; EAX=EAX-ECX=ECX*8-ECX=ECX*7=a*7
        shl     eax, 2
; EAX=EAX<<2=(a*7)*4=a*28
        ret     0
_f2     ENDP

; a*17
_a$ = 8
_f3     PROC
        mov     eax, DWORD PTR _a$[esp-4]
; EAX=a
        shl     eax, 4
; EAX=EAX<<4=EAX*16=a*16
        add     eax, DWORD PTR _a$[esp-4]
; EAX=EAX+a=a*16+a=a*17
        ret     0
_f3     ENDP
```

ARM

Keil pour le mode ARM tire partie du décalage de registre du second opérande:

Listing 1.191: avec optimisation Keil 6/2013 (Mode ARM)

```
; a*7
||f1|| PROC
        RSB      r0,r0,r0,LSL #3
; R0=R0<<3-R0=R0*8-R0=a*8-a=a*7
        BX       lr
        ENDP

; a*28
||f2|| PROC
        RSB      r0,r0,r0,LSL #3
; R0=R0<<3-R0=R0*8-R0=a*8-a=a*7
        LSL      r0,r0,#2
; R0=R0<<2=R0*4=a*7*4=a*28
        BX       lr
        ENDP

; a*17
||f3|| PROC
        ADD      r0,r0,r0,LSL #4
; R0=R0+R0<<4=R0+R0*16=R0*17=a*17
        BX       lr
        ENDP
```

Mais ce n'est pas disponible en mode Thumb. Il ne peut donc pas l'optimiser:

Listing 1.192: avec optimisation Keil 6/2013 (Mode Thumb)

```
; a*7
||f1|| PROC
        LSLS     r1,r0,#3
; R1=R0<<3=a<<3=a*8
        SUBS     r0,r1,r0
; R0=R1-R0=a*8-a=a*7
        BX       lr
        ENDP

; a*28
||f2|| PROC
        MOVS     r1,#0x1c  ; 28
; R1=28
        MULS     r0,r1,r0
; R0=R1*R0=28*a
        BX       lr
        ENDP

; a*17
||f3|| PROC
        LSLS     r1,r0,#4
; R1=R0<<4=R0*16=a*16
        ADDS     r0,r0,r1
; R0=R0+R1=a+a*16=a*17
        BX       lr
        ENDP
```

MIPS

Listing 1.193: GCC 4.4.5 avec optimisation (IDA)

```
_f1 :
                sll     $v0, $a0, 3
; $v0 = $a0<<3 = $a0*8
                jr      $ra
```

```
                subu    $v0, $a0 ; branch delay slot
; $v0 = $v0-$a0 = $a0*8-$a0 = $a0*7

_f2 :
                sll     $v0, $a0, 5
; $v0 = $a0<<5 = $a0*32
                sll     $a0, 2
; $a0 = $a0<<2 = $a0*4
                jr      $ra
                subu    $v0, $a0 ; branch delay slot
; $v0 = $a0*32-$a0*4 = $a0*28

_f3 :
                sll     $v0, $a0, 4
; $v0 = $a0<<4 = $a0*16
                jr      $ra
                addu    $v0, $a0 ; branch delay slot
; $v0 = $a0*16+$a0 = $a0*17
```

64-bit

```
#include <stdint.h>

int64_t f1(int64_t a)
{
        return a*7;
};

int64_t f2(int64_t a)
{
        return a*28;
};

int64_t f3(int64_t a)
{
        return a*17;
};
```

x64

Listing 1.194: MSVC 2012 avec optimisation

```
; a*7
f1 :
        lea     rax, [0+rdi*8]
; RAX=RDI*8=a*8
        sub     rax, rdi
; RAX=RAX-RDI=a*8-a=a*7
        ret

; a*28
f2 :
        lea     rax, [0+rdi*4]
; RAX=RDI*4=a*4
        sal     rdi, 5
; RDI=RDI<<5=RDI*32=a*32
        sub     rdi, rax
; RDI=RDI-RAX=a*32-a*4=a*28
        mov     rax, rdi
        ret

; a*17
f3 :
        mov     rax, rdi
        sal     rax, 4
```

```
; RAX=RAX<<4=a*16
        add     rax, rdi
; RAX=a*16+a=a*17
        ret
```

ARM64

GCC 4.9 pour ARM64 est aussi concis, grâce au modificateur de décalage:

Listing 1.195: GCC (Linaro) 4.9 avec optimisation ARM64

```
; a*7
f1 :
        lsl     x1, x0, 3
; X1=X0<<3=X0*8=a*8
        sub     x0, x1, x0
; X0=X1-X0=a*8-a=a*7
        ret

; a*28
f2 :
        lsl     x1, x0, 5
; X1=X0<<5=a*32
        sub     x0, x1, x0, lsl 2
; X0=X1-X0<<2=a*32-a<<2=a*32-a*4=a*28
        ret

; a*17
f3 :
        add     x0, x0, x0, lsl 4
; X0=X0+X0<<4=a+a*16=a*17
        ret
```

Algorithme de multiplication de Booth

Il fût un temps où les ordinateurs étaient si gros et chers, que certains d'entre eux ne disposaient pas de la multiplication dans le CPU, comme le Data General Nova. Et lorsque l'on avait besoin de l'opérateur de multiplication, il pouvait être fourni au niveau logiciel, par exemple, en utilisant l'algorithme de multiplication de Booth. C'est un algorithme de multiplication qui utilise seulement des opérations d'addition et de décalage.

Ce que les optimiseurs des compilateurs modernes font n'est pas la même chose, mais le but (multiplication) et les ressources (des opérations plus rapides) sont les mêmes.

1.18.2 Division

Division en utilisant des décalages

Exemple de division par 4:

```
unsigned int f(unsigned int a)
{
        return a/4;
};
```

Nous obtenons (MSVC 2010):

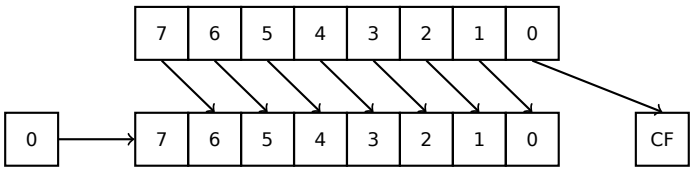
Listing 1.196: MSVC 2010

```
_a$ = 8         ; size = 4
_f      PROC
        mov     eax, DWORD PTR _a$[esp-4]
        shr     eax, 2
        ret     0
```

```
_f      ENDP
```

L'instruction SHR (*SHift Right* décalage à droite) dans cet exemple décale un nombre de 2 bits vers la droite. Les deux bits libérés à gauche (i.e., les deux bits les plus significatifs) sont mis à zéro. Les deux bits les moins significatifs sont perdus. En fait, ces deux bits perdus sont le reste de la division.

L'instruction SHR fonctionne tout comme SHL, mais dans l'autre direction.



Il est facile de comprendre si vous imaginez le nombre 23 dans le système décimal. 23 peut être facilement divisé par 10, juste en supprimant le dernier chiffre (3—le reste de la division). Il reste 2 après l'opération, qui est le quotient.

Donc, le reste est perdu, mais c'est OK, nous travaillons de toutes façons sur des valeurs entières, ce sont sont pas des nombres réels !

Division par 4 en ARM:

Listing 1.197: sans optimisation Keil 6/2013 (Mode ARM)

```
f PROC
        LSR      r0,r0,#2
        BX       lr
        ENDP
```

Division par 4 en MIPS:

Listing 1.198: GCC 4.4.5 avec optimisation (IDA)

```
        jr      $ra
        srl     $v0, $a0, 2 ; slot de délai de branchement
```

L'instruction SLR est « Shift Right Logical » (décalage logique à droite).

1.18.3 Exercice

- http://challenges.re/59

1.19 Unité à virgule flottante

Le FPU est un dispositif à l'intérieur du CPU, spécialement conçu pour traiter les nombres à virgules flottantes.

Il était appelé « coprocesseur » dans le passé et il était en dehors du CPU.

1.19.1 IEEE 754

Un nombre au format IEEE 754 consiste en un *signe*, un *significande* (aussi appelé *fraction*) et un *exposant*.

1.19.2 x86

Ça vaut la peine de jeter un œil sur les machines à base de piles[109]ou d'apprendre les bases du langage Forth[110], avant d'étudier le FPU en x86.

[109] wikipedia.org/wiki/Stack_machine
[110] wikipedia.org/wiki/Forth_(programming_language)

Il est intéressant de savoir que dans le passé (avant le CPU 80486) le coprocesseur était une puce séparée et n'était pas toujours préinstallé sur la carte mère. Il était possible de l'acheter séparément et de l'installer[111].

A partir du CPU 80486 DX, le FPU est intégré dans le CPU.

L'instruction FWAIT nous rappelle le fait qu'elle passe le CPU dans un état d'attente, jusqu'à ce que le FPU ait fini son traitement.

Un autre rudiment est le fait que les opcodes d'instruction FPU commencent avec ce qui est appelé l'opcode-« d'échappement » (D8..DF), i.e., opcodes passés à un coprocesseur séparé.

Le FPU a une pile capable de contenir 8 registres de 80-bit, et chaque registre peut contenir un nombre au format IEEE 754[112].

Ce sont ST(0)..ST(7). Par concision, IDA et OllyDbg montrent ST(0) comme ST, qui est représenté dans certains livres et manuels comme « Stack Top ».

1.19.3 ARM, MIPS, x86/x64 SIMD

En ARM et MIPS le FPU n'a pas de pile, mais un ensemble de registres, qui peuvent être accédés aléatoirement, comme GPR.

La même idéologie est utilisée dans l'extension SIMD des CPUs x86/x64.

1.19.4 C/C++

Le standard des langages C/C++ offre au moins deux types de nombres à virgule flottante, *float* (*simple-précision*[113], 32 bits) [114] et *double* (*double-précision*[115], 64 bits).

Dans [Donald E. Knuth, *The Art of Computer Programming*, Volume 2, 3rd ed., (1997)246] nous pouvons trouver que *simple-précision* signifie que la valeur flottante peut être stockée dans un simple mot machine [32-bit], *double-précision* signifie qu'elle peut être stockée dans deux mots (64 bits).

GCC supporte également le type *long double* (*précision étendue*[116], 80 bit), que MSVC ne supporte pas.

Le type *float* nécessite le même nombre de bits que le type *int* dans les environnements 32-bit, mais la représentation du nombre est complètement différente.

1.19.5 Exemple simple

Considérons cet exemple simple:

```
#include <stdio.h>

double f (double a, double b)
{
        return a/3.14 + b*4.1;
};

int main()
{
        printf ("%f\n", f(1.2, 3.4));
};
```

[111] Par exemple, John Carmack a utilisé des valeurs arithmétiques à virgule fixe (wikipedia.org/wiki/Fixed-point_arithmetic) dans son jeu vidéo Doom, stockées dans des registres 32-bit GPR (16 bit pour la partie entière et 16 bit pour la partie fractionnaire), donc Doom pouvait fonctionner sur des ordinateurs 32-bit sans FPU, i.e., 80386 et 80486 SX.
[112] wikipedia.org/wiki/IEEE_floating_point
[113] wikipedia.org/wiki/Single-precision_floating-point_format
[114] le format des nombres à virgule flottante simple précision est aussi abordé dans la section *Travailler avec le type float comme une structure* (1.24.6 on page 375)
[115] wikipedia.org/wiki/Double-precision_floating-point_format
[116] wikipedia.org/wiki/Extended_precision

x86

MSVC

Compilons-le avec MSVC 2010:

Listing 1.199: MSVC 2010: f()

```
CONST    SEGMENT
__real@4010666666666666 DQ 04010666666666666r    ; 4.1
CONST    ENDS
CONST    SEGMENT
__real@40091eb851eb851f DQ 040091eb851eb851fr    ; 3.14
CONST    ENDS
_TEXT    SEGMENT
_a$ = 8           ; size = 8
_b$ = 16          ; size = 8
_f  PROC
    push   ebp
    mov    ebp, esp
    fld    QWORD PTR _a$[ebp]

; état courant de la pile : ST(0) = _a

    fdiv   QWORD PTR __real@40091eb851eb851f

; état courant de la pile : ST(0) = résultat de _a divisé par 3.14

    fld    QWORD PTR _b$[ebp]

; état courant de la pile : ST(0) = _b;
; ST(1) = résultat de  _a divisé par 3.14

    fmul   QWORD PTR __real@4010666666666666

; état courant de la pile :
; ST(0) = résultat de _b * 4.1;
; ST(1) = résultat de _a divisé par 3.14

    faddp  ST(1), ST(0)

; état courant de la pile : ST(0) = résultat de l'addition

    pop    ebp
    ret    0
_f  ENDP
```

FLD prend 8 octets depuis la pile et charge le nombre dans le registre ST(0), en le convertissant automatiquement dans le format interne sur 80-bit (*précision étendue*):

FDIV divise la valeur dans ST(0) par le nombre stocké à l'adresse __real@40091eb851eb851f —la valeur 3.14 est encodée ici. La syntaxe assembleur ne supporte pas les nombres à virgule flottante, donc ce que l'on voit ici est la représentation hexadécimale de 3.14 au format 64-bit IEEE 754.

Après l'exécution de FDIV, ST(0) contient le quotient.

À propos, il y a aussi l'instruction FDIVP, qui divise ST(1) par ST(0), prenant ces deux valeurs dans la pile et poussant le résultant. Si vous connaissez le langage Forth[117], vous pouvez comprendre rapidement que ceci est une machine à pile[118].

L'instruction FLD subséquente pousse la valeur de b sur la pile.

Après cela, le quotient est placé dans ST(1), et ST(0) a la valeur de b.

L'instruction suivante effectue la multiplication: b de ST(0) est multiplié par la valeur en __real@4010666666666666 (le nombre 4.1 est là) et met le résultat dans le registre ST(0).

[117] wikipedia.org/wiki/Forth_(programming_language)
[118] wikipedia.org/wiki/Stack_machine

La dernière instruction `FADDP` ajoute les deux valeurs au sommet de la pile, stockant le résultat dans `ST(1)` et supprimant la valeur de `ST(0)`, laissant ainsi le résultat au sommet de la pile, dans `ST(0)`.

La fonction doit renvoyer son résultat dans le registre `ST(0)`, donc il n'y a aucune autre instruction après `FADDP`, excepté l'épilogue de la fonction.

MSVC + OllyDbg

2 paires de mots 32-bit sont marquées en rouge sur la pile. Chaque paire est un double au format IEEE 754 et est passée depuis `main()`.

Nous voyons comment le premier FLD charge une valeur (1.2) depuis la pile et la stocke dans `ST(0)` :

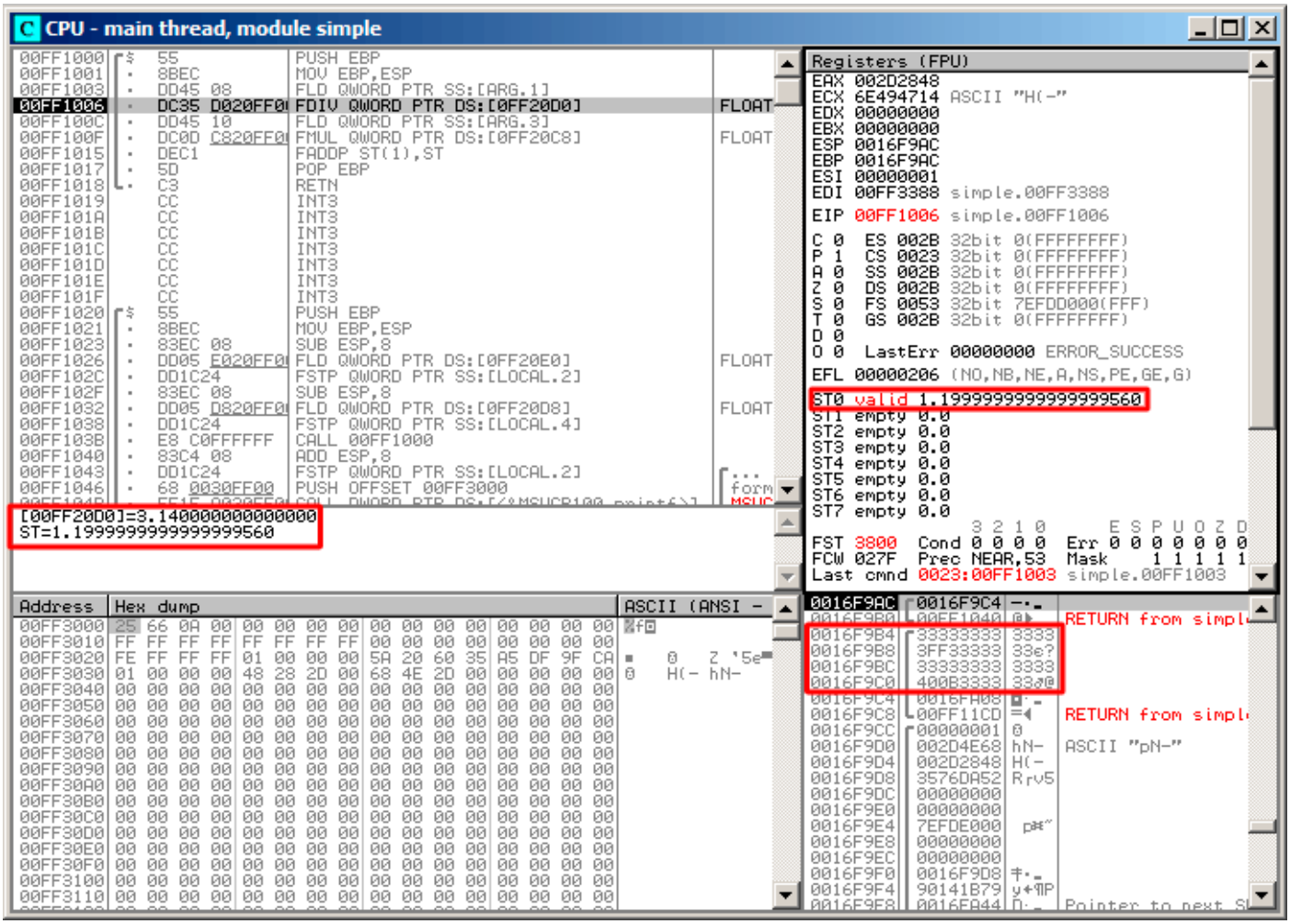


Fig. 1.62: OllyDbg : le premier FLD a été exécuté

À cause des inévitables erreurs de conversion depuis un nombre flottant 64-bit au format IEEE 754 vers 80-bit (utilisé en interne par le FPU), ici nous voyons 1.199..., qui est proche de 1.2.

EIP pointe maintenant sur l'instruction suivante (FDIV), qui charge un double (une constante) depuis la mémoire. Par commodité, OllyDbg affiche sa valeur: 3.14

Continuons l'exécution pas à pas. FDIV a été exécuté, maintenant ST(0) contient 0.382...(quotient):

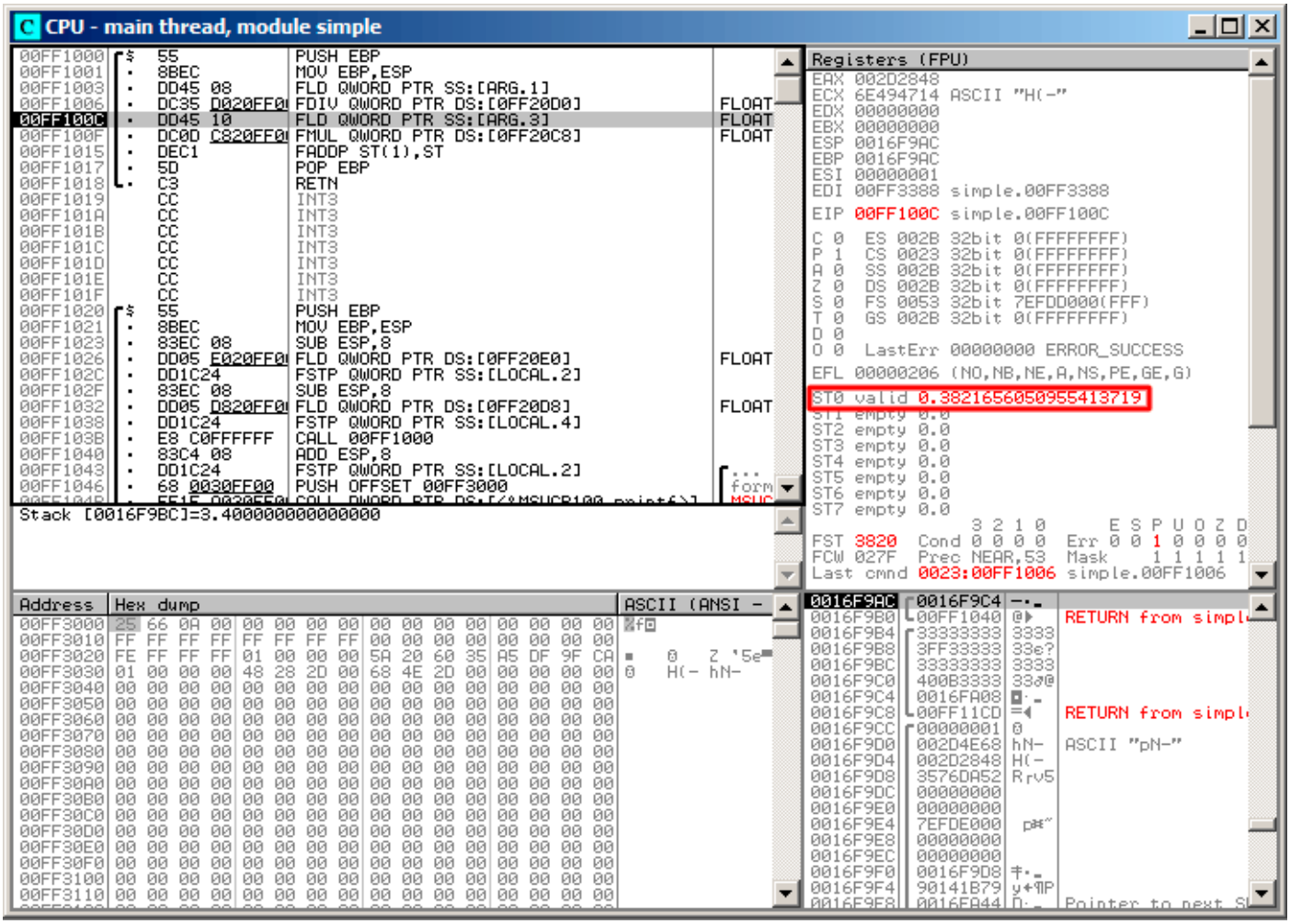


Fig. 1.63: OllyDbg : FDIV a été exécuté

Troisième étape: le FLD suivant a été exécuté, chargeant 3.4 dans ST(0) (ici nous voyons la valeur approximative 3.39999...):

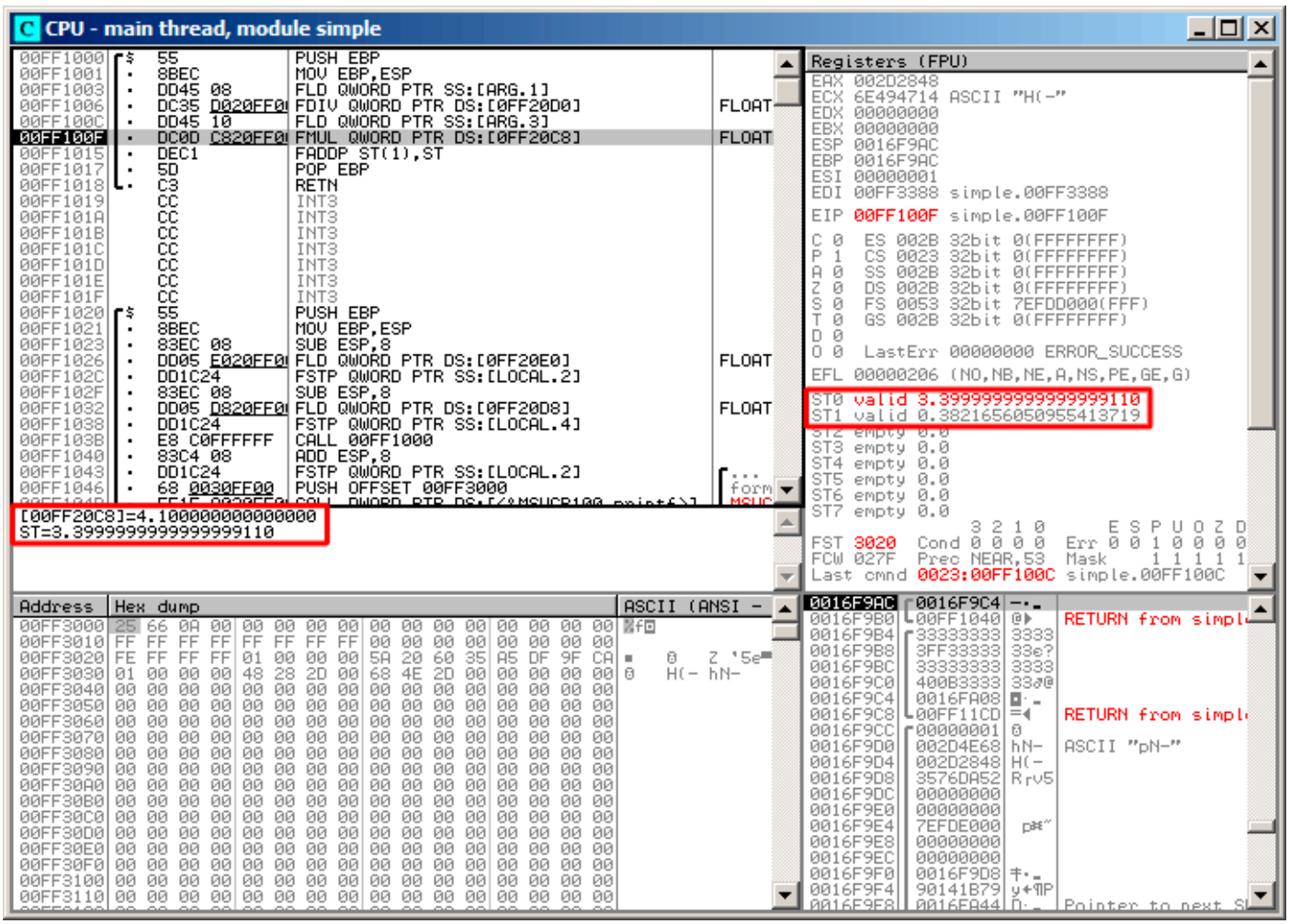


Fig. 1.64: OllyDbg : le second FLD a été exécuté

En même temps, le quotient *est poussé* dans ST(1). Exactement maintenant, EIP pointe sur la prochaine instruction: FMUL. Ceci charge la constante 4.1 depuis la mémoire, ce que montre OllyDbg.

Suivante: FMUL a été exécutée, donc maintenant le produit est dans ST(0) :

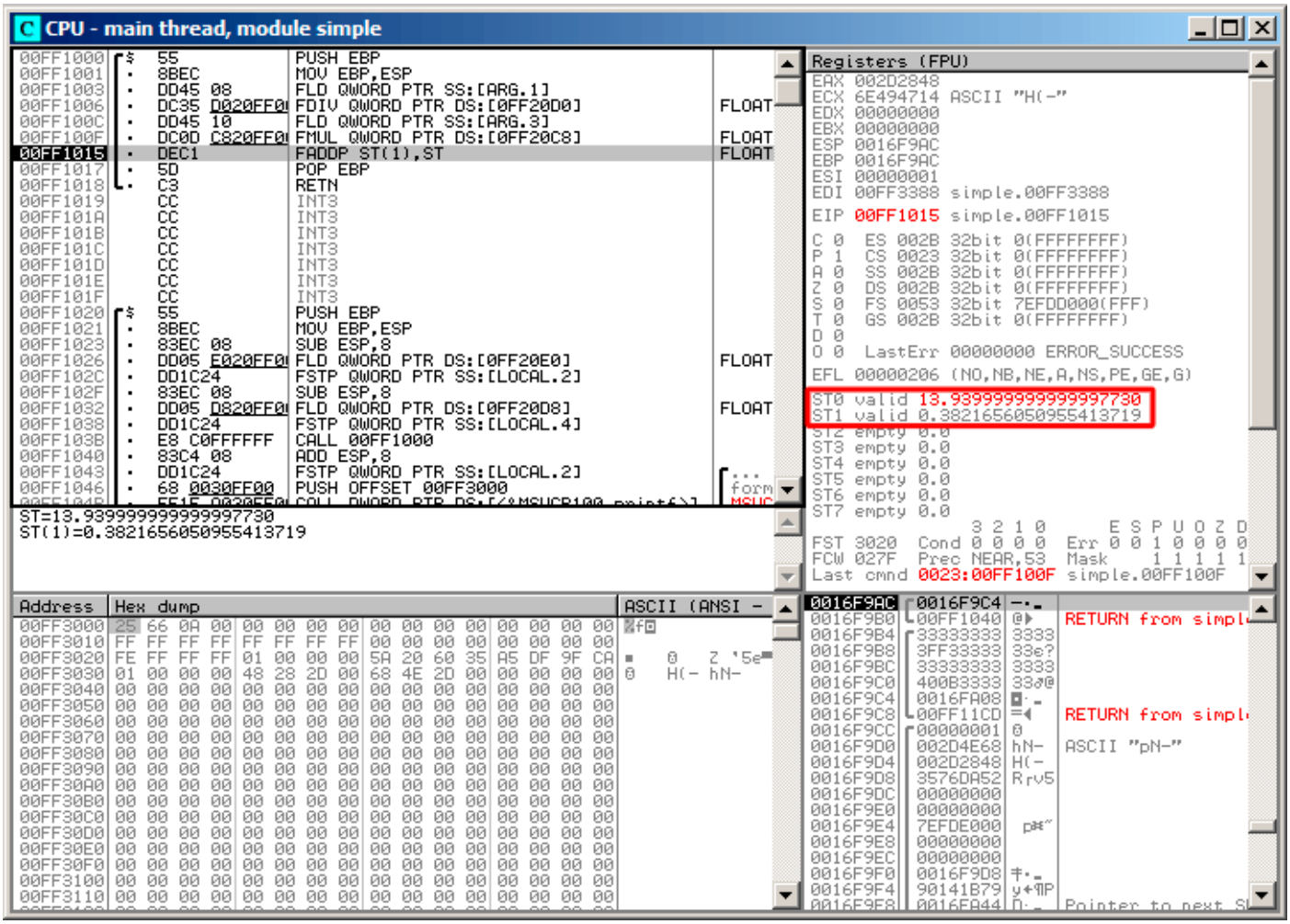


Fig. 1.65: OllyDbg : FMUL a été exécuté

Suivante: FADDP a été exécutée, maintenant le résultat de l'addition est dans ST(0), et ST(1) est vidé.

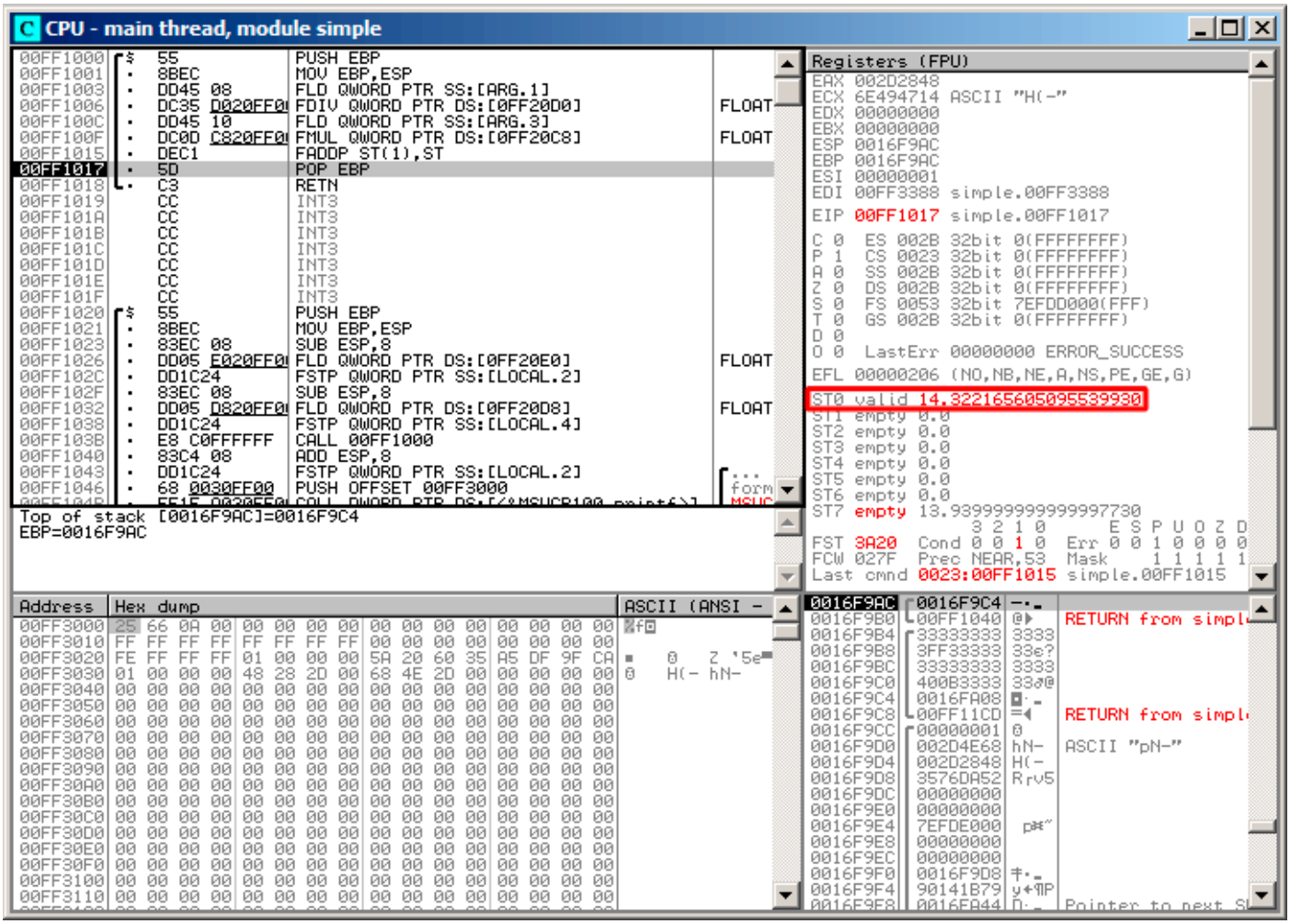


Fig. 1.66: OllyDbg : FADDP a été exécuté

Le résultat est laissé dans ST(0), car la fonction renvoie son résultat dans ST(0).

main() prend cette valeur depuis le registre plus loin.

Nous voyons quelque chose d'inhabituel: la valeur 13.93...se trouve maintenant dans ST(7). Pourquoi?

Comme nous l'avons lu il y a quelque temps dans ce livre, les registres FPU sont une pile: 1.19.2 on page 219. Mais ceci est une simplification.

Imaginez si cela était implémenté *en hardware* comme cela est décrit, alors tout le contenu des 7 registres devrait être déplacé (ou copié) dans les registres adjacents lors d'un push ou d'un pop, et ceci nécessite beaucoup de travail.

En réalité, le FPU a seulement 8 registres et un pointeur (appelé TOP) qui contient un numéro de registre, qui est le « haut de la pile » courant.

Lorsqu'une valeur est poussée sur la pile, TOP est déplacé sur le registre disponible suivant, et une valeur est écrite dans ce registre.

La procédure est inversée si la valeur est lue, toutefois, le registre qui a été libéré n'est pas vidé (il serait possible de le vider, mais ceci nécessite plus de travail qui peut dégrader les performances). Donc, c'est ce que nous voyons ici.

On peut dire que FADDP sauve la somme sur la pile, et y supprime un élément.

Mais en fait, cette instruction sauve la somme et ensuite décale TOP.

Plus précisément, les registres du FPU sont un tampon circulaire.

GCC

GCC 4.4.1 (avec l'option -03) génère le même code, juste un peu différent:

Listing 1.200: GCC 4.4.1 avec optimisation

```
                public f
f               proc near

arg_0           = qword ptr  8
arg_8           = qword ptr  10h

                push    ebp
                fld     ds :dbl_8048608 ; 3.14

; état de la pile maintenant : ST(0) = 3.14

                mov     ebp, esp
                fdivr   [ebp+arg_0]

; état de la pile maintenant : ST(0) = résultat de la division

                fld     ds :dbl_8048610 ; 4.1

; état de la pile maintenant : ST(0) = 4.1, ST(1) = résultat de la division

                fmul    [ebp+arg_8]

; état de la pile maintenant : ST(0) = résultat de la multiplication, ST(1) = résultat de la ↙
    ↳ division

                pop     ebp
                faddp   st(1), st

; état de la pile maintenant : ST(0) = résultat de l'addition

                retn
f               endp
```

La différence est que, tout d'abord, 3.14 est poussé sur la pile (dans ST(0)), et ensuite la valeur dans arg_0 est divisée par la valeur dans ST(0).

FDIVR signifie *Reverse Divide* —pour diviser avec le diviseur et le dividende échangés l'un avec l'autre. Il n'y a pas d'instruction de ce genre pour la multiplication puisque c'est une opération commutative, donc nous avons seulement FMUL sans son homologue -R.

FADDP ajoute les deux valeurs mais supprime aussi une valeur de la pile. Après cette opération, ST(0) contient la somme.

ARM: avec optimisation Xcode 4.6.3 (LLVM) (Mode ARM)

Jusqu'à la standardisation du support de la virgule flottante, certains fabricants de processeur ont ajouté leur propre instructions étendues. Ensuite, VFP (*Vector Floating Point*) a été standardisé.

Une différence importante par rapport au x86 est qu'en ARM, il n'y a pas de pile, vous travaillez seulement avec des registres.

Listing 1.201: avec optimisation Xcode 4.6.3 (LLVM) (Mode ARM)

```
f
                VLDR            D16, =3.14
                VMOV            D17, R0, R1 ; charge "a"
                VMOV            D18, R2, R3 ; charge "b"
                VDIV.F64        D16, D17, D16 ; a/3.14
                VLDR            D17, =4.1
                VMUL.F64        D17, D18, D17 ; b*4.1
                VADD.F64        D16, D17, D16 ; +
                VMOV            R0, R1, D16
                BX              LR

dbl_2C98        DCFD 3.14               ; DATA XREF : f
dbl_2CA0        DCFD 4.1                ; DATA XREF : f+10
```

Donc, nous voyons ici que des nouveaux registres sont utilisés, avec le préfixe D.

Ce sont des registres 64-bits, il y en a 32, et ils peuvent être utilisés tant pour des nombres à virgules flottantes (double) que pour des opérations SIMD (c'est appelé NEON ici en ARM).

Il y a aussi 32 S-registres 32 bits, destinés à être utilisés pour les nombres à virgules flottantes simple précision (float).

C'est facile à retenir: les registres D sont pour les nombres en double précision, tandis que les registres S—-pour les nombres en simple précision Pour aller plus loin: **??** on page ??.

Les deux constantes (3.14 et 4.1) sont stockées en mémoire au format IEEE 754.

VLDR et VMOV, comme il peut en être facilement déduit, sont analogues aux instructions LDR et MOV, mais travaillent avec des registres D.

Il est à noter que ces instructions, tout comme les registres D, sont destinées non seulement pour les nombres à virgules flottantes, mais peuvent aussi être utilisées pour des opérations SIMD (NEON) et cela va être montré bientôt.

Les arguments sont passés à la fonction de manière classique, via les R-registres, toutefois, chaque nombre en double précision a une taille de 64 bits, donc deux R-registres sont nécessaires pour passer chacun d'entre eux.

VMOV D17, R0, R1 au début, combine les deux valeurs 32-bit de R0 et R1 en une valeur 64-bit et la sauve dans D17.

VMOV R0, R1, D16 est l'opération inverse: ce qui est dans D16 est séparé dans deux registres, R0 et R1, car un nombre en double précision qui nécessite 64 bit pour le stockage, est renvoyé dans R0 et R1.

VDIV, VMUL and VADD, sont des instructions pour traiter des nombres à virgule flottante, qui calculent respectivement le quotient, produit et la somme.

Le code pour Thumb-2 est similaire.

ARM: avec optimisation Keil 6/2013 (Mode Thumb)

```
f
                PUSH    {R3-R7,LR}
                MOVS    R7, R2
                MOVS    R4, R3
                MOVS    R5, R0
                MOVS    R6, R1
                LDR     R2, =0x66666666 ; 4.1
                LDR     R3, =0x40106666
                MOVS    R0, R7
                MOVS    R1, R4
                BL      __aeabi_dmul
                MOVS    R7, R0
                MOVS    R4, R1
                LDR     R2, =0x51EB851F ; 3.14
                LDR     R3, =0x40091EB8
                MOVS    R0, R5
                MOVS    R1, R6
                BL      __aeabi_ddiv
                MOVS    R2, R7
                MOVS    R3, R4
                BL      __aeabi_dadd
                POP     {R3-R7,PC}

; 4.1 au format IEEE 754:
dword_364       DCD 0x66666666          ; DATA XREF : f+A
dword_368       DCD 0x40106666          ; DATA XREF : f+C
; 3.14 au format IEEE 754:
dword_36C       DCD 0x51EB851F          ; DATA XREF : f+1A
dword_370       DCD 0x40091EB8          ; DATA XREF : f+1C
```

Code généré par Keil pour un processeur sans FPU ou support pour NEON.

Les nombres en virgule flottante double précision sont passés par des R-registres génériques et au lieu d'instructions FPU, des fonctions d'une bibliothèque de service sont appelées (comme __aeabi_dmul,

__aeabi_ddiv, __aeabi_dadd) qui émulent la multiplication, la division et l'addition pour les nombres à virgule flottante.

Bien sûr, c'est plus lent qu'un coprocesseur FPU, mais toujours mieux que rien.

À propos, de telles bibliothèques d'émulation de FPU étaient très populaires dans le monde x86 lorsque les coprocesseurs étaient rares et chers, et étaient installés seulement dans des ordinateurs coûteux.

L'émulation d'un coprocesseur FPU est appelée *soft float* ou *armel* (*emulation*) dans le monde ARM, alors que l'utilisation des instructions d'un coprocesseur FPU est appelée *hard float* ou *armhf*.

ARM64: GCC avec optimisation (Linaro) 4.9

Code très compact:

Listing 1.202: GCC avec optimisation (Linaro) 4.9

```
f :
; D0 = a, D1 = b
        ldr     d2, .LC25       ; 3.14
; D2 = 3.14
        fdiv    d0, d0, d2
; D0 = D0/D2 = a/3.14
        ldr     d2, .LC26       ; 4.1
; D2 = 4.1
        fmadd   d0, d1, d2, d0
; D0 = D1*D2+D0 = b*4.1+a/3.14
        ret

; constantes au format IEEE 754:
.LC25 :
        .word   1374389535      ; 3.14
        .word   1074339512
.LC26 :
        .word   1717986918      ; 4.1
        .word   1074816614
```

ARM64: GCC sans optimisation (Linaro) 4.9

Listing 1.203: GCC sans optimisation (Linaro) 4.9

```
f :
        sub     sp, sp, #16
        str     d0, [sp,8]      ; sauve "a" dans le Register Save Area
        str     d1, [sp]        ; sauve "b" dans le Register Save Area
        ldr     x1, [sp,8]
; X1 = a
        ldr     x0, .LC25
; X0 = 3.14
        fmov    d0, x1
        fmov    d1, x0
; D0 = a, D1 = 3.14
        fdiv    d0, d0, d1
; D0 = D0/D1 = a/3.14

        fmov    x1, d0
; X1 = a/3.14
        ldr     x2, [sp]
; X2 = b
        ldr     x0, .LC26
; X0 = 4.1
        fmov    d0, x2
; D0 = b
        fmov    d1, x0
; D1 = 4.1
        fmul    d0, d0, d1
; D0 = D0*D1 = b*4.1
```

```
        fmov    x0, d0
; X0 = D0 = b*4.1
        fmov    d0, x1
; D0 = a/3.14
        fmov    d1, x0
; D1 = X0 = b*4.1
        fadd    d0, d0, d1
; D0 = D0+D1 = a/3.14 + b*4.1

        fmov    x0, d0 ; \ code redondant
        fmov    d0, x0 ; /
        add     sp, sp, 16
        ret
.LC25 :
        .word   1374389535      ; 3.14
        .word   1074339512
.LC26 :
        .word   1717986918      ; 4.1
        .word   1074816614
```

GCC sans optimisation est plus verbeux.

Il y a des nombreuses modifications de valeur inutiles, incluant du code clairement redondant (les deux dernières instructions FMOV). Sans doute que GCC 4.9 n'est pas encore très bon pour la génération de code ARM64.

Il est utile de noter qu'ARM64 possède des registres 64-bit, et que les D-registres sont aussi 64-bit.

Donc le compilateur est libre de sauver des valeurs de type *double* dans GPRs au lieu de la pile locale. Ce n'est pas possible sur des CPUs 32-bit.

Et encore, à titre d'exercice, vous pouvez essayer d'optimiser manuellement cette fonction, sans introduire de nouvelles instructions comme FMADD.

MIPS

MIPS peut supporter plusieurs coprocesseurs (jusqu'à 4), le zérotième[119] est un coprocesseur contrôleur spécial, et celui d'indice 1 est le FPU.

Comme en ARM, le coprocesseur MIPS n'est pas une machine à pile, il comprend 32 registres 32-bit ($F0-$F31): **??** on page ??.

Lorsque l'on doit travailler avec des valeurs *double* 64-bit, une paire de F-registres 32-bit est utilisée.

Listing 1.204: GCC 4.4.5 avec optimisation (IDA)

```
f :
; $f12-$f13=A
; $f14-$f15=B
                lui     $v0, (dword_C4 >> 16) ; ?
; charge la partie 32-bit basse de la constante 3.14 dans $f0 :
                lwc1    $f0, dword_BC
                or      $at, $zero            ; slot de délai de chargement, NOP
; charge la partie 32-bit haute de la constante 3.14 dans $f1 :
                lwc1    $f1, $LC0
                lui     $v0, ($LC1 >> 16)     ; ?
; A dans $f12-$f13, la constante 3.14 dans $f0-$f1, effectuer la division :
                div.d   $f0, $f12, $f0
; $f0-$f1=A/3.14
; charge la partie 32-bit basse de la constante 4.1 dans $f2 :
                lwc1    $f2, dword_C4
                or      $at, $zero            ; slot de délai de chargement, NOP
; charge la partie 32-bit haute de la constante 4.1 dans $f3 :
                lwc1    $f3, $LC1
                or      $at, $zero            ; slot de délai de chargement, NOP
; B dans $f14-$f15, la constante 4.1 dans $f2-$f3, effectuer la multiplication :
                mul.d   $f2, $f14, $f2
; $f2-$f3=B*4.1
                jr      $ra
```

[119] Barbarisme pour rappeler que les indices commencent à zéro.

```
; ajouter les parties 64-bit et laisser le résultat dans $f0-$f1 :
                add.d   $f0, $f2              ; slot de délai de branchement, NOP


.rodata.cst8 :000000B8 $LC0 :           .word 0x40091EB8         # DATA XREF : f+C
.rodata.cst8 :000000BC dword_BC :       .word 0x51EB851F         # DATA XREF : f+4
.rodata.cst8 :000000C0 $LC1 :           .word 0x40106666         # DATA XREF : f+10
.rodata.cst8 :000000C4 dword_C4 :       .word 0x66666666         # DATA XREF : f
```

Les nouvelles instructions ici sont:

- LWC1 charge un mot de 32-bit dans un registre du premier coprocesseur (d'où le « 1 » dans le nom de l'instruction).

  Une paire d'instructions LWC1 peut être combinée en une pseudo instruction L.D.

- DIV.D, MUL.D, ADD.D effectuent respectivement la division, la multiplication, et l'addition (« .D » est le suffixe standard pour la double précision, « .S » pour la simple précision)

Il y a une anomalie bizarre du compilateur: l'instruction LUI que nous avons marqué avec un point d'interrogation. Il m'est difficile de comprendre pourquoi charger une partie de la constante de type 64-bit *double* dans le registre $V0. Cette instruction n'a pas d'effet. Si quelqu'un en sait plus sur ceci, s'il vous plaît, envoyez moi un email[120].

1.19.6 Passage de nombres en virgule flottante par les arguments

```
#include <math.h>
#include <stdio.h>

int main ()
{
        printf ("32.01 ^ 1.54 = %lf\n", pow (32.01,1.54));

        return 0;
}
```

x86

Regardons ce que nous obtenons avec MSVC 2010:

Listing 1.205: MSVC 2010

```
CONST    SEGMENT
__real@40400147ae147ae1 DQ 040400147ae147ae1r    ; 32.01
__real@3ff8a3d70a3d70a4 DQ 03ff8a3d70a3d70a4r    ; 1.54
CONST    ENDS

_main    PROC
    push   ebp
    mov    ebp, esp
    sub    esp, 8  ; allouer de l'espace pour la première variable
    fld    QWORD PTR __real@3ff8a3d70a3d70a4
    fstp   QWORD PTR [esp]
    sub    esp, 8  ; allouer de l'espace pour la seconde variable
    fld    QWORD PTR __real@40400147ae147ae1
    fstp   QWORD PTR [esp]
    call   _pow
    add    esp, 8  ; rendre l'espace d'une variable.

; sur la pile locale, il y a ici encore 8 octets réservés pour nous.
; le résultat se trouve maintenant dans ST(0)

    fstp   QWORD PTR [esp] ; déplace le résultat de ST(0) vers la pile locale pour printf()
    push   OFFSET $SG2651
    call   _printf
```

[120] dennis@yurichev.com

```
    add    esp, 12
    xor    eax, eax
    pop    ebp
    ret    0
_main    ENDP
```

FLD et FSTP déplacent des variables entre le segment de données et la pile du FPU. pow()[121] prend deux valeurs depuis la pile et renvoie son résultat dans le registre ST(0). printf() prend 8 octets de la pile locale et les interprète comme des variables de type *double*.

À propos, une paire d'instructions MOV pourrait être utilisée ici pour déplacer les valeurs depuis la mémoire vers la pile, car les valeurs en mémoire sont stockées au format IEEE 754, et pow() les prend aussi dans ce format, donc aucune conversion n'est nécessaire. C'est fait ainsi dans l'exemple suivant, pour ARM: 1.19.6.

ARM + sans optimisation Xcode 4.6.3 (LLVM) (Mode Thumb-2)

```
_main

var_C     = -0xC

          PUSH     {R7,LR}
          MOV      R7, SP
          SUB      SP, SP, #4
          VLDR     D16, =32.01
          VMOV     R0, R1, D16
          VLDR     D16, =1.54
          VMOV     R2, R3, D16
          BLX      _pow
          VMOV     D16, R0, R1
          MOV      R0, 0xFC1 ; "32.01 ^ 1.54 = %lf\n"
          ADD      R0, PC
          VMOV     R1, R2, D16
          BLX      _printf
          MOVS     R1, 0
          STR      R0, [SP,#0xC+var_C]
          MOV      R0, R1
          ADD      SP, SP, #4
          POP      {R7,PC}

dbl_2F90  DCFD 32.01      ; DATA XREF : _main+6
dbl_2F98  DCFD 1.54       ; DATA XREF : _main+E
```

Comme nous l'avons déjà mentionné, les pointeurs sur des nombres flottants 64-bit sont passés dans une paire de R-registres.

Ce code est un peu redondant (probablement car l'optimisation est désactivée), puisqu'il est possible de charger les valeurs directement dans les R-registres sans toucher les D-registres.

Donc, comme nous le voyons, la fonction _pow reçoit son premier argument dans R0 et R1, et le second dans R2 et R3. La fonction laisse son résultat dans R0 et R1. Le résultat de _pow est déplacé dans D16, puis dans la paire R1 et R2, d'où printf() prend le nombre résultant.

ARM + sans optimisation Keil 6/2013 (Mode ARM)

```
_main
      STMFD   SP!, {R4-R6,LR}
      LDR     R2, =0xA3D70A4  ; y
      LDR     R3, =0x3FF8A3D7
      LDR     R0, =0xAE147AE1 ; x
      LDR     R1, =0x40400147
      BL      pow
      MOV     R4, R0
      MOV     R2, R4
      MOV     R3, R1
```

[121] une fonction C standard, qui élève un nombre à la puissance donnée (puissance)

```
      ADR     R0, a32_011_54Lf ; "32.01 ^ 1.54 = %lf\n"
      BL      __2printf
      MOV     R0, #0
      LDMFD   SP!, {R4-R6,PC}

y               DCD 0xA3D70A4           ; DATA XREF : _main+4
dword_520       DCD 0x3FF8A3D7          ; DATA XREF : _main+8
x               DCD 0xAE147AE1          ; DATA XREF : _main+C
dword_528       DCD 0x40400147          ; DATA XREF : _main+10
a32_011_54Lf    DCB "32.01 ^ 1.54 = %lf",0xA,0
                                        ; DATA XREF : _main+24
```

Les D-registres ne sont pas utilisés ici, juste des paires de R-registres.

ARM64 + GCC (Linaro) 4.9 avec optimisation

Listing 1.206: GCC (Linaro) 4.9 avec optimisation

```
f :
        stp     x29, x30, [sp, -16]!
        add     x29, sp, 0
        ldr     d1, .LC1 ; charger 1.54 dans D1
        ldr     d0, .LC0 ; charger 32.01 dans D0
        bl      pow
; résultat de pow() dans D0
        adrp    x0, .LC2
        add     x0, x0,  :lo12 :.LC2
        bl      printf
        mov     w0, 0
        ldp     x29, x30, [sp], 16
        ret
.LC0 :
; 32.01 au format IEEE 754
        .word   -1374389535
        .word   1077936455
.LC1 :
; 1.54 au format IEEE 754
        .word   171798692
        .word   1073259479
.LC2 :
        .string "32.01 ^ 1.54 = %lf\n"
```

Les constantes sont chargées dans `D0` et `D1` : `pow()` les prend d'ici. Le résultat sera dans `D0` après l'exécution de `pow()`. Il est passé à `printf()` sans aucune modification ni déplacement, car `printf()` prend ces arguments de type intégral et pointeurs depuis des X-registres, et les arguments en virgule flottante depuis des D-registres.

MIPS

Listing 1.207: avec optimisation GCC 4.4.5 (IDA)

```
main :

var_10          = -0x10
var_4           = -4

; prologue de la fonction :
                lui     $gp, (dword_9C >> 16)
                addiu   $sp, -0x20
                la      $gp, (__gnu_local_gp & 0xFFFF)
                sw      $ra, 0x20+var_4($sp)
                sw      $gp, 0x20+var_10($sp)
                lui     $v0, (dword_A4 >> 16) ; ?
; charger la partie 32-bit basse de 32.01:
                lwc1    $f12, dword_9C
; charger l'adresse de la fonction pow() :
```

```
                lw      $t9, (pow & 0xFFFF)($gp)
; charger la partie 32-bit haute de 32.01:
                lwc1    $f13, $LC0
                lui     $v0, ($LC1 >> 16) ; ?
; charger la partie 32-bit basse de 1.54:
                lwc1    $f14, dword_A4
                or      $at, $zero ; slot de délai de chargement, NOP
; charger la partie 32-bit haute de 1.54:
                lwc1    $f15, $LC1
; appeler pow() :
                jalr    $t9
                or      $at, $zero ; slot de délai de branchement, NOP
                lw      $gp, 0x20+var_10($sp)
; copier le résultat depuis $f0 et $f1 dans $a3 et $a2 :
                mfc1    $a3, $f0
                lw      $t9, (printf & 0xFFFF)($gp)
                mfc1    $a2, $f1
; appeler printf() :
                lui     $a0, ($LC2 >> 16)  # "32.01 ^ 1.54 = %lf\n"
                jalr    $t9
                la      $a0, ($LC2 & 0xFFFF)  # "32.01 ^ 1.54 = %lf\n"
; épilogue de la fonction :
                lw      $ra, 0x20+var_4($sp)
; renvoyer 0:
                move    $v0, $zero
                jr      $ra
                addiu   $sp, 0x20

.rodata.str1.4:00000084 $LC2 :           .ascii "32.01 ^ 1.54 = %lf\n"<0>

; 32.01:
.rodata.cst8 :00000098 $LC0 :           .word 0x40400147         # DATA XREF : main+20
.rodata.cst8 :0000009C dword_9C :       .word 0xAE147AE1         # DATA XREF : main
.rodata.cst8 :0000009C                                          # main+18
; 1.54:
.rodata.cst8 :000000A0 $LC1 :           .word 0x3FF8A3D7         # DATA XREF : main+24
.rodata.cst8 :000000A0                                          # main+30
.rodata.cst8 :000000A4 dword_A4 :       .word 0xA3D70A4          # DATA XREF : main+14
```

À nouveau, nous voyons ici LUI qui charge une partie 32-bit d'un nombre *double* dans $V0. À nouveau, c'est difficile de comprendre pourquoi.

La nouvelle instruction pour nous ici est MFC1 (« Move From Coprocessor 1 » charger depuis le coprocesseur 1). Le FPU est le coprocesseur numéro 1, d'où le « 1 » dans le nom de l'instruction. Cette instruction transfère des valeurs depuis des registres du coprocesseur dans les registres du CPU (GPR). Donc à la fin, le résultat de `pow()` est transféré dans les registres $A3 et $A2, et `printf()` prend une valeur double 64-bit depuis cette paire de registres.

1.19.7 Exemple de comparaison

```
#include <stdio.h>

double d_max (double a, double b)
{
        if (a>b)
                return a;

        return b;
};

int main()
{
        printf ("%f\n", d_max (1.2, 3.4));
        printf ("%f\n", d_max (5.6, -4));
};
```

Malgré la simplicité de la fonction, il va être difficile de comprendre comment elle fonctionne.

x86

MSVC sans optimisation

MSVC 2010 génère ce qui suit:

Listing 1.208: MSVC 2010 sans optimisation

```
PUBLIC    _d_max
_TEXT    SEGMENT
_a$ = 8                 ; size = 8
_b$ = 16                ; size = 8
_d_max     PROC
    push   ebp
    mov    ebp, esp
    fld    QWORD PTR _b$[ebp]

; état courant de la pile : ST(0) = _b
; comparer _b (ST(0)) et _a, et dépiler un registre

    fcomp  QWORD PTR _a$[ebp]

; la pile est vide ici

    fnstsw ax
    test   ah, 5
    jp     SHORT $LN1@d_max

; nous sommes ici seulement si if a>b

    fld    QWORD PTR _a$[ebp]
    jmp    SHORT $LN2@d_max
$LN1@d_max :
    fld    QWORD PTR _b$[ebp]
$LN2@d_max :
    pop    ebp
    ret    0
_d_max     ENDP
```

Ainsi, FLD charge _b dans ST(0).

FCOMP compare la valeur dans ST(0) avec ce qui est dans _a et met les bits C3/C2/C0 du mot registre d'état du FPU, suivant le résultat. Ceci est un registre 16-bit qui reflète l'état courant du FPU.

Après que les bits ont été mis, l'instruction FCOMP dépile une variable depuis la pile. C'est ce qui la différencie de FCOM, qui compare juste les valeurs, laissant la pile dans le même état.

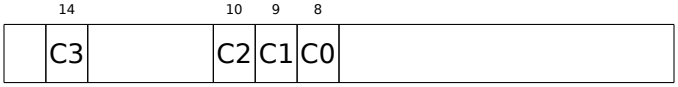
Malheureusement, les CPUs avant les Intel P6[122] ne possèdent aucune instruction de saut conditionnel qui teste les bits C3/C2/C0. Peut-être est-ce une raison historique (rappel: le FPU était une puce séparée dans le passé).
Les CPU modernes, à partir des Intel P6 possèdent les instructions FCOMI/FCOMIP/FUCOMI/FUCOMIP —qui font la même chose, mais modifient les flags ZF/PF/CF du CPU.

L'instruction FNSTSW copie le le mot du registre d'état du FPU dans AX. Les bits C3/C2/C0 sont placés aux positions 14/10/8, ils sont à la même position dans le registre AX et tous sont placés dans la partie haute de AX —AH.

- Si $b > a$ dans notre exemple, alors les bits C3/C2/C0 sont mis comme ceci: 0, 0, 0.
- Si $a > b$, alors les bits sont: 0, 0, 1.
- Si $a = b$, alors les bits sont: 1, 0, 0.

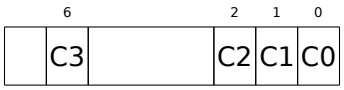
  Si le résultat n'est pas ordonné (en cas d'erreur), alors les bits sont: 1, 1, 1.

Voici comment les bits C3/C2/C0 sont situés dans le registre AX :



[122] Intel P6 comprend les Pentium Pro, Pentium II, etc.

Voici comment les bits C3/C2/C0 sont situés dans le registre AH :



Après l'exécution de `test ah, 5`[123], seul les bits C0 et C2 (en position 0 et 2) sont considérés, tous les autres bits sont simplement ignorés.

Parlons maintenant du *parity flag* (flag de parité), un autre rudiment historique remarquable.

Ce flag est mis à 1 si le nombre de un dans le résultat du dernier calcul est pair, et à 0 s'il est impair.

Regardons sur Wikipédia[124] :

> Une raison commune de tester le bit de parité n'a rien à voir avec la parité. Le FPU possède quatre flags de condition (C0 à C3), mais ils ne peuvent pas être testés directement, et doivent d'abord être copiés dans le registre d'états. Lorsque ça se produit, C0 est mis dans le flag de retenue, C2 dans le flag de parité et C3 dans le flag de zéro. Le flag C2 est mis lorsque e.g. des valeurs en virgule flottantes incomparable (NaN ou format non supporté) sont comparées avec l'instruction FUCOM.

Comme indiqué dans Wikipédia, le flag de parité est parfois utilisé dans du code FPU, voyons comment.

Le flag PF est mis à 1 si à la fois C0 et C2 sont mis à 0 ou si les deux sont à 1, auquel cas le JP (*jump if PF==1*) subséquent est déclenché. Si l'on se rappelle les valeurs de C3/C2/C0 pour différents cas, nous pouvons voir que le saut conditionnel JP est déclenché dans deux cas: si $b > a$ ou $a = b$ (le bit C3 n'est pris en considération ici, puisqu'il a été mis à 0 par l'instruction `test ah, 5`).

C'est très simple ensuite. Si le saut conditionnel a été déclenché, FLD charge la valeur de _b dans `ST(0)`, et sinon, la valeur de _a est chargée ici.

Et à propos du test de C2 ?

Le flag C2 est mis en cas d'erreur (NaN, etc.), mais notre code ne le teste pas.

Si le programmeur veut prendre en compte les erreurs FPU, il doit ajouter des tests supplémentaires.

[123] 5=101b

[124] wikipedia.org/wiki/Parity_flag

Premier exemple sous OllyDbg : a=1.2 et b=3.4

Chargeons l'exemple dans OllyDbg :

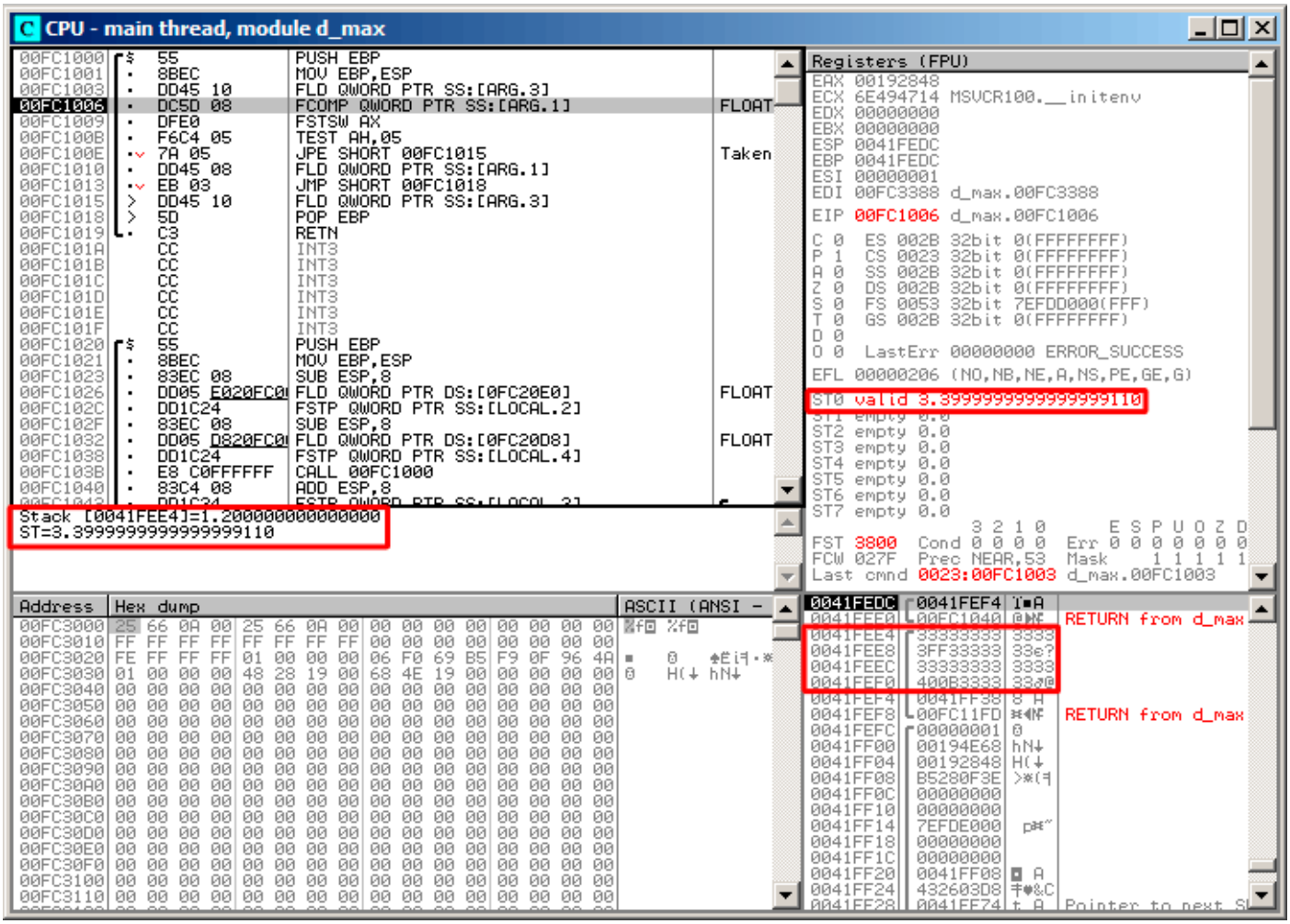


Fig. 1.67: OllyDbg : la première instruction FLD a été exécutée

Arguments courants de la fonction: $a = 1.2$ et $b = 3.4$ (Nous pouvons les voir dans la pile: deux paires de valeurs 32-bit). b (3.4) est déjà chargé dans `ST(0)`. Maintenant `FCOMP` est train d'être exécutée. OllyDbg montre le second argument de `FCOMP`, qui se trouve sur la pile à ce moment.

FCOMP a été exécutée:

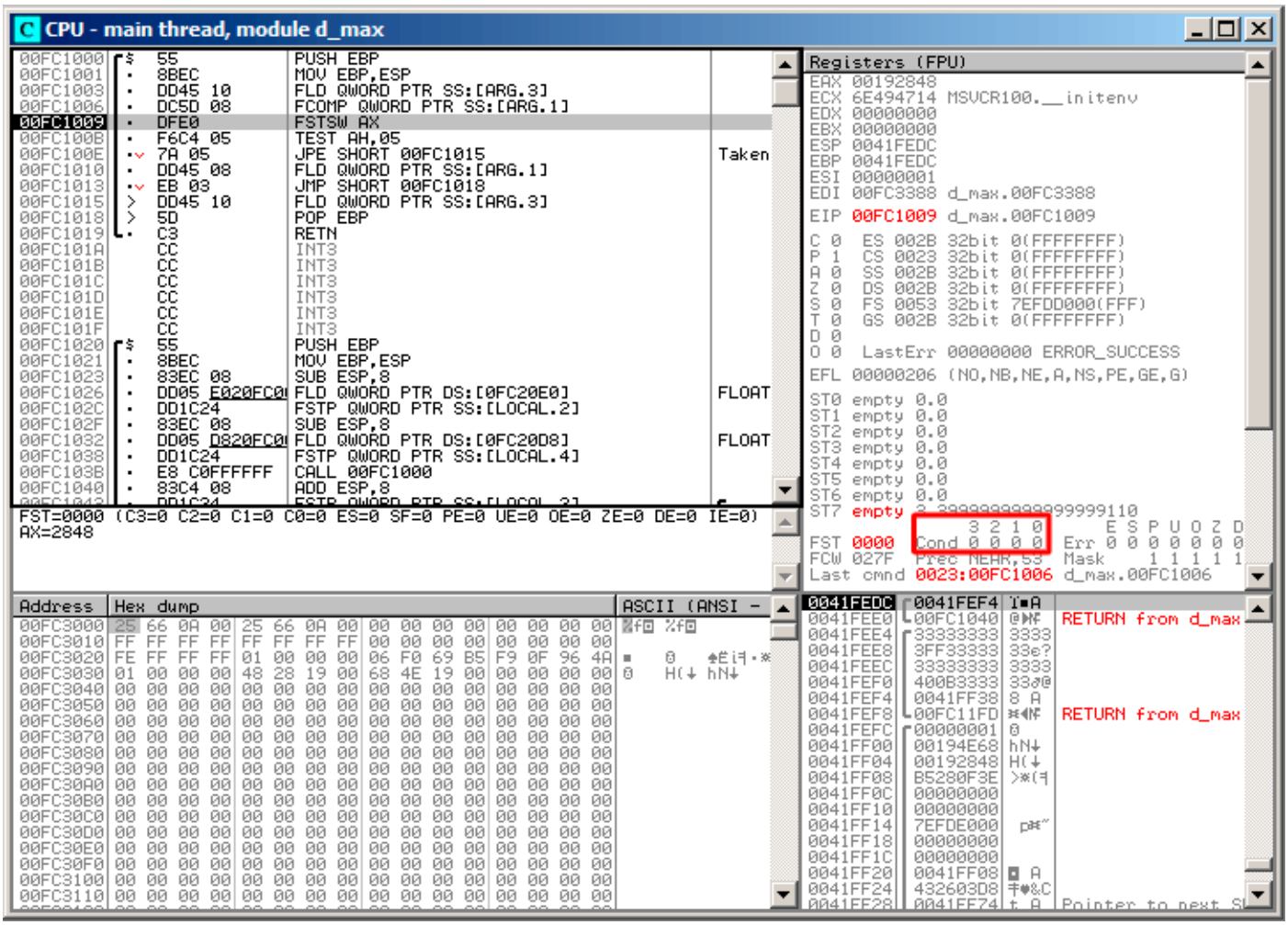


Fig. 1.68: OllyDbg : FCOMP a été exécutée

Nous voyons l'état des flags de condition du FPU : tous à zéro. La valeur dépilée est vue ici comme ST(7), la raison a été décrite ici: 1.19.5 on page 226.

FNSTSW a été exécutée:

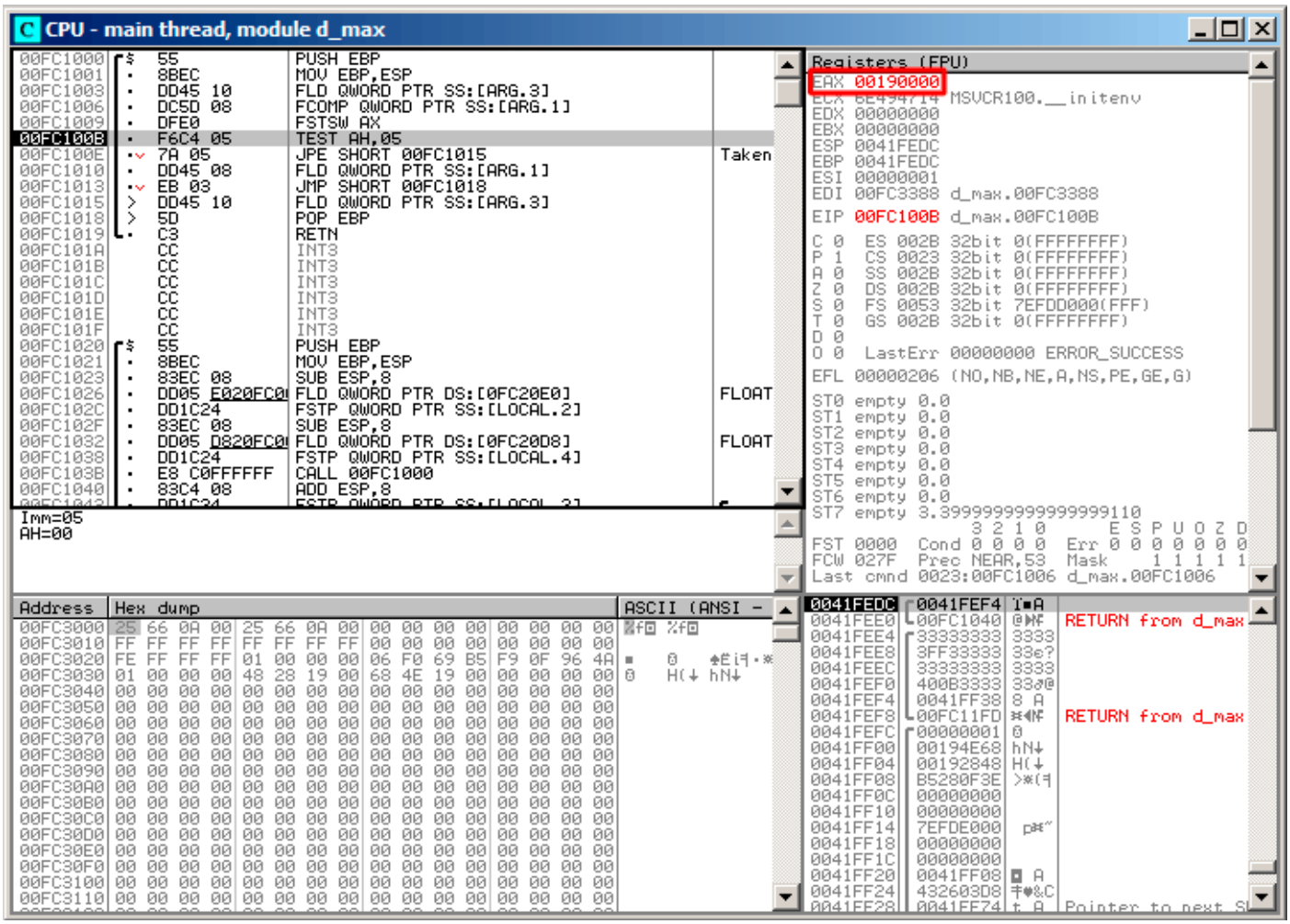


Fig. 1.69: OllyDbg : FNSTSW a été exécutée

Nous voyons que le registre AX contient des zéro: en effet, tous les flags de condition sont à zéro. (OllyDbg désassemble l'instruction FNSTSW comme FSTSW—elles sont synonymes).

TEST a été exécutée:

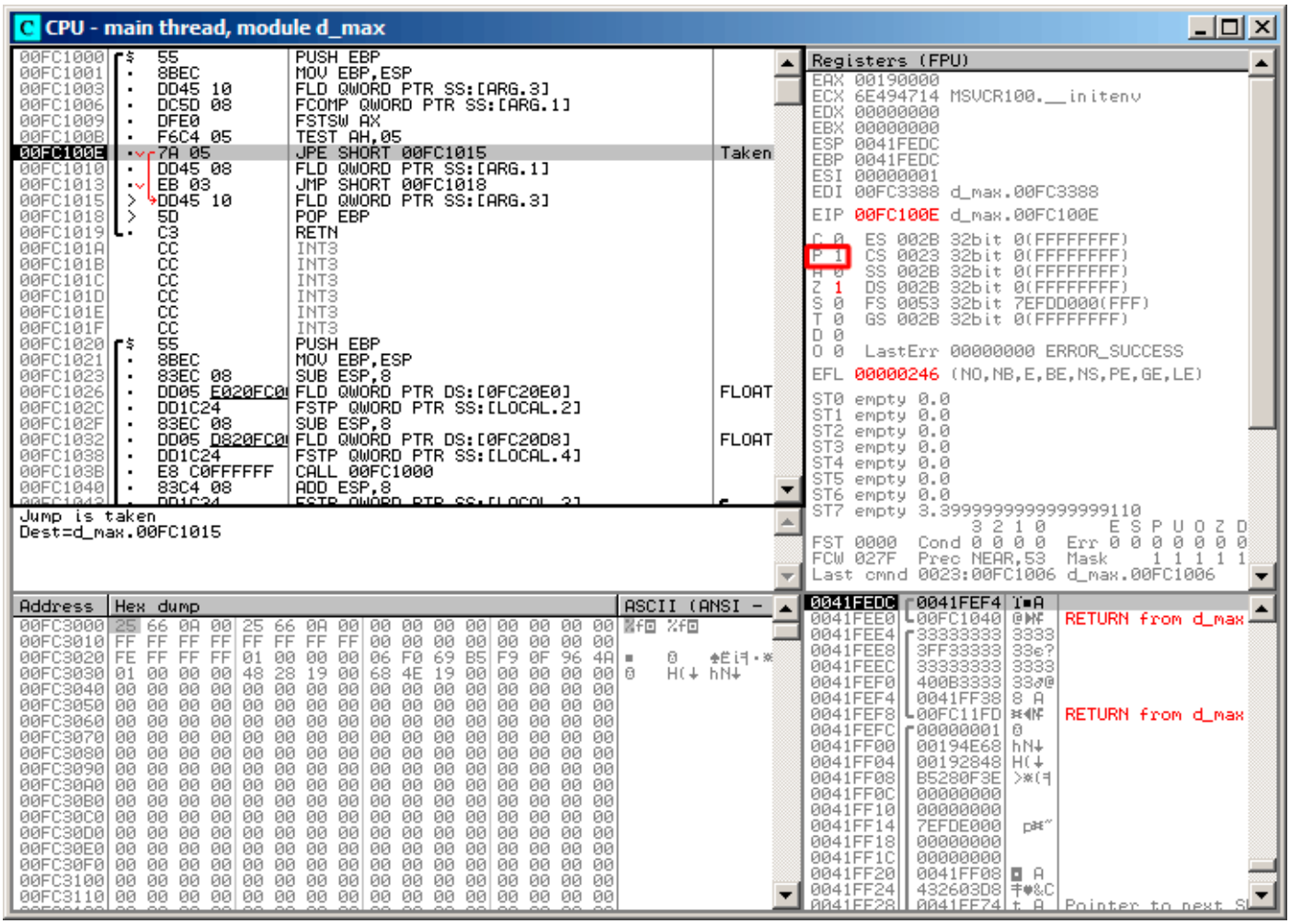


Fig. 1.70: OllyDbg : TEST a été exécutée

Le flag PF est mis à 1.

En effet: le nombre de bit mis à 0 est 0 et 0 est un nombre pair. olly désassemble l'instruction JP comme JPE[125]—elles sont synonymes. Et elle va maintenant se déclencher.

[125] Jump Parity Even (instruction x86)

JPE déclenchée, FLD charge la valeur de b (3.4) dans ST(0) :

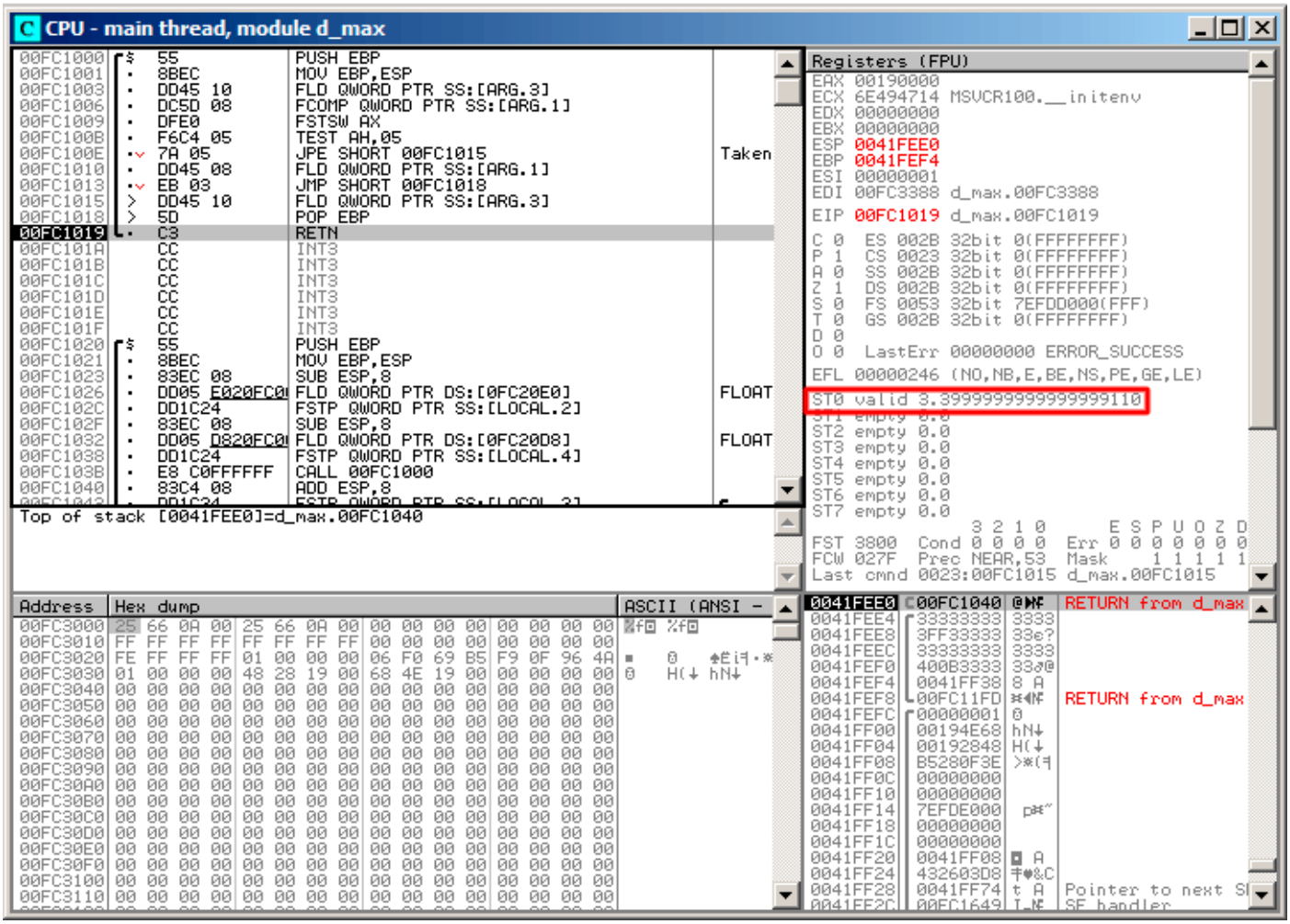


Fig. 1.71: OllyDbg : la seconde instruction FLD a été exécutée

La fonction a fini son travail.

Second exemple sous OllyDbg : a=5.6 et b=-4

Chargeons l'exemple dans OllyDbg :

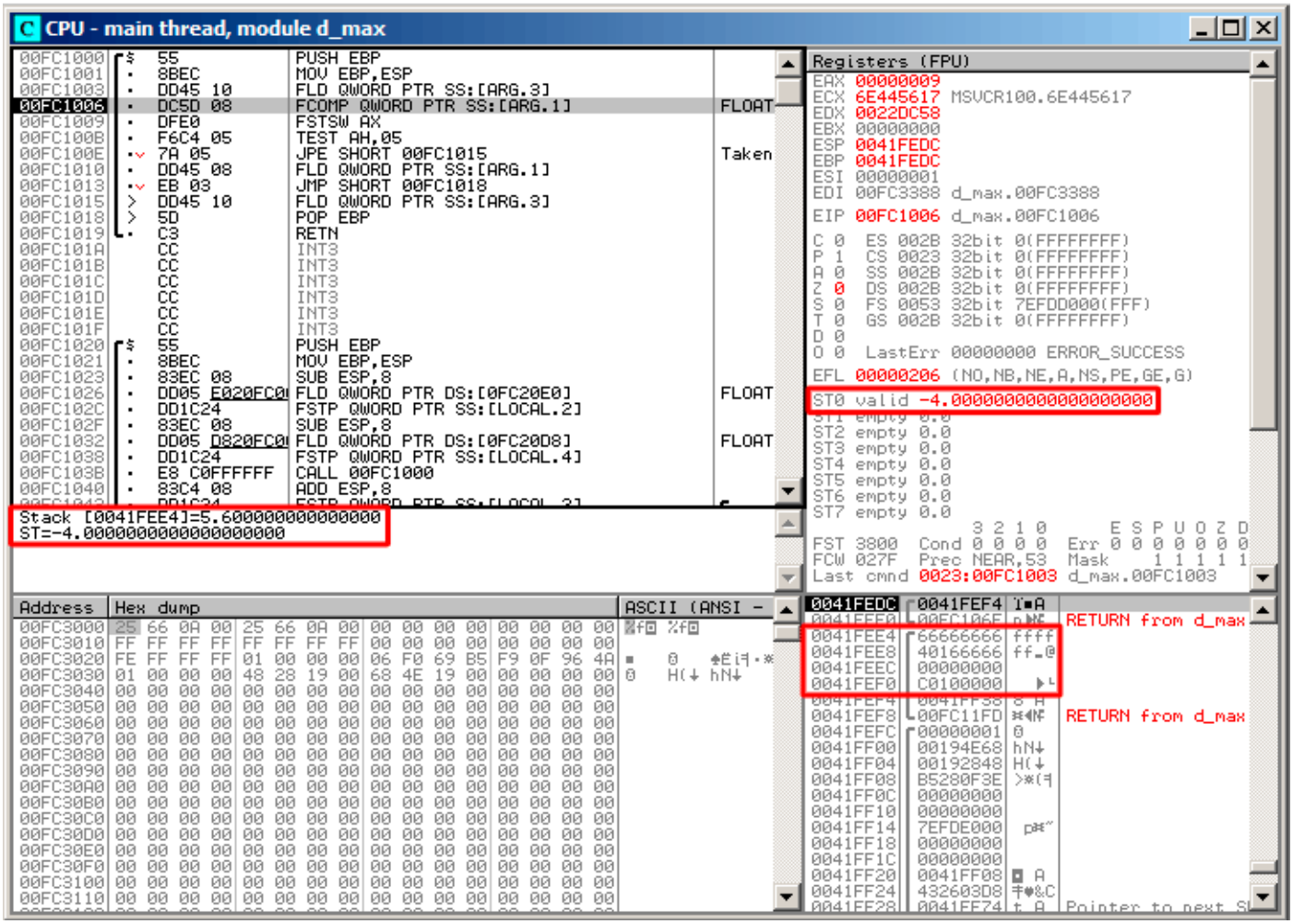


Fig. 1.72: OllyDbg : premier FLD exécutée

Arguments de la fonction courante: $a = 5.6$ et $b = -4$. b (-4) est déjà chargé dans `ST(0)`. `FCOMP` va s'exécuter maintenant. OllyDbg montre le second argument de `FCOMP`, qui est sur la pile juste maintenant.

FCOMP a été exécutée:

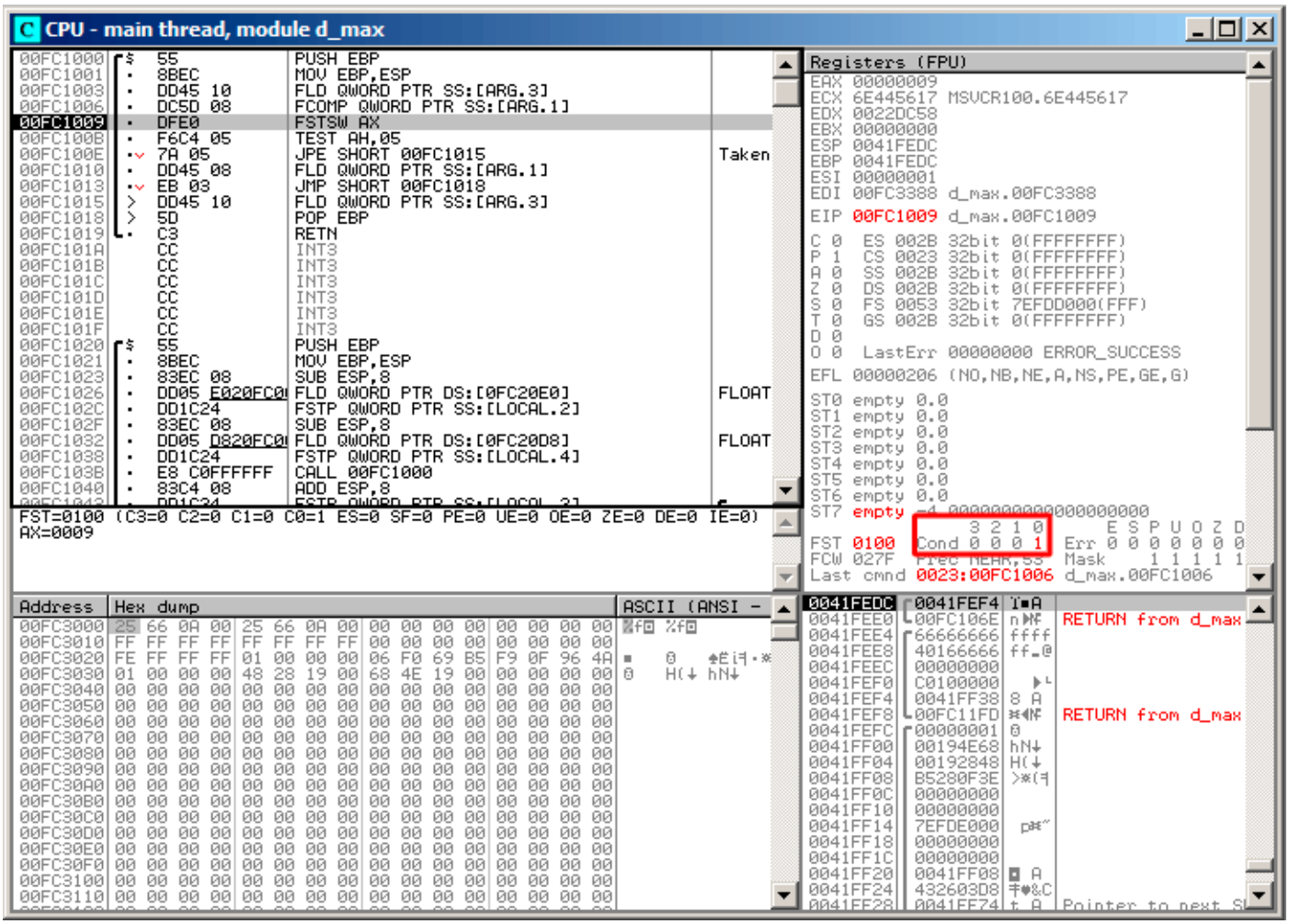


Fig. 1.73: OllyDbg : FCOMP exécutée

Nous voyons l'état des flags de condition du FPU : tous à zéro sauf C0.

FNSTSW a été exécutée:

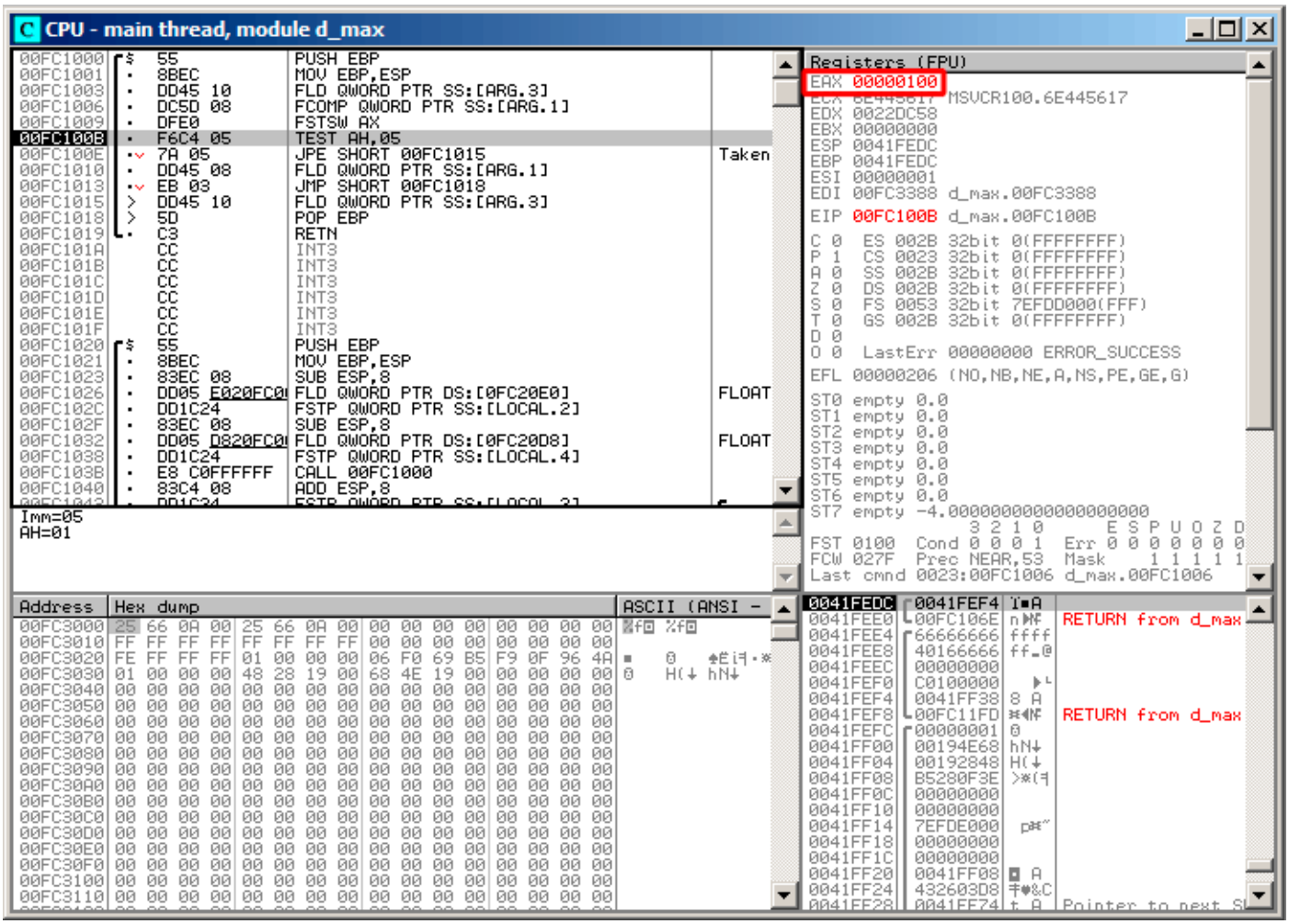


Fig. 1.74: OllyDbg : FNSTSW exécutée

Nous voyons que le registre AX contient 0x100 : le flag C0 est au 8ième bit.

TEST a été exécutée:

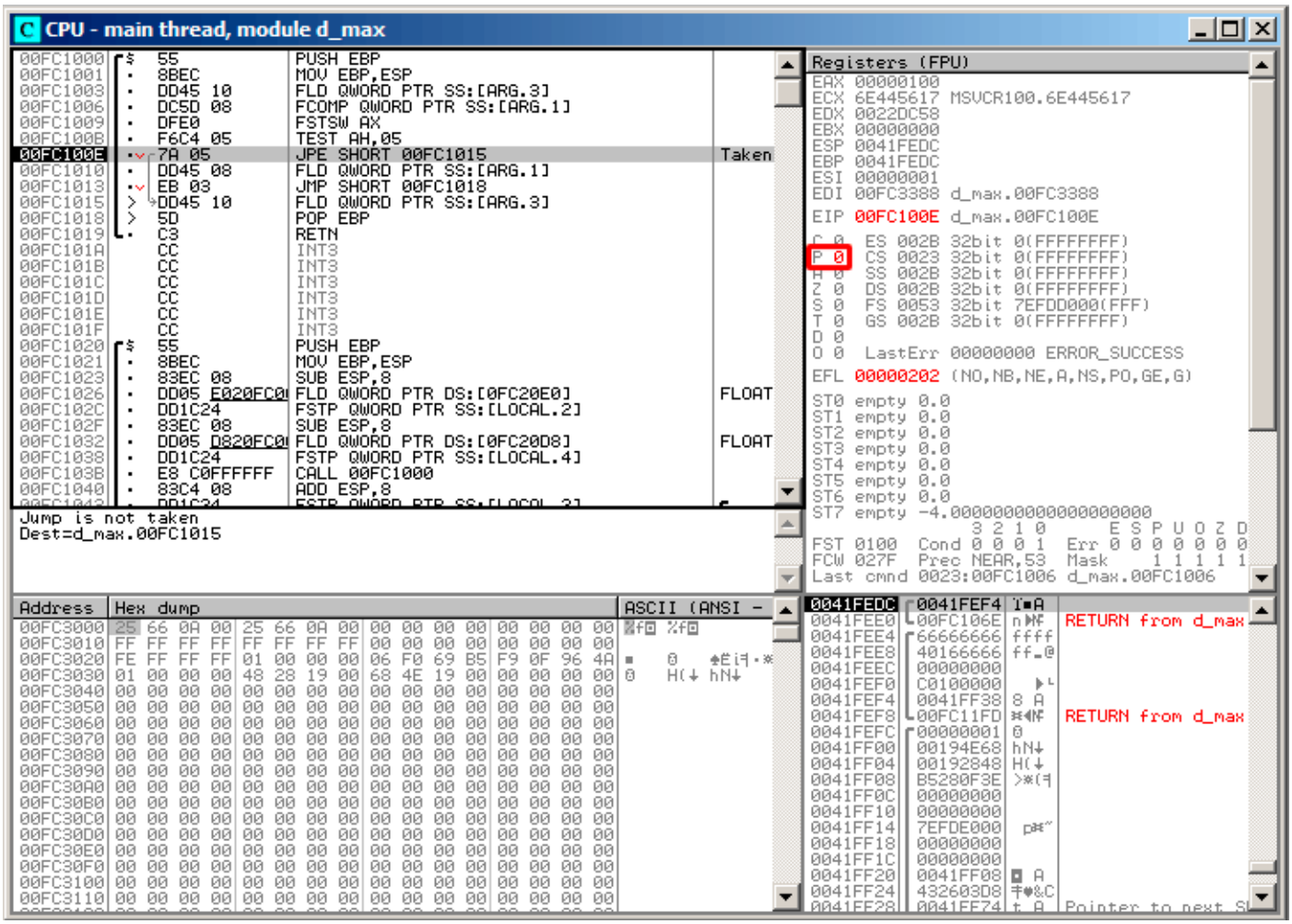


Fig. 1.75: OllyDbg : TEST exécutée

Le flag PF est mis à zéro. En effet:

le nombre de bit mis à 1 dans `0x100` est 1, et 1 est un nombre impair. JPE est sautée maintenant.

JPE n'a pas été déclenchée, donc FLD charge la valeur de a (5.6) dans ST(0) :

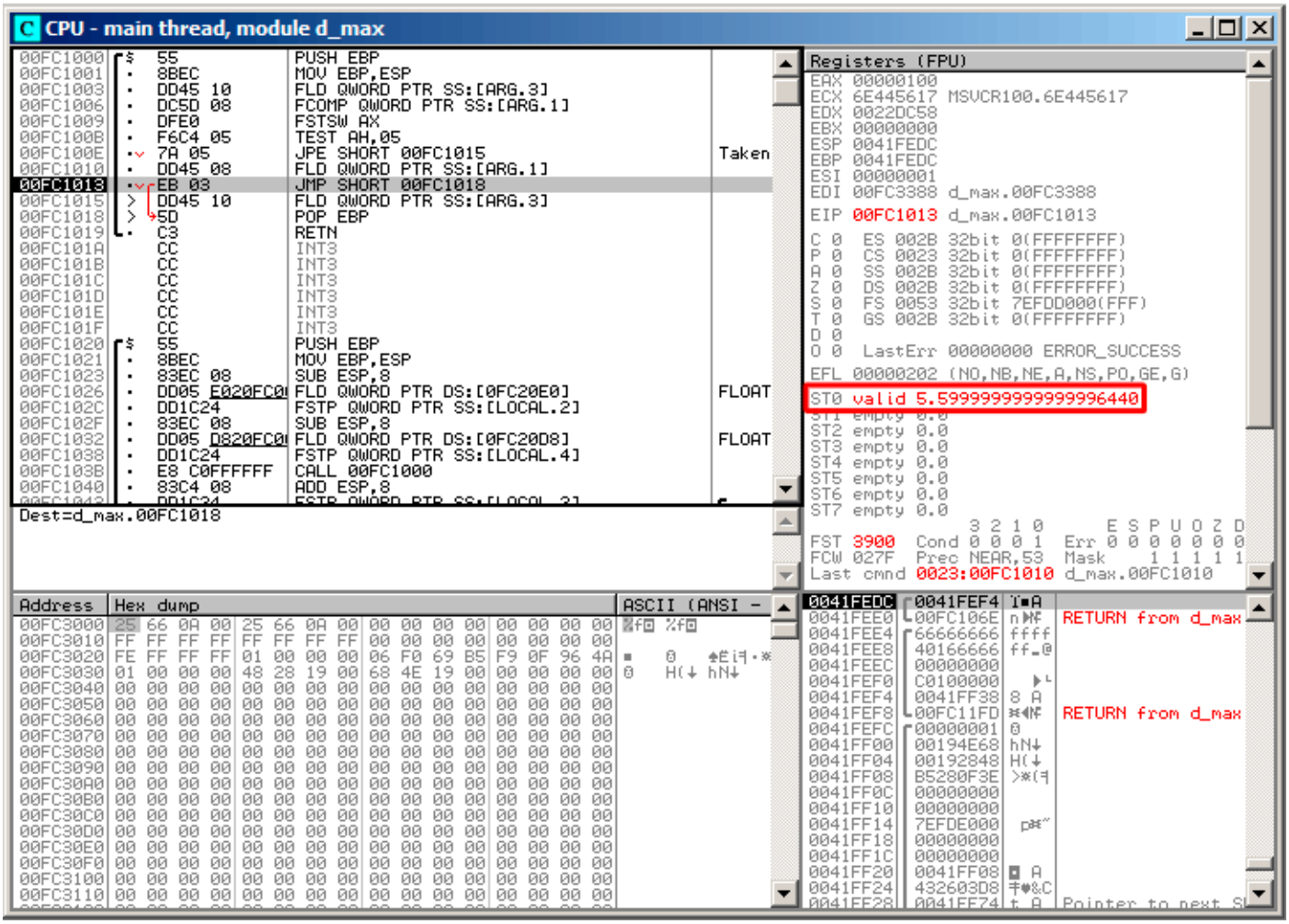


Fig. 1.76: OllyDbg : second FLD exécutée

La fonction a fini son travail.

MSVC 2010 avec optimisation

Listing 1.209: MSVC 2010 avec optimisation

```
_a$ = 8                 ; size = 8
_b$ = 16                ; size = 8
_d_max    PROC
    fld    QWORD PTR _b$[esp-4]
    fld    QWORD PTR _a$[esp-4]

; état courant de la pile : ST(0) = _a, ST(1) = _b

    fcom    ST(1) ; comparer _a et ST(1) = (_b)
    fnstsw  ax
    test    ah, 65 ; 00000041H
    jne     SHORT $LN5@d_max
; copier ST(0) dans ST(1) et dépiler le registre,
; laisser (_a) au sommet
    fstp    ST(1)

; état courant de la pile : ST(0) = _a

    ret    0
$LN5@d_max :
; copier ST(0) dans ST(0) et dépiler le registre,
```

```
; laisser (_b) au sommet
    fstp    ST(0)

; état courant de la pile : ST(0) = _b

    ret    0
_d_max    ENDP
```

FCOM diffère de FCOMP dans le sens où il compare seulement les deux valeurs, et ne change pas la pile du FPU. Contrairement à l'exemple précédent, ici les opérandes sont dans l'ordre inverse, c'est pourquoi le résultat de la comparaison dans C3/C2/C0 est différent.

- si $a > b$ dans notre exemple, alors les bits C3/C2/C0 sont mis comme suit: 0, 0, 0.
- si $b > a$, alors les bits sont: 0, 0, 1.
- si $a = b$, alors les bits sont: 1, 0, 0.

L'instruction `test ah, 65` laisse seulement deux bits —C3 et C0. Les deux seront à zéro si $a > b$: dans ce cas le saut JNE ne sera pas effectué. Puis FSTP ST(1) suit —cette instruction copie la valeur de ST(0) dans l'opérande et supprime une valeur de la pile du FPU. En d'autres mots, l'instruction copie ST(0) (où la valeur de _a se trouve) dans ST(1). Après cela, deux copies de _a sont sur le sommet de la pile. Puis, une valeur est supprimée. Après cela, ST(0) contient _a et la fonction se termine.

Le saut conditionnel JNE est effectué dans deux cas: si $b > a$ ou $a = b$. ST(0) est copié dans ST(0), c'est comme une opération sans effet (NOP), puis une valeur est supprimée de la pile et le sommet de la pile (ST(0)) contient la valeur qui était avant dans ST(1) (qui est _b). Puis la fonction se termine. La raison pour laquelle cette instruction est utilisée ici est sans doute que le FPU n'a pas d'autre instruction pour prendre une valeur sur la pile et la supprimer.

Premier exemple sous OllyDbg : a=1.2 et b=3.4

Les deux instructions FLD ont été exécutées:

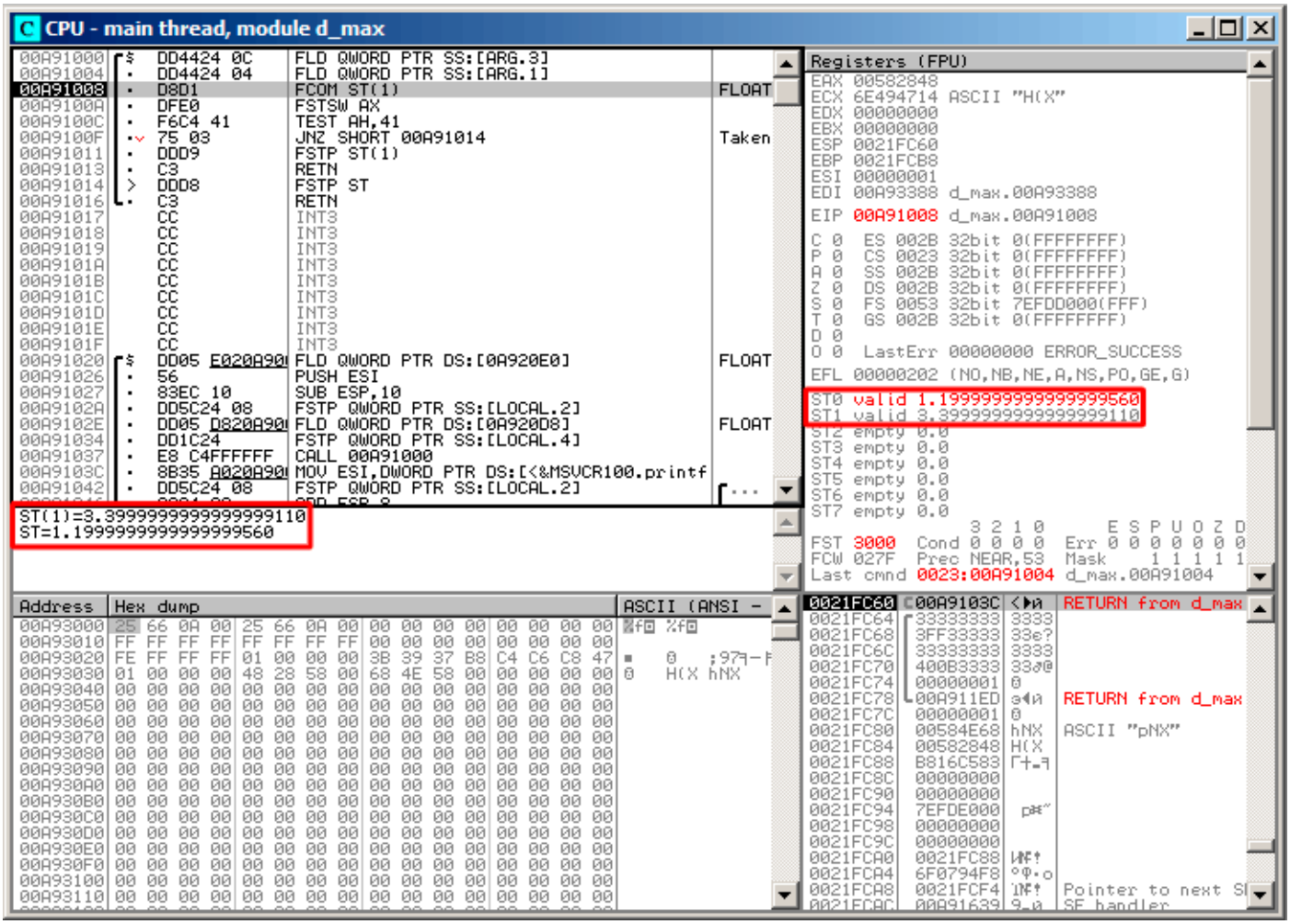


Fig. 1.77: OllyDbg : les deux FLD exécutées

FCOM exécutée: OllyDbg montre le contenu de ST(0) et ST(1) par commodité.

FCOM a été exécutée:

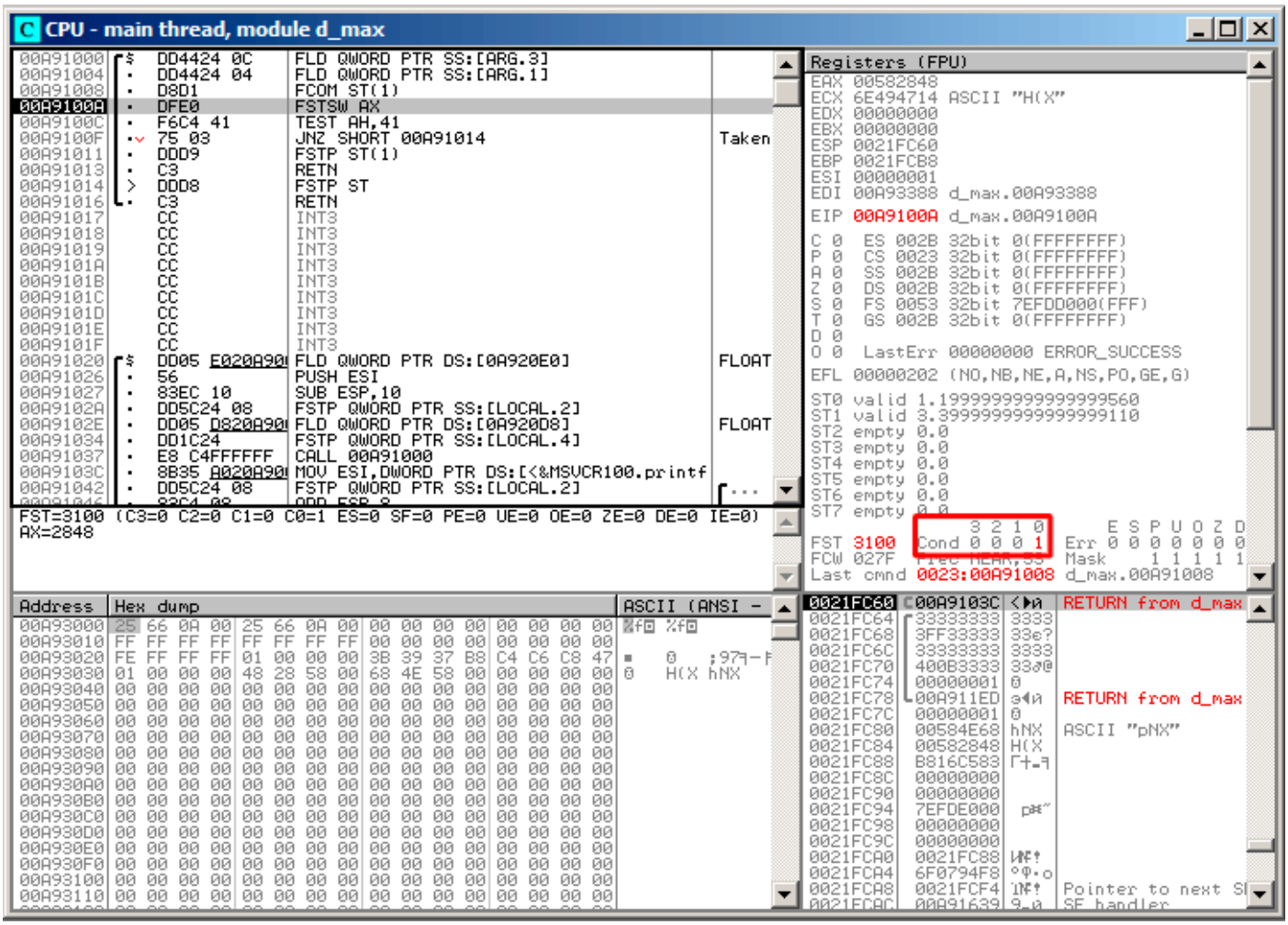


Fig. 1.78: OllyDbg : FCOM a été exécutée

C0 est mis, tous les autres flags de condition sont à zéro.

FNSTSW a été exécutée, AX=0x3100:

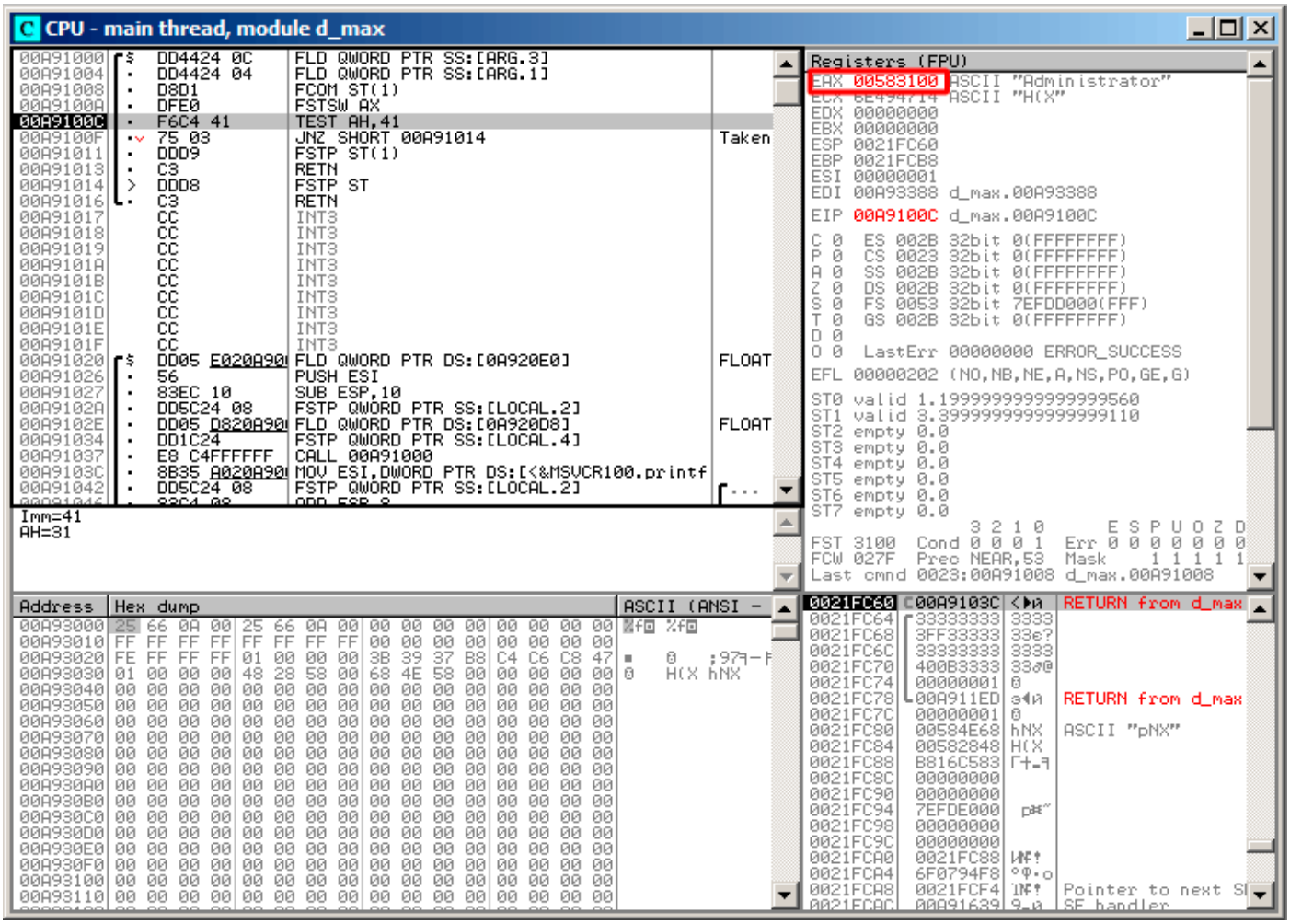


Fig. 1.79: OllyDbg : FNSTSW est exécutée

TEST est exécutée:

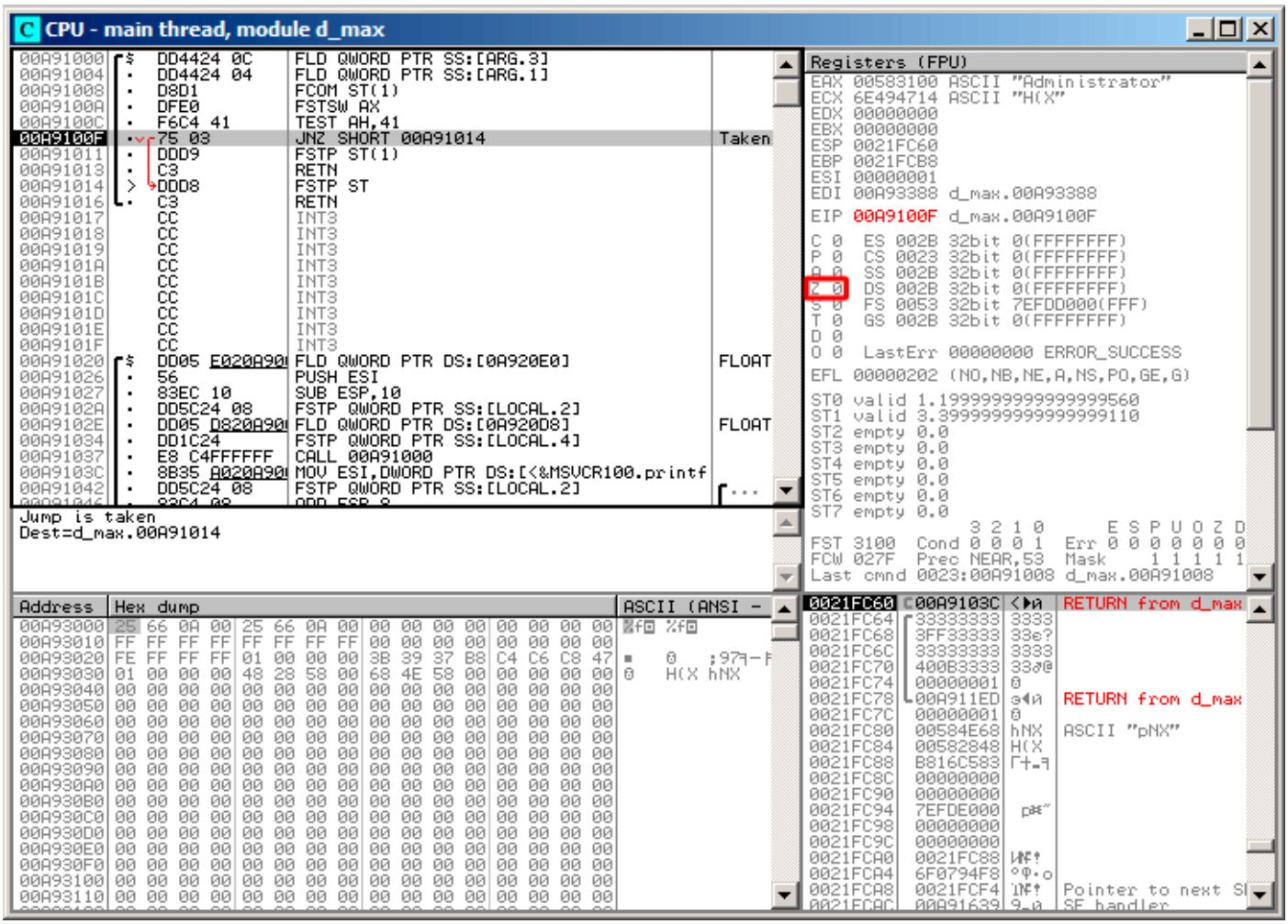


Fig. 1.80: OllyDbg : TEST est exécutée

ZF=0, le saut conditionnel va être déclenché maintenant.

FSTP ST (ou FSTP ST(0)) a été exécuté —1.2 a été dépilé, et 3.4 laissé au sommet de la pile:

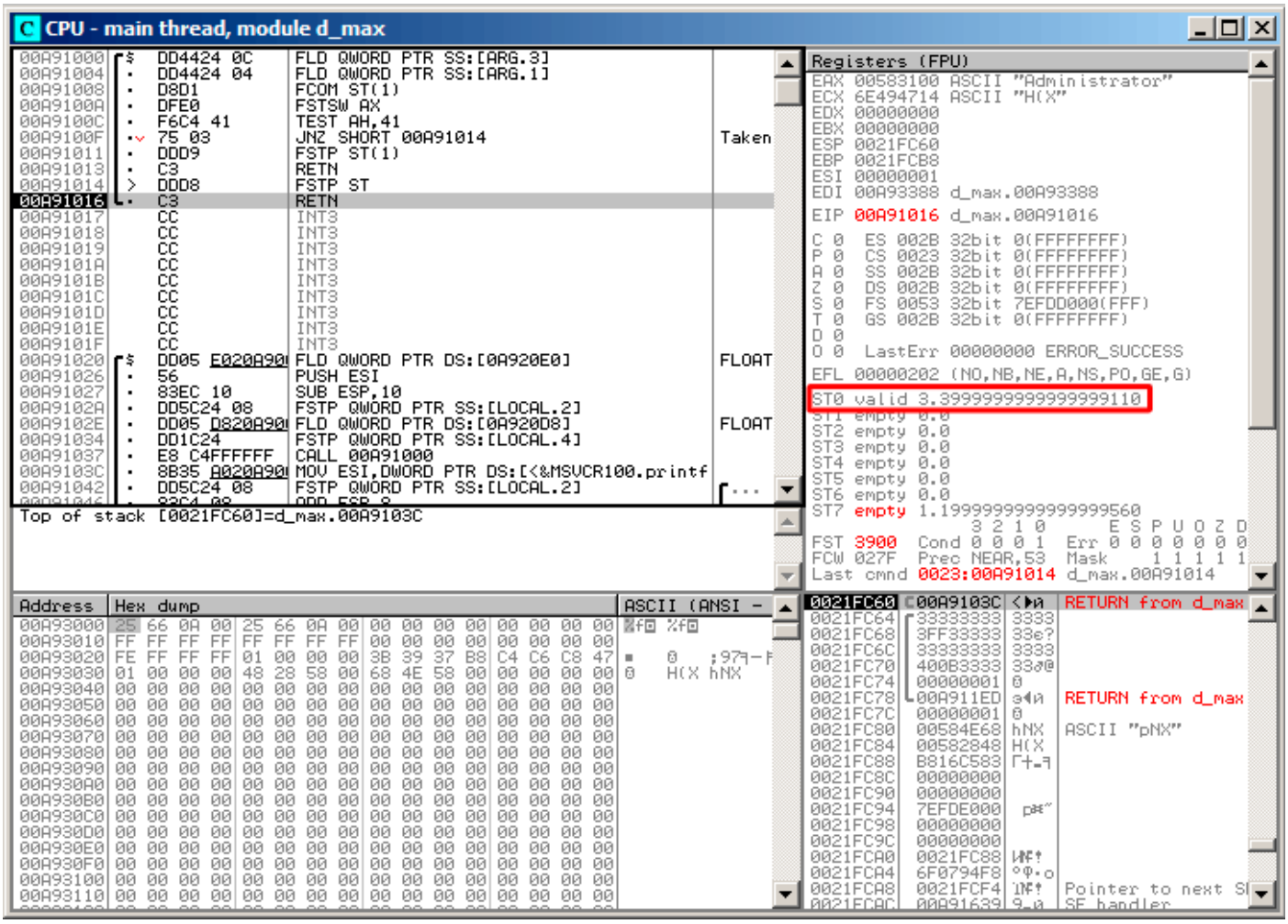


Fig. 1.81: OllyDbg : FSTP est exécutée

Nous voyons que l'instruction FSTP ST

fonctionne comme dépiler une valeur de la pile du FPU.

Second exemple sous OllyDbg : a=5.6 et b=-4

Les deux FLD sont exécutées:

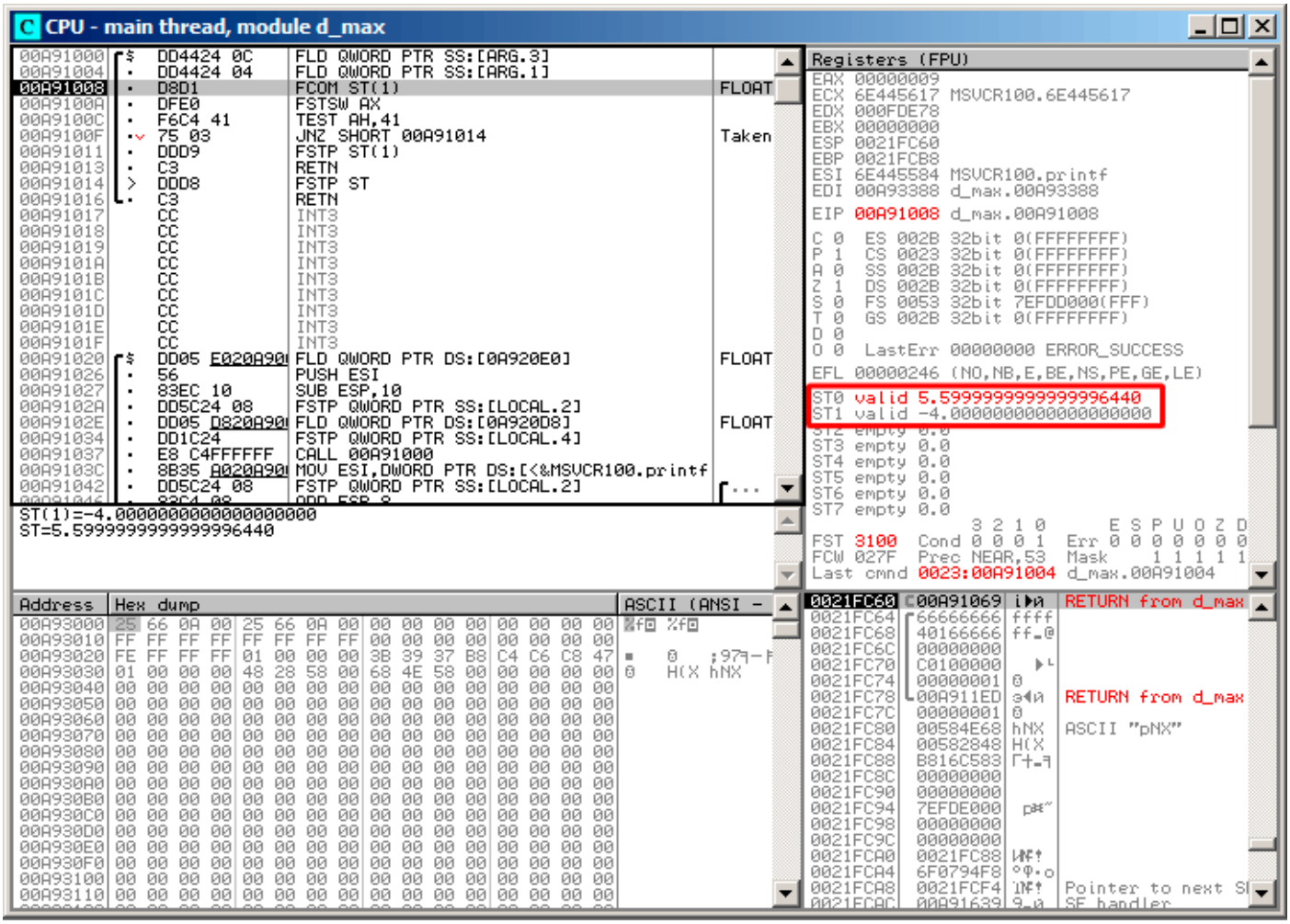


Fig. 1.82: OllyDbg : les deux FLD sont exécutée

FCOM est sur le point de s'exécuter.

FCOM a été exécutée:

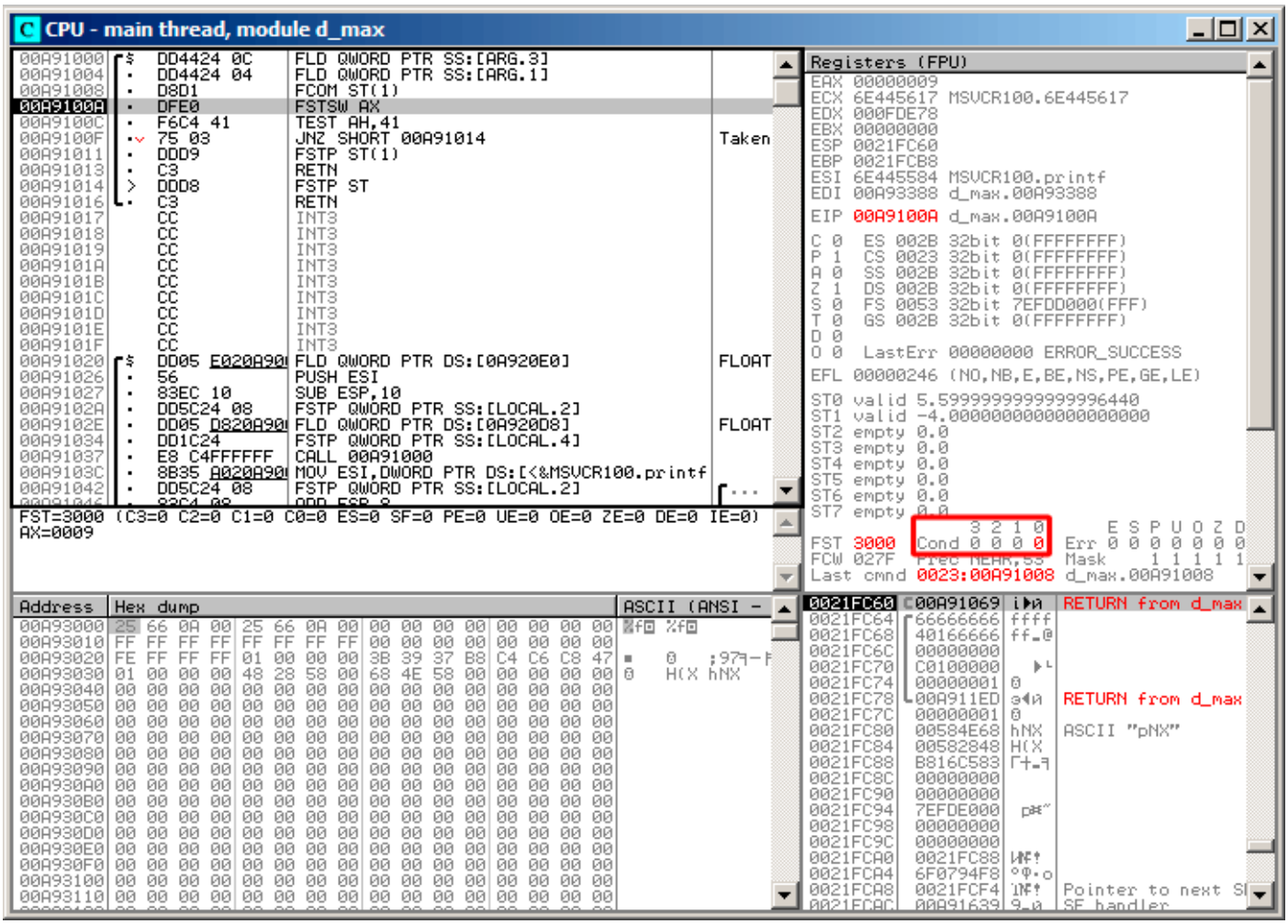


Fig. 1.83: OllyDbg : FCOM est terminé

Tous les flags de conditions sont à zéro.

FNSTSW fait, AX=0x3000:

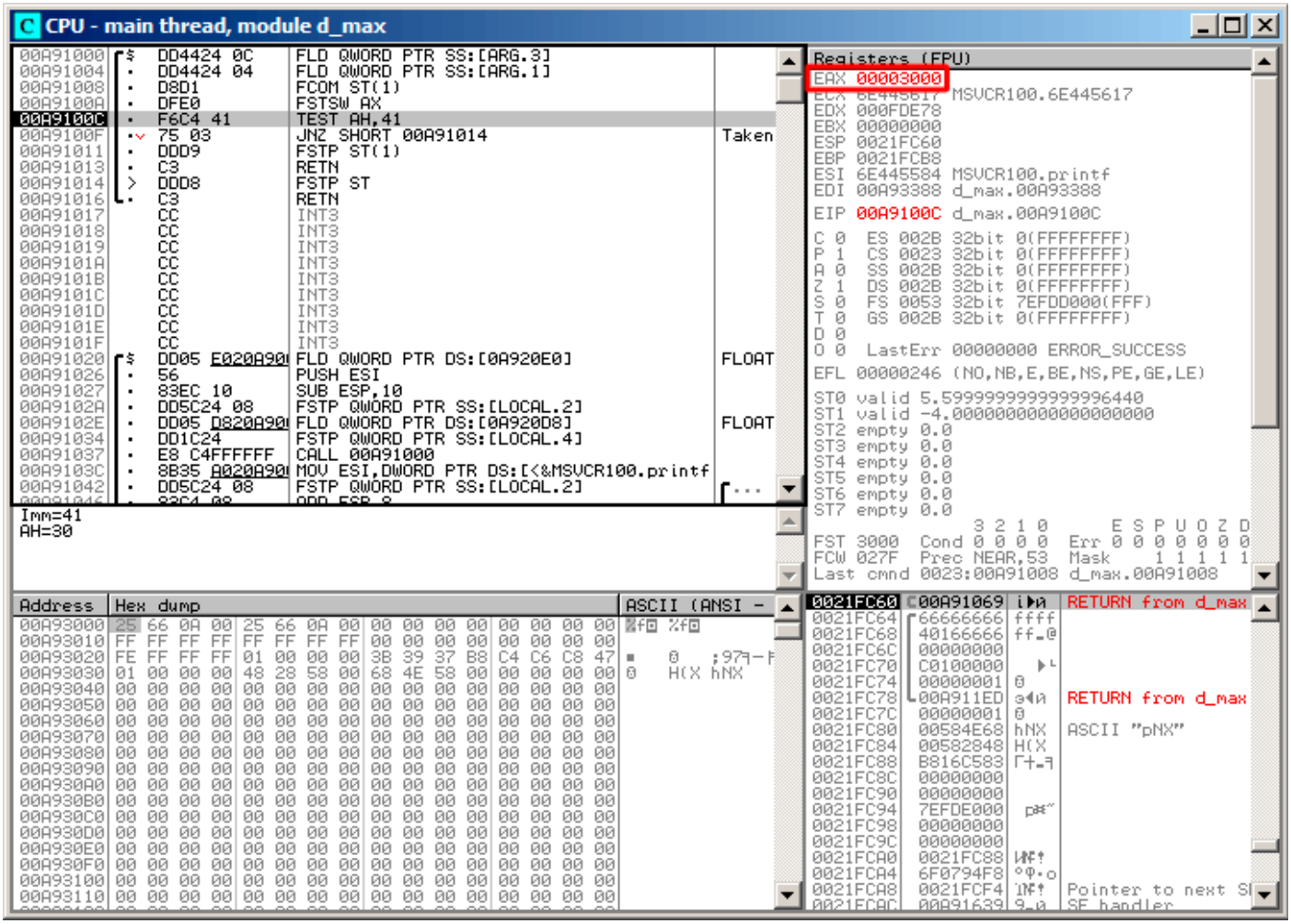


Fig. 1.84: OllyDbg : FNSTSW a été exécutée

TEST a été exécutée:

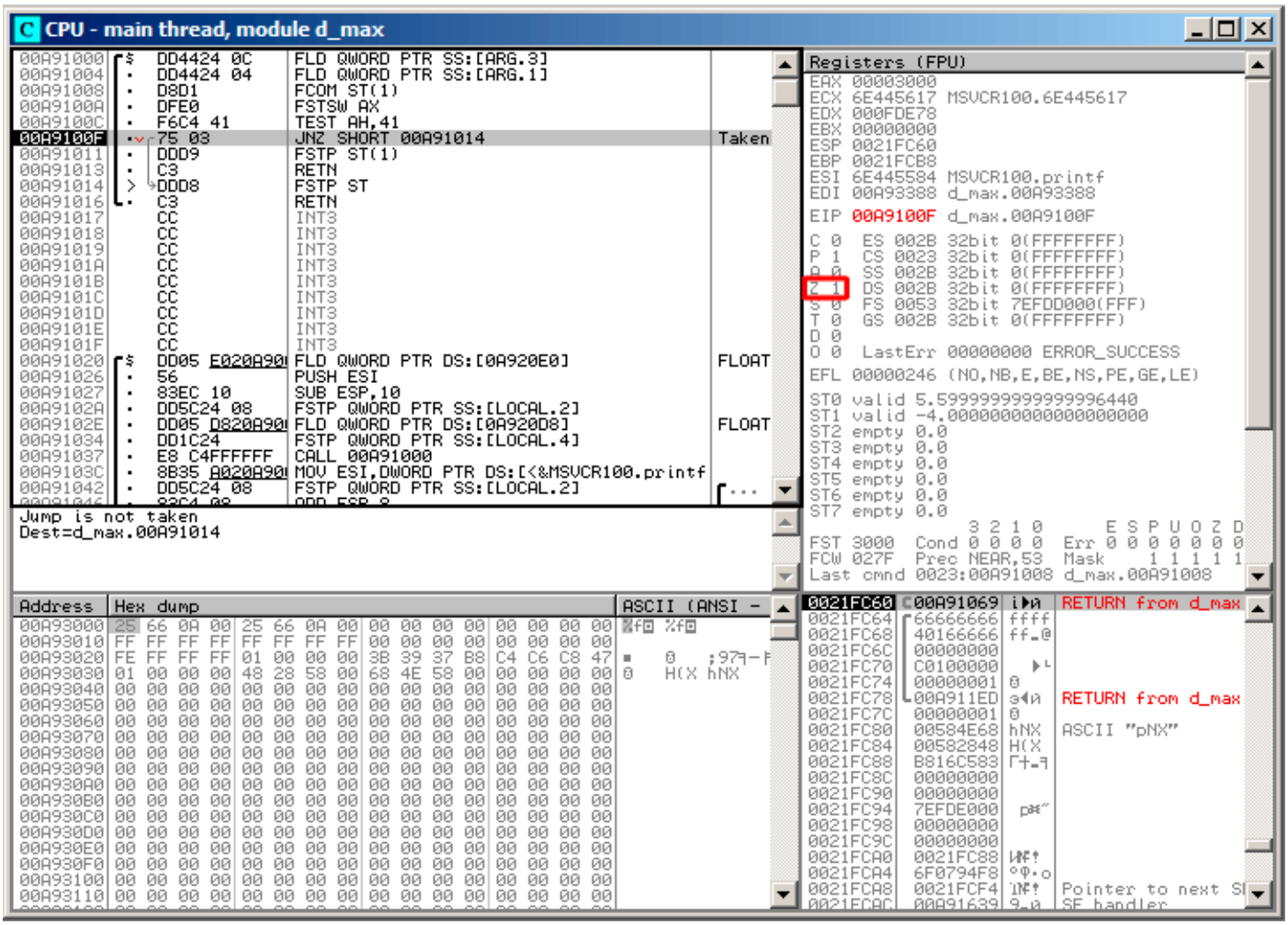


Fig. 1.85: OllyDbg : TEST a été exécutée

ZF=1, le saut ne va pas se produire maintenant.

FSTP ST(1) a été exécutée: une valeur de 5.6 est maintenant au sommet de la pile du FPU.

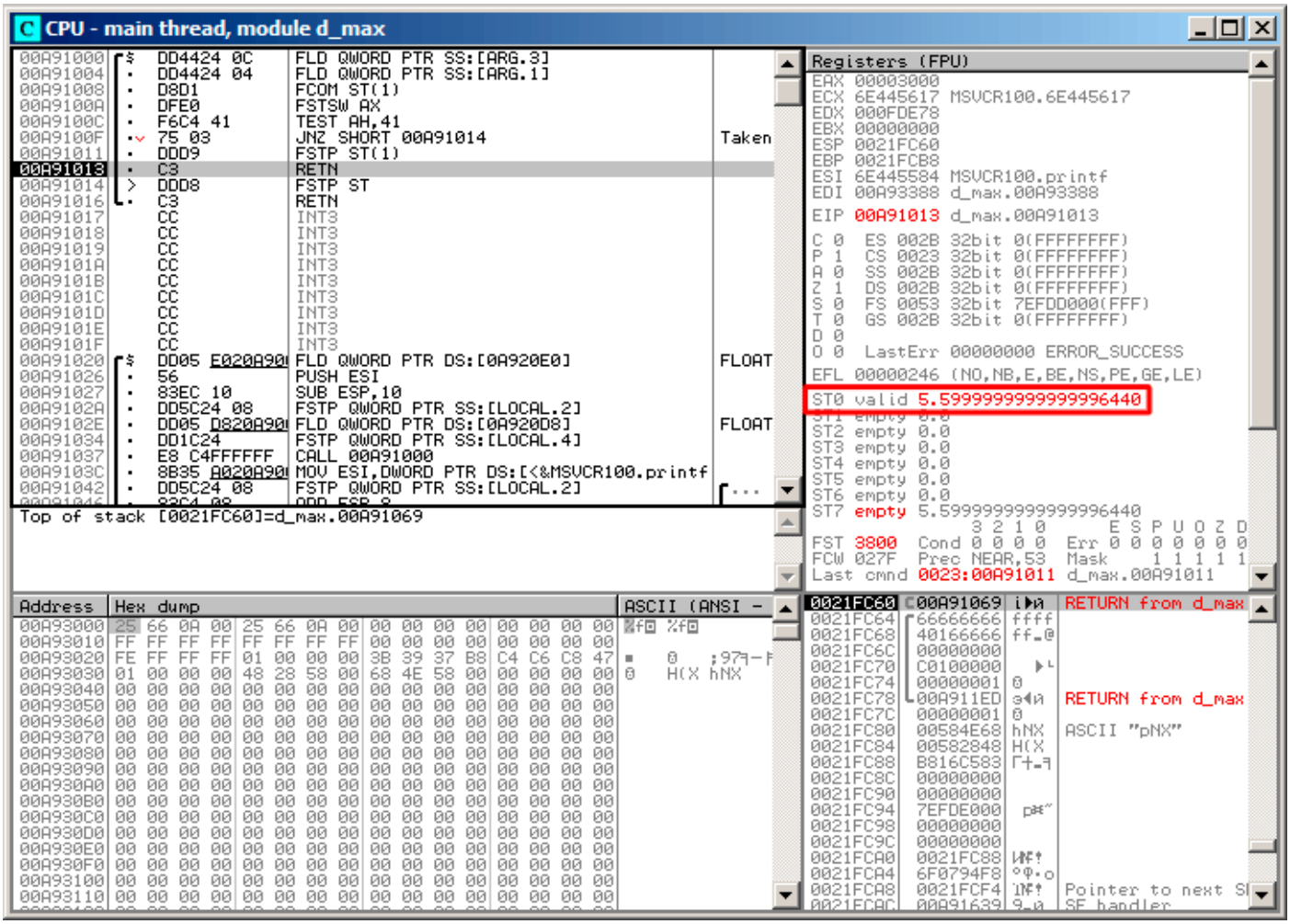


Fig. 1.86: OllyDbg : FSTP a été exécutée

Nous voyons maintenant que l'instruction FSTP ST(1) fonctionne comme suit: elle laisse ce qui était au sommet de la pile, mais met ST(1) à zéro.

GCC 4.4.1

Listing 1.210: GCC 4.4.1

```
d_max proc near

b               = qword ptr -10h
a               = qword ptr -8
a_first_half    = dword ptr  8
a_second_half   = dword ptr  0Ch
b_first_half    = dword ptr  10h
b_second_half   = dword ptr  14h

    push    ebp
    mov     ebp, esp
    sub     esp, 10h

; mettre a et b sur la pile locale :

    mov     eax, [ebp+a_first_half]
    mov     dword ptr [ebp+a], eax
    mov     eax, [ebp+a_second_half]
    mov     dword ptr [ebp+a+4], eax
    mov     eax, [ebp+b_first_half]
```

```
    mov     dword ptr [ebp+b], eax
    mov     eax, [ebp+b_second_half]
    mov     dword ptr [ebp+b+4], eax

; charger a et b sur la pile du FPU :

    fld     [ebp+a]
    fld     [ebp+b]

; état courant de la pile : ST(0) - b; ST(1) - a

    fxch    st(1) ; cette instruction échange ST(1) et ST(0)

; état courant de la pile : ST(0) - a; ST(1) - b

    fucompp     ; comparer a et b et prendre deux valeurs depuis la pile, i.e., a et b
    fnstsw  ax  ; stocker l'état du FPU dans AX
    sahf        ; charger l'état des flags SF, ZF, AF, PF, et CF depuis AH
    setnbe  al  ; mettre 1 dans AL, si CF=0 et ZF=0
    test    al, al              ; AL==0 ?
    jz      short loc_8048453 ; oui
    fld     [ebp+a]
    jmp     short locret_8048456

loc_8048453 :
    fld     [ebp+b]

locret_8048456 :
    leave
    retn
d_max endp
```

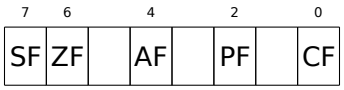
FUCOMPP est presque comme FCOM, mais dépile deux valeurs de la pile et traite les « non-nombres » différemment.

Quelques informations à propos des *not-a-numbers* (non-nombres).

Le FPU est capable de traiter les valeurs spéciales que sont les *not-a-numbers* (non-nombres) ou NaNs[126]. Ce sont les infinis, les résultat de division par 0, etc. Les non-nombres peuvent être « quiet » et « signaling ». Il est possible de continuer à travailler avec les « quiet » NaNs, mais si l'on essaye de faire une opération avec un « signaling » NaNs, une exception est levée.

FCOM lève une exception si un des opérandes est NaN. FUCOM lève une exception seulement si un des opérandes est un signaling NaN (SNaN).

L'instruction suivante est SAHF (*Store AH into Flags* stocker AH dans les Flags) —est une instruction rare dans le code non relatif au FPU. 8 bits de AH sont copiés dans les 8-bits bas dans les flags du CPU dans l'ordre suivant:



Rappelons que FNSTSW déplace des bits qui nous intéressent (C3/C2/C0) dans AH et qu'ils sont aux positions 6, 2, 0 du registre AH.

En d'autres mots, la paire d'instructions `fnstsw ax / sahf` déplace C3/C2/C0 dans ZF, PF et CF.

Maintenant, rappelons les valeurs de C3/C2/C0 sous différentes conditions:

- Si a est plus grand que b dans notre exemple, alors les C3/C2/C0 sont mis à: 0, 0, 0.
- Si a est plus petit que b, alors les bits sont mis à: 0, 0, 1.
- Si $a = b$, alors: 1, 0, 0.

En d'autres mots, ces états des flags du CPU sont possible après les trois instructions FUCOMPP/FNSTSW/SAHF :

- Si $a > b$, les flags du CPU sont mis à: ZF=0, PF=0, CF=0.

[126] wikipedia.org/wiki/NaN

- Si $a < b$, alors les flags sont mis à: ZF=0, PF=0, CF=1.
- Et si $a = b$, alors: ZF=1, PF=0, CF=0.

Suivant les flags du CPU et les conditions, SETNBE met 1 ou 0 dans AL. C'est presque la contrepartie de JNBE, avec l'exception que SETcc[127] met 1 ou 0 dans AL, mais Jcc effectue un saut ou non. SETNBE met 1 seulement si CF=0 et ZF=0. Si ce n'est pas vrai, 0 est mis dans AL.

Il y a un seul cas où CF et ZF sont à 0: si $a > b$.

Alors 1 est mis dans AL, le JZ subséquent n'est pas pris et la fonction va renvoyer _a. Dans tous les autres cas, _b est renvoyé.

GCC 4.4.1 avec optimisation

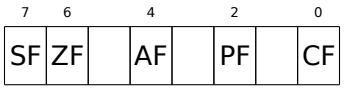
Listing 1.211: GCC 4.4.1 avec optimisation

```
                public d_max
d_max           proc near

arg_0           = qword ptr  8
arg_8           = qword ptr  10h

                push    ebp
                mov     ebp, esp
                fld     [ebp+arg_0] ; _a
                fld     [ebp+arg_8] ; _b

; état de la pile maintenant : ST(0) = _b, ST(1) = _a
                fxch    st(1)

; état de la pile maintenant : ST(0) = _a, ST(1) = _b
                fucom   st(1) ; comparer _a et _b
                fnstsw  ax
                sahf
                ja      short loc_8048448

; stocker ST(0) dans ST(0) (opération sans effet),
; dépiler une valeur du sommet de la pile,
; laisser _b au sommet
                fstp    st
                jmp     short loc_804844A

loc_8048448 :
; stocker _a dans ST(1), dépiler une valeur du sommet de la pile, laisser _a au sommet
                fstp    st(1)

loc_804844A :
                pop     ebp
                retn
d_max           endp
```

C'est presque le même, à l'exception que JA est utilisé après SAHF. En fait, les instructions de sauts conditionnels qui vérifient « plus », « moins » ou « égal » pour les comparaisons de nombres non signés (ce sont JA, JAE, JB, JBE, JE/JZ, JNA, JNAE, JNB, JNBE, JNE/JNZ) vérifient seulement les flags CF et ZF.

Rappelons comment les bits C3/C2/C0 sont situés dans le registre AH après l'exécution de FSTSW/FNSTSW :

Rappelons également, comment les bits de AH sont stockés dans les flags du CPU après l'exécution de SAHF :



[127] *cc* est un *condition code*

Après la comparaison, les bits C3 et C0 sont copiés dans ZF et CF, donc les sauts conditionnels peuvent fonctionner après. JA est déclenché si CF et ZF sont tout les deux à zéro.

Ainsi, les instructions de saut conditionnel listées ici peuvent être utilisées après une paire d'instructions FNSTSW/SAHF.

Apparemment, les bits d'état du FPU C3/C2/C0 ont été mis ici intentionnellement, pour facilement les relier aux flags du CPU de base sans permutations supplémentaires?

GCC 4.8.1 avec l'option d'optimisation -03

De nouvelles instructions FPU ont été ajoutées avec la famille Intel P6[128]. Ce sont FUCOMI (comparer les opérandes et positionner les flags du CPU principal) et FCMOVcc (fonctionne comme CMOVcc, mais avec les registres du FPU).

Apparemment, les mainteneurs de GCC ont décidé de supprimer le support des CPUs Intel pré-P6 (premier Pentium, 80486, etc.).

Et donc, le FPU n'est plus une unité séparée dans la famille Intel P6, ainsi il est possible de modifier/vérifier un flag du CPU principal depuis le FPU.

Voici ce que nous obtenons:

Listing 1.212: GCC 4.8.1 avec optimisation

```
        fld     QWORD PTR [esp+4]       ; charger "a"
        fld     QWORD PTR [esp+12]      ; charger "b"
        ; ST0=b, ST1=a
        fxch    st(1)
        ; ST0=a, ST1=b
        ; comparer "a" et "b"
        fucomi  st, st(1)
        ; copier ST1 ("b" ici) dans ST0 si a<=b
        ; laisser "a" dans ST0 autrement
        fcmovbe st, st(1)
        ; supprimer la valeur dans ST1
        fstp    st(1)
        ret
```

Difficile de deviner pourquoi FXCH (échange les opérandes) est ici.

Il est possible de s'en débarrasser facilement en échangeant les deux premières instructions FLD ou en remplaçant FCMOVBE (*below or equal* inférieur ou égal) par FCMOVA (*above*). Il s'agit probablement d'une imprécision du compilateur.

Donc FUCOMI compare ST(0) (a) et ST(1) (b) et met certains flags dans le CPU principal. FCMOVBE vérifie les flags et copie ST(1) (b ici à ce moment) dans ST(0) (a ici) si $ST0(a) <= ST1(b)$. Autrement ($a > b$), a est laissé dans ST(0).

Le dernier FSTP laisse ST(0) sur le sommet de la pile, supprimant le contenu de ST(1).

Exécutons pas à pas cette fonction dans GDB:

Listing 1.213: GCC 4.8.1 avec optimisation and GDB

```
1  dennis@ubuntuvm :~/polygon$ gcc -O3 d_max.c -o d_max -fno-inline
2  dennis@ubuntuvm :~/polygon$ gdb d_max
3  GNU gdb (GDB) 7.6.1-ubuntu
4  ...
5  Reading symbols from /home/dennis/polygon/d_max...(no debugging symbols found)...done.
6  (gdb) b d_max
7  Breakpoint 1 at 0x80484a0
8  (gdb) run
9  Starting program : /home/dennis/polygon/d_max
10
11 Breakpoint 1, 0x080484a0 in d_max ()
12 (gdb) ni
13 0x080484a4 in d_max ()
14 (gdb) disas $eip
15 Dump of assembler code for function d_max :
```

[128] À partir du Pentium Pro, Pentium-II, etc.

```
   0x080484a0 <+0> :     fldl   0x4(%esp)
=> 0x080484a4 <+4> :     fldl   0xc(%esp)
   0x080484a8 <+8> :     fxch   %st(1)
   0x080484aa <+10> :    fucomi %st(1),%st
   0x080484ac <+12> :    fcmovbe %st(1),%st
   0x080484ae <+14> :    fstp   %st(1)
   0x080484b0 <+16> :    ret
End of assembler dump.
(gdb) ni
0x080484a8 in d_max ()
(gdb) info float
  R7 : Valid   0x3fff9999999999999800 +1.199999999999999956
=>R6 : Valid   0x4000d999999999999800 +3.399999999999999911
  R5 : Empty   0x00000000000000000000
  R4 : Empty   0x00000000000000000000
  R3 : Empty   0x00000000000000000000
  R2 : Empty   0x00000000000000000000
  R1 : Empty   0x00000000000000000000
  R0 : Empty   0x00000000000000000000

Status Word :         0x3000
                       TOP : 6
Control Word :        0x037f   IM DM ZM OM UM PM
                       PC : Extended Precision (64-bits)
                       RC : Round to nearest
Tag Word :            0x0fff
Instruction Pointer : 0x73 :0x080484a4
Operand Pointer :     0x7b :0xbffff118
Opcode :              0x0000
(gdb) ni
0x080484aa in d_max ()
(gdb) info float
  R7 : Valid   0x4000d999999999999800 +3.399999999999999911
=>R6 : Valid   0x3fff9999999999999800 +1.199999999999999956
  R5 : Empty   0x00000000000000000000
  R4 : Empty   0x00000000000000000000
  R3 : Empty   0x00000000000000000000
  R2 : Empty   0x00000000000000000000
  R1 : Empty   0x00000000000000000000
  R0 : Empty   0x00000000000000000000

Status Word :         0x3000
                       TOP : 6
Control Word :        0x037f   IM DM ZM OM UM PM
                       PC : Extended Precision (64-bits)
                       RC : Round to nearest
Tag Word :            0x0fff
Instruction Pointer : 0x73 :0x080484a8
Operand Pointer :     0x7b :0xbffff118
Opcode :              0x0000
(gdb) disas $eip
Dump of assembler code for function d_max :
   0x080484a0 <+0> :     fldl   0x4(%esp)
   0x080484a4 <+4> :     fldl   0xc(%esp)
   0x080484a8 <+8> :     fxch   %st(1)
=> 0x080484aa <+10> :    fucomi %st(1),%st
   0x080484ac <+12> :    fcmovbe %st(1),%st
   0x080484ae <+14> :    fstp   %st(1)
   0x080484b0 <+16> :    ret
End of assembler dump.
(gdb) ni
0x080484ac in d_max ()
(gdb) info registers
eax            0x1      1
ecx            0xbffff1c4       -1073745468
edx            0x8048340        134513472
ebx            0xb7fbf000       -1208225792
esp            0xbffff10c       0xbffff10c
ebp            0xbffff128       0xbffff128
esi            0x0      0
```

```
edi            0x0      0
eip            0x80484ac        0x80484ac <d_max+12>
eflags         0x203    [ CF IF ]
cs             0x73     115
ss             0x7b     123
ds             0x7b     123
es             0x7b     123
fs             0x0      0
gs             0x33     51
(gdb) ni
0x080484ae in d_max ()
(gdb) info float
  R7 : Valid   0x4000d999999999999800 +3.399999999999999911
=>R6 : Valid   0x4000d999999999999800 +3.399999999999999911
  R5 : Empty   0x00000000000000000000
  R4 : Empty   0x00000000000000000000
  R3 : Empty   0x00000000000000000000
  R2 : Empty   0x00000000000000000000
  R1 : Empty   0x00000000000000000000
  R0 : Empty   0x00000000000000000000

Status Word :         0x3000
                       TOP : 6
Control Word :        0x037f   IM DM ZM OM UM PM
                       PC : Extended Precision (64-bits)
                       RC : Round to nearest
Tag Word :            0x0fff
Instruction Pointer : 0x73 :0x080484ac
Operand Pointer :     0x7b :0xbffff118
Opcode :              0x0000
(gdb) disas $eip
Dump of assembler code for function d_max :
   0x080484a0 <+0> :     fldl   0x4(%esp)
   0x080484a4 <+4> :     fldl   0xc(%esp)
   0x080484a8 <+8> :     fxch   %st(1)
   0x080484aa <+10> :    fucomi %st(1),%st
   0x080484ac <+12> :    fcmovbe %st(1),%st
=> 0x080484ae <+14> :    fstp   %st(1)
   0x080484b0 <+16> :    ret
End of assembler dump.
(gdb) ni
0x080484b0 in d_max ()
(gdb) info float
=>R7 : Valid   0x4000d999999999999800 +3.399999999999999911
  R6 : Empty   0x4000d999999999999800
  R5 : Empty   0x00000000000000000000
  R4 : Empty   0x00000000000000000000
  R3 : Empty   0x00000000000000000000
  R2 : Empty   0x00000000000000000000
  R1 : Empty   0x00000000000000000000
  R0 : Empty   0x00000000000000000000

Status Word :         0x3800
                       TOP : 7
Control Word :        0x037f   IM DM ZM OM UM PM
                       PC : Extended Precision (64-bits)
                       RC : Round to nearest
Tag Word :            0x3fff
Instruction Pointer : 0x73 :0x080484ae
Operand Pointer :     0x7b :0xbffff118
Opcode :              0x0000
(gdb) quit
A debugging session is active.

        Inferior 1 [process 30194] will be killed.

Quit anyway? (y or n) y
dennis@ubuntuvm :~/polygon$
```

En utilisant « ni », exécutons les deux premières instructions FLD.

Examinons les registres du FPU (ligne 33).

Comme cela a déjà été mentionné, l'ensemble des registres FPU est un buffeur circulaire plutôt qu'une pile (1.19.5 on page 226). Et GDB ne montre pas les registres STx, mais les registre internes du FPU (Rx). La flèche (à la ligne 35) pointe sur le haut courant de la pile.

Vous pouvez voir le contenu du registre TOP dans le *Status Word* (ligne 44)—c'est 6 maintenant, donc le haut de la pile pointe maintenant sur le registre interne 6.

Les valeurs de a et b sont échangées après l'exécution de FXCH (ligne 54).

FUCOMI est exécuté (ilgne 83). Regardons les flags: CF est mis (ligne 95).

FCMOVBE a copié la valeur de b (voir ligne 104).

FSTP dépose une valeur au sommet de la pile (ligne 136). La valeur de TOP est maintenant 7, donc le sommet de la pile du FPU pointe sur le registre interne 7.

ARM

avec optimisation Xcode 4.6.3 (LLVM) (Mode ARM)

Listing 1.214: avec optimisation Xcode 4.6.3 (LLVM) (Mode ARM)

```
VMOV            D16, R2, R3 ; b
VMOV            D17, R0, R1 ; a
VCMPE.F64       D17, D16
VMRS            APSR_nzcv, FPSCR
VMOVGT.F64      D16, D17 ; copier "a" dans D16
VMOV            R0, R1, D16
BX              LR
```

Un cas très simple. Les valeurs en entrée sont placées dans les registres D17 et D16 puis comparées en utilisant l'instruction VCMPE.

Tout comme dans le coprocesseur x86, le coprocesseur ARM a son propre registre de flags (FPSCR[129]), puisqu'il est nécessaire de stocker des flags spécifique au coprocesseur. Et tout comme en x86, il n'y a pas d'instruction de saut conditionnel qui teste des bits dans le registre de status du coprocesseur. Donc il y a VMRS, qui copie 4 bits (N, Z, C, V) du mot d'état du coprocesseur dans les bits du registre de status *général* (APSR[130]).

VMOVGT est l'analogue de l'instruction MOVGT pour D-registres, elle s'exécute si un opérande est plus grand que l'autre lors de la comparaison (*GT—Greater Than*).

Si elle est exécutée, la valeur de a sera écrite dans D16 (ce qui est écrit en ce moment dans D17). Sinon, la valeur de b reste dans le registre D16.

La pénultième instruction VMOV prépare la valeur dans la registre D16 afin de la renvoyer dans la paire de registres R0 et R1.

avec optimisation Xcode 4.6.3 (LLVM) (Mode Thumb-2)

Listing 1.215: avec optimisation Xcode 4.6.3 (LLVM) (Mode Thumb-2)

```
VMOV            D16, R2, R3 ; b
VMOV            D17, R0, R1 ; a
VCMPE.F64       D17, D16
VMRS            APSR_nzcv, FPSCR
IT GT
VMOVGT.F64      D16, D17
VMOV            R0, R1, D16
BX              LR
```

[129](ARM) Floating-Point Status and Control Register
[130](ARM) Application Program Status Register

Presque comme dans l'exemple précédent, toutefois légèrement différent. Comme nous le savons déjà, en mode ARM, beaucoup d'instructions peuvent avoir un prédicat de condition. Mais il n'y a rien de tel en mode Thumb. Il n'y a pas d'espace dans les instructions sur 16-bit pour 4 bits dans lesquels serait encodée la condition.

Toutefois, cela à été étendu en un mode Thumb-2 pour rendre possible de spécifier un prédicat aux instructions de l'ancien mode Thumb. Ici, dans le listing généré par IDA, nous voyons l'instruction VMOVGT, comme dans l'exemple précédent.

En fait, le VMOV usuel est encodé ici, mais IDA lui ajoute le suffixe -GT, puisque que l'instruction IT GT se trouve juste avant.

L'instruction IT défini ce que l'on appelle un *bloc if-then*.

Après cette instruction, il est possible de mettre jusqu'à 4 instructions, chacune d'entre elles ayant un suffixe de prédicat. Dans notre exemple, IT GT implique que l'instruction suivante ne sera exécutée que si la condition *GT* (*Greater Than* plus grand que) est vraie.

Voici un exemple de code plus complexe, à propos, d'Angry Birds (pour iOS):

Listing 1.216: Angry Birds Classic

```
...
ITE NE
VMOVNE          R2, R3, D16
VMOVEQ          R2, R3, D17
BLX             _objc_msgSend ; not suffixed
...
```

ITE est l'acronyme de *if-then-else* et elle encode un suffixe pour les deux prochaines instructions.

La première instruction est exécutée si la condition encodée dans ITE (*NE, not equal*) est vraie, et la seconde—si la condition n'est pas vraie (l'inverse de la condition NE est EQ (*equal*)).

L'instruction qui suit le second VMOV (ou VMOVEQ) est normale, non suffixée (BLX).

Un autre exemple qui est légèrement plus difficile, qui est aussi d'Angry Birds:

Listing 1.217: Angry Birds Classic

```
...
ITTTT EQ
MOVEQ           R0, R4
ADDEQ           SP, SP, #0x20
POPEQ.W         {R8,R10}
POPEQ           {R4-R7,PC}
BLX             ___stack_chk_fail ; not suffixed
...
```

Les quatre symboles « T » dans le mnémonique de l'instruction signifient que les quatre instructions suivantes seront exécutées si la condition est vraie.

C'est pourquoi IDA ajoute le suffixe -EQ à chacune d'entre elles.

Et si il y avait, par exemple, ITEEE EQ (*if-then-else-else-else*), alors les suffixes seraient mis comme suit:

```
-EQ
-NE
-NE
-NE
```

Un autre morceau de code d'Angry Birds:

Listing 1.218: Angry Birds Classic

```
...
CMP.W           R0, #0xFFFFFFFF
ITTE LE
SUBLE.W         R10, R0, #1
NEGLE           R0, R0
MOVGT           R10, R0
MOVS            R6, #0          ; not suffixed
CBZ             R0, loc_1E7E32  ; not suffixed
...
```

ITTE (*if-then-then-else*)

implique que les 1ère et 2ème instructions seront exécutées si la condition LE (*Less or Equal* moins ou égal) est vraie, et que la 3ème—si la condition inverse (GT—*Greater Than* plus grand que) est vraie.

En général, les compilateurs ne génèrent pas toutes les combinaisons possible.

Par exemple, dans le jeu Angry Birds mentionné ((*classic* version pour iOS) seules les les variantes suivantes de l'instruction IT sont utilisées: IT, ITE, ITT, ITTE, ITTT, ITTTT. Comment savoir cela? Dans IDA, il est possible de produire un listing dans un fichier, ce qui a été utilisé pour en créer un avec l'option d'afficher 4 octets pour chaque opcode. Ensuite, en connaissant la partie haute de l'opcode de 16-bit (0xBF pour IT), nous utilisons grep ainsi:

```
cat AngryBirdsClassic.lst | grep " BF" | grep "IT" > results.lst
```

À propos, si vous programmez en langage d'assemblage ARM pour le mode Thumb-2, et que vous ajoutez des suffixes conditionnels, l'assembleur ajoutera automatiquement l'instruction IT avec les flags là où ils sont nécessaires.

sans optimisation Xcode 4.6.3 (LLVM) (Mode ARM)

Listing 1.219: sans optimisation Xcode 4.6.3 (LLVM) (Mode ARM)

```
b               = -0x20
a               = -0x18
val_to_return   = -0x10
saved_R7        = -4

                STR             R7, [SP,#saved_R7]!
                MOV             R7, SP
                SUB             SP, SP, #0x1C
                BIC             SP, SP, #7
                VMOV            D16, R2, R3
                VMOV            D17, R0, R1
                VSTR            D17, [SP,#0x20+a]
                VSTR            D16, [SP,#0x20+b]
                VLDR            D16, [SP,#0x20+a]
                VLDR            D17, [SP,#0x20+b]
                VCMPE.F64       D16, D17
                VMRS            APSR_nzcv, FPSCR
                BLE             loc_2E08
                VLDR            D16, [SP,#0x20+a]
                VSTR            D16, [SP,#0x20+val_to_return]
                B               loc_2E10

loc_2E08
                VLDR            D16, [SP,#0x20+b]
                VSTR            D16, [SP,#0x20+val_to_return]

loc_2E10
                VLDR            D16, [SP,#0x20+val_to_return]
                VMOV            R0, R1, D16
                MOV             SP, R7
                LDR             R7, [SP+0x20+b],#4
                BX              LR
```

Presque la même chose que nous avons déjà vu, mais ici il y a beaucoup de code redondant car les variables a et b sont stockées sur la pile locale, tout comme la valeur de retour.

avec optimisation Keil 6/2013 (Mode Thumb)

Listing 1.220: avec optimisation Keil 6/2013 (Mode Thumb)

```
                PUSH    {R3-R7,LR}
                MOVS    R4, R2
                MOVS    R5, R3
                MOVS    R6, R0
```

```
                MOVS    R7, R1
                BL      __aeabi_cdrcmple
                BCS     loc_1C0
                MOVS    R0, R6
                MOVS    R1, R7
                POP     {R3-R7,PC}

loc_1C0
                MOVS    R0, R4
                MOVS    R1, R5
                POP     {R3-R7,PC}
```

Keil ne génère pas les instructions pour le FPU car il ne peut pas être sûr qu'elles sont supportées sur le CPU cible, et cela ne peut pas être fait directement en comparant les bits. Donc il appelle une fonction d'une bibliothèque externe pour effectuer la comparaison: `__aeabi_cdrcmple`.

N.B. Le résultat de la comparaison est laissé dans les flags par cette fonction, donc l'instruction BCS (*Carry set—Greater than or equal* plus grand ou égal) fonctionne sans code additionnel.

ARM64

GCC (Linaro) 4.9 avec optimisation

```
d_max :
; D0 - a, D1 - b
        fcmpe   d0, d1
        fcsel   d0, d0, d1, gt
; maintenant le résultat est dans D0
        ret
```

L'ARM64 ISA possède des instructions FPU qui mettent les flags CPU APSR au lieu de FPSCR, par commodité. Le FPU n'est plus un device séparé (au moins, logiquement). Ici, nous voyons FCMPE. Ceci compare les deux valeurs passées dans D0 et D1 (qui sont le premier et le second argument de la fonction) et met les flags APSR (N, Z, C, V).

FCSEL (*Floating Conditional Select* (sélection de flottant conditionnelle) copie la valeur de D0 ou D1 dans D0 suivant le résultat de la comparaison (GT—Greater Than), et de nouveau, il utilise les flags dans le registre APSR au lieu de FPSCR.

Ceci est bien plus pratique, comparé au jeu d'instructions des anciens CPUs.

Si la condition est vraie (GT), alors la valeur de D0 est copiée dans D0 (i.e., il ne se passe rien). Si la condition n'est pas vraie, la valeur de D1 est copiée dans D0.

GCC (Linaro) 4.9 sans optimisation

```
d_max :
; sauver les arguments en entrée dans la "Register Save Area"
;                             "zone de sauvegarde des registres"
        sub     sp, sp, #16
        str     d0, [sp,8]
        str     d1, [sp]
; recharger les valeurs
        ldr     x1, [sp,8]
        ldr     x0, [sp]
        fmov    d0, x1
        fmov    d1, x0
; D0 - a, D1 - b
        fcmpe   d0, d1
        ble     .L76
; a>b ; charger D0 (a) dans X0
        ldr     x0, [sp,8]
        b       .L74
.L76 :
; a<=b ; charger D1 (b) dans X0
```

```
        ldr     x0, [sp]
.L74 :
; résultat dans X0
        fmov    d0, x0
; résultat dans D0
        add     sp, sp, 16
        ret
```

GCC sans optimisation est plus verbeux.

Tout d'abord, la fonction sauve la valeur de ses arguments en entrée dans la pile locale (*Register Save Area*, espace de sauvegarde des registres). Ensuite, le code recharge ces valeurs dans les registres X0/X1 et finalement les copie dans D0/D1 afin de les comparer en utilisant FCMPE. Beaucoup de code redondant, mais c'est ainsi que fonctionne les compilateurs sans optimisation. FCMPE compare les valeurs et met les flags du registre APSR. À ce moment, le compilateur ne pense pas encore à l'instruction plus commode FCSEL, donc il procède en utilisant de vieilles méthodes: en utilisant l'instruction BLE (*Branch if Less than or Equal* branchement si inférieur ou égal). Dans le premier cas ($a > b$), la valeur de a est chargée dans X0. Dans les autres cas ($a <= b$), la valeur de b est chargée dans X0. Enfin, la valeur dans X0 est copiée dans D0, car la valeur de retour doit être dans ce registre.

Exercice

À titre d'exercice, vous pouvez essayer d'optimiser ce morceau de code manuellement en supprimant les instructions redondantes et sans en introduire de nouvelles (incluant FCSEL).

GCC (Linaro) 4.9 avec optimisation—float

Ré-écrivons cet exemple en utilisant des *float* à la place de *double*.

```
float f_max (float a, float b)
{
        if (a>b)
                return a;

        return b;
};
```

```
f_max :
; S0 - a, S1 - b
        fcmpe   s0, s1
        fcsel   s0, s0, s1, gt
; maintenant le résultat est dans S0
        ret
```

C'est le même code, mais des S-registres sont utilisés à la place de D-registres. C'est parce que les nombres de type *float* sont passés dans des S-registres de 32-bit (qui sont en fait la partie basse des D-registres 64-bit).

MIPS

Le coprocesseur du processeur MIPS possède un bit de condition qui peut être mis par le FPU et lu par le CPU.

Les premiers MIPSs avaient seulement un bit de condition (appelé FCC0), les derniers en ont 8 (appelés FCC7-FCC0).

Ce bit (ou ces bits) sont situés dans un registre appelé FCCR.

Listing 1.221: avec optimisation GCC 4.4.5 (IDA)

```
d_max :
; mettre le bit de condition du FPU si $f14<$f12 (b<a) :
                c.lt.d  $f14, $f12
                or      $at, $zero ; NOP
```

```
; sauter en locret_14 si le bit de condition est mis
                bc1t    locret_14
; cette instruction est toujours exécutée (mettre la valeur de retour à  "a") :
                mov.d   $f0, $f12  ; slot de délai de branchement
; cette instruction est exécutée seulement si la branche n'a pas été prise (i.e., si b>=a)
; mettre la valeur de retour à "b":
                mov.d   $f0, $f14

locret_14 :
                jr      $ra
                or      $at, $zero ; slot de délai de branchement, NOP
```

C.LT.D compare deux valeurs. LT est la condition « Less Than » (plus petit que). D implique des valeurs de type *double*. Suivant le résultat de la comparaison, le bit de condition FCC0 est mis à 1 ou à 0.

BC1T teste le bit FCC0 et saute si le bit est mis à 1. T signifie que le saut sera effectué si le bit est mis à 1 (« True »). Il y a aussi une instruction BC1F qui saute si le bit n'est pas mis (donc est à 0) (« False »).

Dépendant du saut, un des arguments de la fonction est placé dans $F0.

1.19.8 Quelques constantes

Il est facile de trouver la représentation de certaines constantes pour des nombres encodés au format IEEE 754 sur Wikipédia. Il est intéressant de savoir que 0,0 en IEEE 754 est représenté par 32 bits à zéro (pour la simple précision) ou 64 bits à zéro (pour la double). Donc pour mettre une variable flottante à 0,0 dans un registre ou en mémoire, on peut utiliser l'instruction MOV ou XOR reg, reg. Ceci est utilisable pour les structures où des variables de types variés sont présentes. Avec la fonction usuelle memset() il est possible de mettre toutes les variables entières à 0, toutes les variables booléennes à *false*, tous les pointeurs à NULL, et toutes les variables flottantes (de n'importe quelle précision) à 0,0.

1.19.9 Copie

On peut tout d'abord penser qu'il faut utiliser les instructions FLD/FST pour charger et stocker (et donc, copier) des valeurs IEEE 754. Néanmoins, la même chose peut-être effectuée plus facilement avec l'instruction usuelle MOV, qui, bien sûr, copie les valeurs au niveau binaire.

1.19.10 Pile, calculateurs et notation polonaise inverse

Maintenant nous comprenons pourquoi certains anciens calculateurs utilisent la notation Polonaise inverse [131].

Par exemple, pour additionner 12 et 34, on doit entrer 12, puis 34, et presser le signe « plus ».

C'est parce que les anciens calculateurs étaient juste des implémentations de machine à pile, et c'était bien plus simple que de manipuler des expressions complexes avec parenthèses.

1.19.11 80 bits?

Représentation interne des nombres dans le FPU — 80-bit. Nombre étrange, car il n'est pas de la forme 2^n. Il y a une hypothèse que c'est probablement dû à des raisons historiques—le standard IBM de carte perforée peut encoder 12 lignes de 80 bits. La résolution en mode texte de $80 \cdot 25$ était aussi très populaire dans le passé.

Il y a une autre explication sur Wikipédia: https://en.wikipedia.org/wiki/Extended_precision.

Si vous en savez plus, s'il vous plaît, envoyez-moi un email: dennis@yurichev.com.

1.19.12 x64

Sur la manière dont sont traités les nombres à virgules flottante en x86-64, lire ici: 1.31 on page 429.

[131] wikipedia.org/wiki/Reverse_Polish_notation

1.19.13 Exercices

- http://challenges.re/60
- http://challenges.re/61

1.20 Tableaux

Un tableau est simplement un ensemble de variables en mémoire qui sont situées les unes à côté des autres et qui ont le même type[132].

1.20.1 Exemple simple

```
#include <stdio.h>

int main()
{
    int a[20];
    int i;

    for (i=0; i<20; i++)
        a[i]=i*2;

    for (i=0; i<20; i++)
        printf ("a[%d]=%d\n", i, a[i]);

    return 0;
};
```

x86

MSVC

Compilons:

Listing 1.222: MSVC 2008

```
_TEXT    SEGMENT
_i$ = -84                       ; size = 4
_a$ = -80                       ; size = 80
_main      PROC
    push   ebp
    mov    ebp, esp
    sub    esp, 84         ; 00000054H
    mov    DWORD PTR _i$[ebp], 0
    jmp    SHORT $LN6@main
$LN5@main :
    mov    eax, DWORD PTR _i$[ebp]
    add    eax, 1
    mov    DWORD PTR _i$[ebp], eax
$LN6@main :
    cmp    DWORD PTR _i$[ebp], 20   ; 00000014H
    jge    SHORT $LN4@main
    mov    ecx, DWORD PTR _i$[ebp]
    shl    ecx, 1
    mov    edx, DWORD PTR _i$[ebp]
    mov    DWORD PTR _a$[ebp+edx*4], ecx
    jmp    SHORT $LN5@main
$LN4@main :
    mov    DWORD PTR _i$[ebp], 0
    jmp    SHORT $LN3@main
```

[132] AKA « container homogène »

```
$LN2@main :
    mov    eax, DWORD PTR _i$[ebp]
    add    eax, 1
    mov    DWORD PTR _i$[ebp], eax
$LN3@main :
    cmp    DWORD PTR _i$[ebp], 20    ; 00000014H
    jge    SHORT $LN1@main
    mov    ecx, DWORD PTR _i$[ebp]
    mov    edx, DWORD PTR _a$[ebp+ecx*4]
    push   edx
    mov    eax, DWORD PTR _i$[ebp]
    push   eax
    push   OFFSET $SG2463
    call   _printf
    add    esp, 12         ; 0000000cH
    jmp    SHORT $LN2@main
$LN1@main :
    xor    eax, eax
    mov    esp, ebp
    pop    ebp
    ret    0
_main      ENDP
```

Rien de très particulier, juste deux boucles: la première est celle de remplissage et la seconde celle d'affichage. L'instruction `shl ecx, 1` est utilisée pour la multiplication par 2 de la valeur dans ECX, voir à ce sujet ci-après 1.18.2 on page 218.

80 octets sont alloués sur la pile pour le tableau, 20 éléments de 4 octets.

Essayons cet exemple dans OllyDbg.

Nous voyons comment le tableau est rempli:

chaque élément est un mot de 32-bit de type *int* et sa valeur est l'index multiplié par 2:

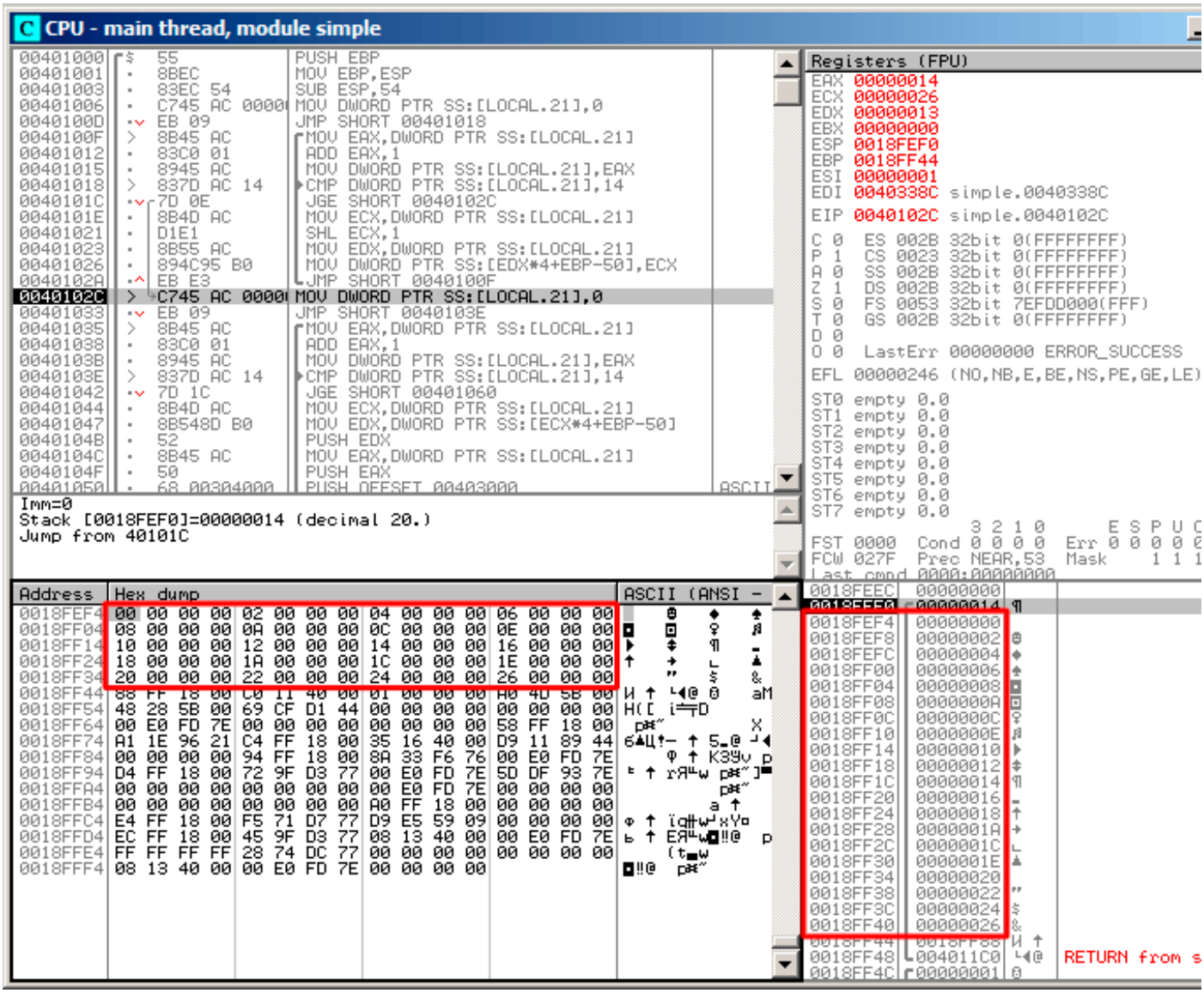


Fig. 1.87: OllyDbg : après remplissage du tableau

Puisque le tableau est situé sur la pile, nous pouvons voir ses 20 éléments ici.

GCC

Voici ce que GCC 4.4.1 génère:

Listing 1.223: GCC 4.4.1

```
                public main
main            proc near               ; DATA XREF : _start+17

var_70          = dword ptr -70h
var_6C          = dword ptr -6Ch
var_68          = dword ptr -68h
i_2             = dword ptr -54h
i               = dword ptr -4

                push    ebp
                mov     ebp, esp
```

```
                and     esp, 0FFFFFFF0h
                sub     esp, 70h
                mov     [esp+70h+i], 0          ; i=0
                jmp     short loc_804840A

loc_80483F7 :
                mov     eax, [esp+70h+i]
                mov     edx, [esp+70h+i]
                add     edx, edx                ; edx=i*2
                mov     [esp+eax*4+70h+i_2], edx
                add     [esp+70h+i], 1          ; i++

loc_804840A :
                cmp     [esp+70h+i], 13h
                jle     short loc_80483F7
                mov     [esp+70h+i], 0
                jmp     short loc_8048441

loc_804841B :
                mov     eax, [esp+70h+i]
                mov     edx, [esp+eax*4+70h+i_2]
                mov     eax, offset aADD ; "a[%d]=%d\n"
                mov     [esp+70h+var_68], edx
                mov     edx, [esp+70h+i]
                mov     [esp+70h+var_6C], edx
                mov     [esp+70h+var_70], eax
                call    _printf
                add     [esp+70h+i], 1

loc_8048441 :
                cmp     [esp+70h+i], 13h
                jle     short loc_804841B
                mov     eax, 0
                leave
                retn
main            endp
```

À propos, la variable a est de type *int** (un pointeur sur un *int*)—vous pouvez passer un pointeur sur un tableau à une autre fonction, mais c'est plus juste de dire qu'un pointeur sur le premier élément du tableau est passé (les adresses du reste des éléments sont calculées de manière évidente).

Si vous indexez ce pointeur en *a[idx]*, il suffit d'ajouter *idx* au pointeur et l'élément placé ici (sur lequel pointe le pointeur calculé) est renvoyé.

Un exemple intéressant: une chaîne de caractères comme *« string »* est un tableau de caractères et a un type *const char[]*.

Un index peut aussi être appliqué à ce pointeur.

Et c'est pourquoi il est possible d'écrire des choses comme « `string `»`[i]`—c'est une expression C/C++ correcte!

ARM

sans optimisation Keil 6/2013 (Mode ARM)

```
            EXPORT _main
_main
            STMFD   SP!, {R4,LR}
            SUB     SP, SP, #0x50       ; allouer de la place pour 20 variables int

; première boucle

            MOV     R4, #0              ; i
            B       loc_4A0
loc_494
            MOV     R0, R4,LSL#1        ; R0=R4*2
            STR     R0, [SP,R4,LSL#2]   ; stocker R0 dans SP+R4<<2 (pareil que SP+R4*4)
```

```
            ADD     R4, R4, #1          ; i=i+1

loc_4A0
            CMP     R4, #20             ; i<20?
            BLT     loc_494             ; oui, effectuer encore le corps de la boucle

; seconde boucle

            MOV     R4, #0              ; i
            B       loc_4C4
loc_4B0
            LDR     R2, [SP,R4,LSL#2]   ; (second argument de printf) R2=*(SP+R4<<4)
                                        ; (pareil que *(SP+R4*4))
            MOV     R1, R4              ; (premier argument de printf) R1=i
            ADR     R0, aADD            ; "a[%d]=%d\n"
            BL      __2printf
            ADD     R4, R4, #1          ; i=i+1

loc_4C4
            CMP     R4, #20             ; i<20?
            BLT     loc_4B0             ; oui, effectuer encore le corps de la boucle
            MOV     R0, #0              ; valeur à renvoyer
            ADD     SP, SP, #0x50       ; libérer le chunk, alloué pour 20 variables int
            LDMFD   SP!, {R4,PC}
```

Le type *int* nécessite 32 bits pour le stockage (ou 4 octets).

donc pour stocker 20 variables, *int* 80 (0x50) octets sont nécessaires.

C'est pourquoi l'instruction `SUB SP, SP, #0x50` dans le prologue de la fonction alloue exactement cet espace sur la pile.

Dans la première et la seconde boucle, la variable de boucle *i* se trouve dans le registre R4.

Le nombre qui doit être écrit dans le tableau est calculé comme $i * 2$, ce qui est effectivement équivalent à décaler d'un bit vers la gauche, ce que fait l'instruction `MOV R0, R4,LSL#1`.

`STR R0, [SP,R4,LSL#2]` écrit le contenu de R0 dans le tableau.

Voici comment le pointeur sur un élément du tableau est calculé: SP pointe sur le début du tableau, R4 est i.

Donc décaler i de 2 bits vers la gauche est effectivement équivalent à la multiplication par 4 (puisque chaque élément du tableau a une taille de 4 octets) et ensuite on l'ajoute à l'adresse du début du tableau.

La seconde boucle a l'instruction inverse `LDR R2, [SP,R4,LSL#2]`. Elle charge la valeur du tableau dont nous avons besoin, et le pointeur est calculé de même.

avec optimisation Keil 6/2013 (Mode Thumb)

```
_main
          PUSH    {R4,R5,LR}
; allouer de l'espace pour 20 variables int + une variable supplémentaire
          SUB     SP, SP, #0x54

; première boucle

          MOVS    R0, #0          ; i
          MOV     R5, SP          ; pointeur sur le premier élément du tableau

loc_1CE
          LSLS    R1, R0, #1      ; R1=i<<1 (pareil que i*2)
          LSLS    R2, R0, #2      ; R2=i<<2 (pareil que i*4)
          ADDS    R0, R0, #1      ; i=i+1
          CMP     R0, #20         ; i<20?
          STR     R1, [R5,R2]     ; stocker R1 dans *(R5+R2) (pareil que R5+i*4)
          BLT     loc_1CE         ; oui, i<20, effectuer encore le corps de la boucle

; seconde boucle
```

```
          MOVS    R4, #0          ; i=0
loc_1DC
          LSLS    R0, R4, #2      ; R0=i<<2 (pareil que i*4)
          LDR     R2, [R5,R0]     ; charger depuis *(R5+R0) (pareil que R5+i*4)
          MOVS    R1, R4
          ADR     R0, aADD        ; "a[%d]=%d\n"
          BL      __2printf
          ADDS    R4, R4, #1      ; i=i+1
          CMP     R4, #20         ; i<20?
          BLT     loc_1DC         ; oui, i<20, effectuer encore le corps de la boucle
          MOVS    R0, #0          ; valeur à renvoyer
; libérer le chunk, alloué pour 20 variables int + une variable supplémentaire
          ADD     SP, SP, #0x54
          POP     {R4,R5,PC}
```

Le code Thumb est très similaire.

Le mode Thumb a des instructions spéciales pour le décalage (comme LSLS), qui calculent la valeur à écrire dans le tableau et l'adresse de chaque élément dans le tableau.

Le compilateur alloue légèrement plus d'espace sur la pile locale, cependant, les 4 derniers octets ne sont pas utilisés.

GCC 4.9.1 sans optimisation (ARM64)

Listing 1.224: GCC 4.9.1 sans optimisation (ARM64)

```
.LC0 :
        .string "a[%d]=%d\n"
main :
; sauver FR et LR sur la stack frame :
        stp     x29, x30, [sp, -112]!
; définir l'environnement de pile (FP=SP)
        add     x29, sp, 0
; initialiser la variable compteur à 0 (WZR est le registre qui contient toujours zéro) :
        str     wzr, [x29,108]
; sauter au code de test de condition de la boucle :
        b       .L2
.L3 :
; charger la valeur de la variable "i":
        ldr     w0, [x29,108]
; la multiplier par 2:
        lsl     w2, w0, 1
; trouver la place du tableau dans la pile locale :
        add     x0, x29, 24
; charger l'entier 32-bit depuis la pile locale et l'étendre en un 64-bit signé :
        ldrsw   x1, [x29,108]
; calculer l'adresse de l'élément (X0+X1<<2=adresse du tableau+i*4) et y stocker W2 (i*2) :
        str     w2, [x0,x1,lsl 2]
; incrémenter le compteur (i) :
        ldr     w0, [x29,108]
        add     w0, w0, 1
        str     w0, [x29,108]
.L2 :
; tester si la boucle est finie :
        ldr     w0, [x29,108]
        cmp     w0, 19
; sauter en L3 (début du corps de la boucle) si non :
        ble     .L3
; La seconde partie de la fonction commence ici.
; mettre la valeur initiale da la variable compteur à 0.
; à propos, le même espace est utilisé dans la pile locale,
; car la même variable locale (i) est utilisée comme compteur.
        str     wzr, [x29,108]
        b       .L4
.L5 :
; calculer l'adresse dans le tableau :
        add     x0, x29, 24
; charger la valeur de "i":
```

```
        ldrsw   x1, [x29,108]
; charger la valeur du tableau à l'adresse (X0+X1<<2 = adresse du tableau + i*4)
        ldr     w2, [x0,x1,lsl 2]
; charger l'adresse de la chaîne "a[%d]=%d\n":
        adrp    x0, .LC0
        add     x0, x0,  :lo12 :.LC0
; charger la variable "i" dans W1 et la passer à printf() comme second argument :
        ldr     w1, [x29,108]
; W2 contient toujours la valeur de l'élément du tableau qui vient d'être chargée.
; appeler printf() :
        bl      printf
; incrémenter la variable "i":
        ldr     w0, [x29,108]
        add     w0, w0, 1
        str     w0, [x29,108]
.L4 :
; est-ce fini?
        ldr     w0, [x29,108]
        cmp     w0, 19
; sauter au début du corps de la boucle si non :
        ble     .L5
; renvoyer 0
        mov     w0, 0
; restaurer FP et LR :
        ldp     x29, x30, [sp], 112
        ret
```

MIPS

La fonction utilise beaucoup de S- registres qui doivent être préservés, c'est pourquoi leurs valeurs sont sauvegardées dans la prologue de la fonction et restaurées dans l'épilogue.

Listing 1.225: GCC 4.4.5 avec optimisation (IDA)

```
main :

var_70          = -0x70
var_68          = -0x68
var_14          = -0x14
var_10          = -0x10
var_C           = -0xC
var_8           = -8
var_4           = -4
; prologue de la fonction :
                lui     $gp, (__gnu_local_gp >> 16)
                addiu   $sp, -0x80
                la      $gp, (__gnu_local_gp & 0xFFFF)
                sw      $ra, 0x80+var_4($sp)
                sw      $s3, 0x80+var_8($sp)
                sw      $s2, 0x80+var_C($sp)
                sw      $s1, 0x80+var_10($sp)
                sw      $s0, 0x80+var_14($sp)
                sw      $gp, 0x80+var_70($sp)
                addiu   $s1, $sp, 0x80+var_68
                move    $v1, $s1
                move    $v0, $zero
; cette valeur va être utilisée comme fin de boucle.
; elle a été pré-calculée par le compilateur GCC à l'étape de la compilation :
                li      $a0, 0x28  # '('

loc_34 :                                # CODE XREF : main+3C
; stocker la valeur en mémoire :
                sw      $v0, 0($v1)
; incrémenter la valeur à sauver de 2 à chaque itération :
                addiu   $v0, 2
; fin de boucle atteinte?
                bne     $v0, $a0, loc_34
; ajouter 4 à l'adresse dans tous les cas
```

```
                addiu   $v1, 4
; la boucle de remplissage du tableau est finie
; la seconde boucle commence
                la      $s3, $LC0        # "a[%d]=%d\n"
; la variable "i" va être dans $s0 :
                move    $s0, $zero
                li      $s2, 0x14

loc_54 :                                 # CODE XREF : main+70
; appeler printf() :
                lw      $t9, (printf & 0xFFFF)($gp)
                lw      $a2, 0($s1)
                move    $a1, $s0
                move    $a0, $s3
                jalr    $t9
; incrémenter "i":
                addiu   $s0, 1
                lw      $gp, 0x80+var_70($sp)
; sauter au corps de la boucle si la fin n'est pas atteinte :
                bne     $s0, $s2, loc_54
; déplacer le pointeur mémoire au prochain mot de 32-bit :
                addiu   $s1, 4
; épilogue de la fonction
                lw      $ra, 0x80+var_4($sp)
                move    $v0, $zero
                lw      $s3, 0x80+var_8($sp)
                lw      $s2, 0x80+var_C($sp)
                lw      $s1, 0x80+var_10($sp)
                lw      $s0, 0x80+var_14($sp)
                jr      $ra
                addiu   $sp, 0x80

$LC0 :           .ascii "a[%d]=%d\n"<0>   # DATA XREF : main+44
```

Quelque chose d'intéressant: il y a deux boucles et la première n'a pas besoin de i, elle a seulement besoin de $i * 2$ (augmenté de 2 à chaque itération) et aussi de l'adresse en mémoire (augmentée de 4 à chaque itération).

Donc ici nous voyons deux variables, une (dans $V0) augmentée de 2 à chaque fois, et une autre (dans $V1) — de 4.

La seconde boucle est celle où `printf()` est appelée et affiche la valeur de i à l'utilisateur, donc il y a une variable qui est incrémentée de 1 à chaque fois (dans $S0) et aussi l'adresse en mémoire (dans $S1) incrémentée de 4 à chaque fois.

Cela nous rappelle l'optimisation de boucle que nous avons examiné avant: 3.7 on page 494.

Leur but est de se passer des multiplications.

1.20.2 Débordement de tampon

Lire en dehors des bornes du tableau

Donc, indexer un tableau est juste *array[index]*. Si vous étudiez le code généré avec soin, vous remarquerez sans doute l'absence de test sur les bornes de l'index, qui devrait vérifier *si il est inférieur à 20*. Que ce passe-t-il si l'index est supérieur à 20? C'est une des caractéristiques de C/C++ qui est souvent critiquée.

Voici un code qui compile et fonctionne:

```
#include <stdio.h>

int main()
{
        int a[20];
        int i;

        for (i=0; i<20; i++)
                a[i]=i*2;
```

```
        printf ("a[20]=%d\n", a[20]);

        return 0;
};
```

Résultat de la compilation (MSVC 2008):

Listing 1.226: MSVC 2008 sans optimisation

```
$SG2474 DB    'a[20]=%d', 0aH, 00H

_i$ = -84 ; size = 4
_a$ = -80 ; size = 80
_main    PROC
    push   ebp
    mov    ebp, esp
    sub    esp, 84
    mov    DWORD PTR _i$[ebp], 0
    jmp    SHORT $LN3@main
$LN2@main :
    mov    eax, DWORD PTR _i$[ebp]
    add    eax, 1
    mov    DWORD PTR _i$[ebp], eax
$LN3@main :
    cmp    DWORD PTR _i$[ebp], 20
    jge    SHORT $LN1@main
    mov    ecx, DWORD PTR _i$[ebp]
    shl    ecx, 1
    mov    edx, DWORD PTR _i$[ebp]
    mov    DWORD PTR _a$[ebp+edx*4], ecx
    jmp    SHORT $LN2@main
$LN1@main :
    mov    eax, DWORD PTR _a$[ebp+80]
    push   eax
    push   OFFSET $SG2474 ; 'a[20]=%d'
    call   DWORD PTR __imp__printf
    add    esp, 8
    xor    eax, eax
    mov    esp, ebp
    pop    ebp
    ret    0
_main    ENDP
_TEXT    ENDS
END
```

Le code produit ce résultat:

Listing 1.227: OllyDbg : sortie sur la console

```
a[20]=1638280
```

C'est juste *quelque chose* qui se trouvait sur la pile à côté du tableau, 80 octets après le début de son premier élément.

Essayons de trouver d'où vient cette valeur, en utilisant OllyDbg.

Chargeons et trouvons la valeur située juste après le dernier élément du tableau:

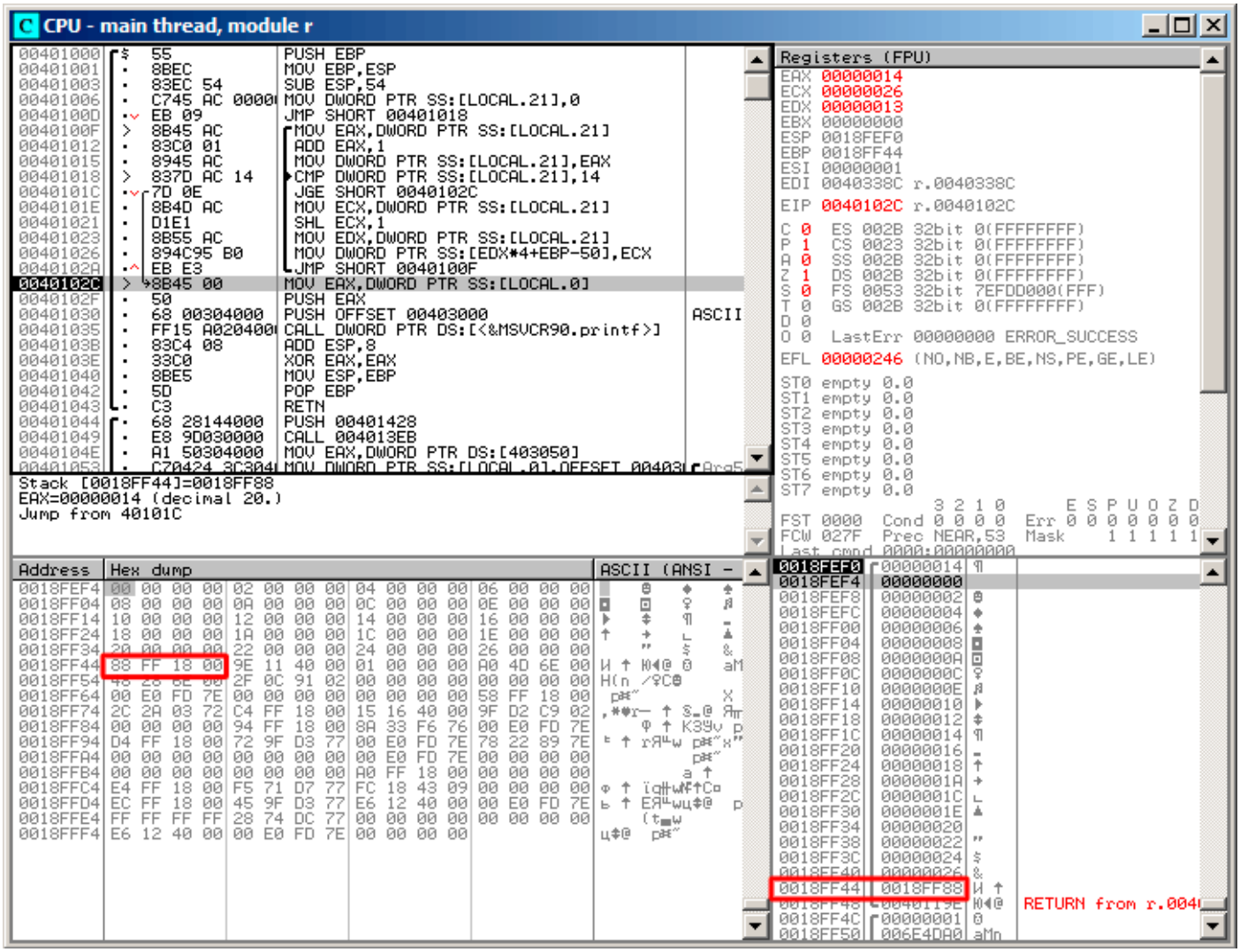


Fig. 1.88: OllyDbg : lecture du 20ème élément et exécution de `printf()`

Qu'est-ce que c'est? D'après le schéma de la pile, c'est la valeur sauvegardée du registre EBP.

Exécutons encore et voyons comment il est restauré:

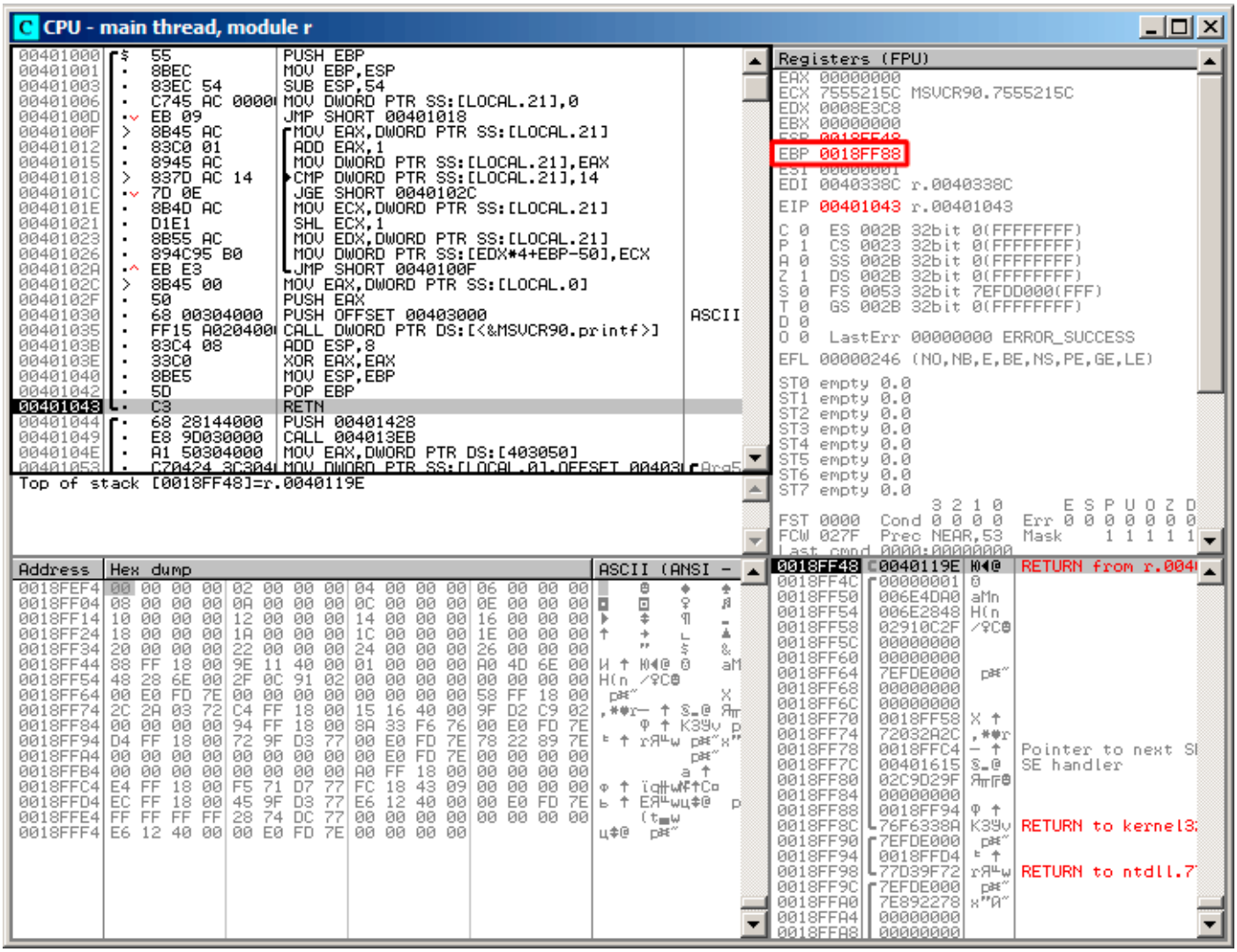


Fig. 1.89: OllyDbg : restaurer la valeur de EBP

En effet, comment est-ce ça pourrait être différent? Le compilateur pourrait générer du code supplémentaire pour vérifier que la valeur de l'index est toujours entre les bornes du tableau (comme dans les langages de programmation de plus haut-niveau[133]) mais cela rendrait le code plus lent.

Écrire hors des bornes du tableau

Ok, nous avons lu quelques valeurs de la pile *illégalement*, mais que se passe-t-il si nous essayons d'écrire quelque chose?

Voici ce que nous avons:

```
#include <stdio.h>

int main()
{
        int a[20];
        int i;

        for (i=0; i<30; i++)
                a[i]=i;

        return 0;
};
```

[133]Java, Python, etc.

MSVC

Et ce que nous obtenons:

Listing 1.228: MSVC 2008 sans optimisation

```
_TEXT    SEGMENT
_i$ = -84 ; taille = 4
_a$ = -80 ; taille = 80
_main   PROC
 push   ebp
 mov    ebp, esp
 sub    esp, 84
 mov    DWORD PTR _i$[ebp], 0
 jmp    SHORT $LN3@main
$LN2@main :
 mov    eax, DWORD PTR _i$[ebp]
 add    eax, 1
 mov    DWORD PTR _i$[ebp], eax
$LN3@main :
 cmp    DWORD PTR _i$[ebp], 30 ; 0000001eH
 jge    SHORT $LN1@main
 mov    ecx, DWORD PTR _i$[ebp]
 mov    edx, DWORD PTR _i$[ebp]      ; cette instruction est évidemment redondante
 mov    DWORD PTR _a$[ebp+ecx*4], edx ; ECX pourrait être utilisé en second opérande ici
 jmp    SHORT $LN2@main
$LN1@main :
 xor    eax, eax
 mov    esp, ebp
 pop    ebp
 ret    0
_main   ENDP
```

Le programme compilé plante après le lancement. Pas de miracle. Voyons exactement où il plante.

Chargeons le dans OllyDbg, et traçons le jusqu'à ce que les 30 éléments du tableau soient écrits:

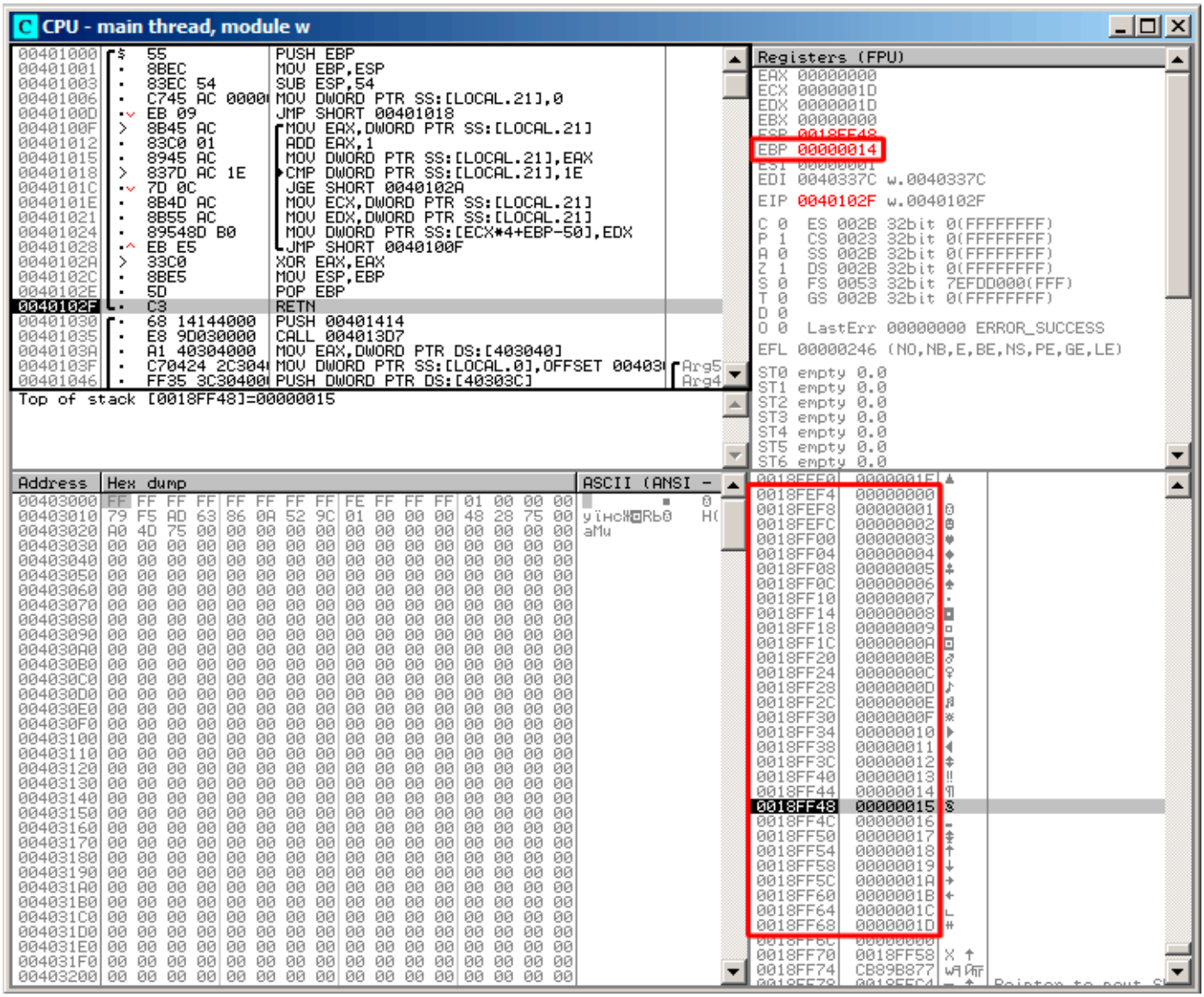


Fig. 1.90: OllyDbg : après avoir restauré la valeur de EBP

Exécutons pas à pas jusqu'à la fin de la fonction:

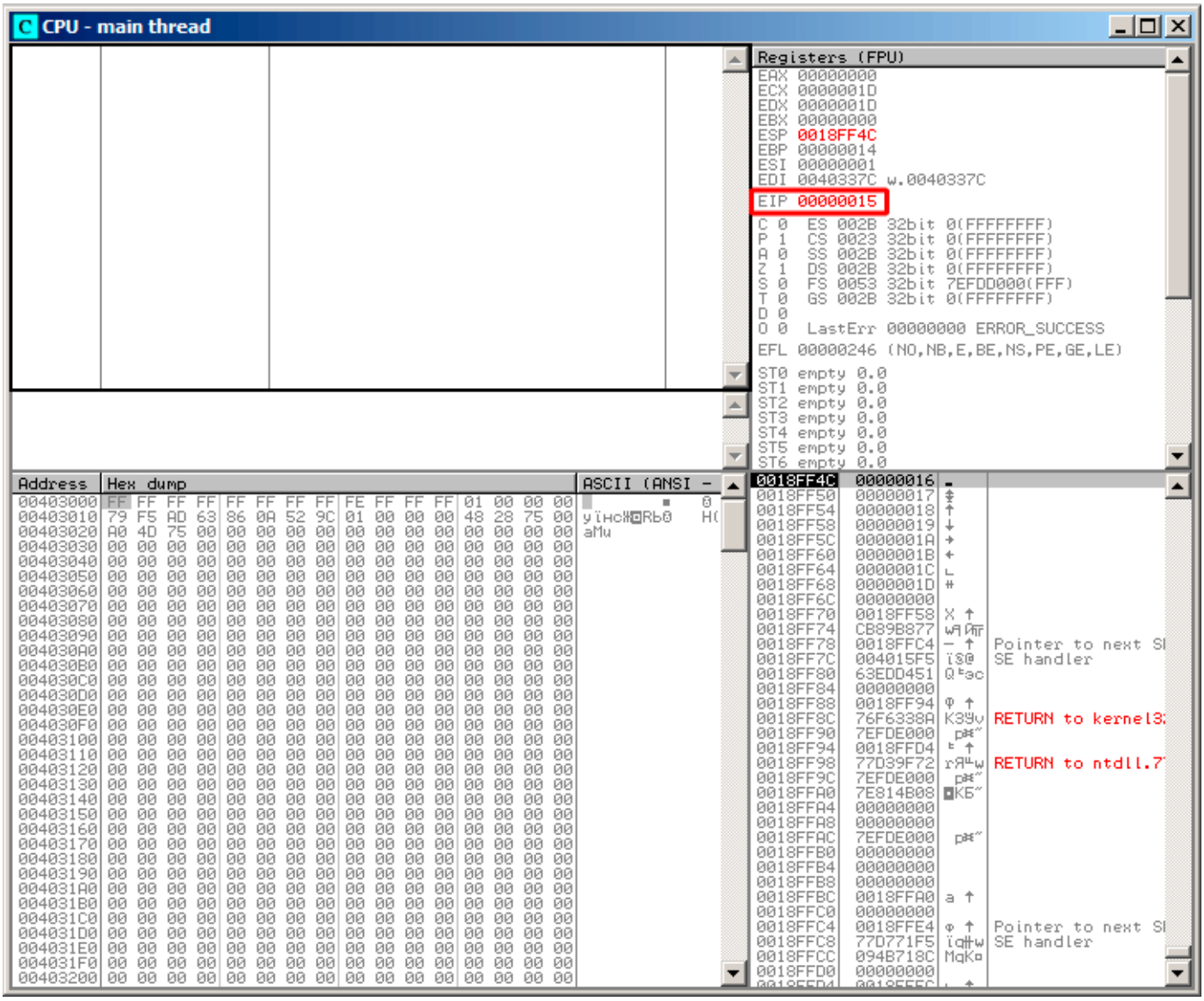


Fig. 1.91: OllyDbg : EIP a été restauré, mais OllyDbg ne peut pas désassembler en 0x15

Maintenant, gardez vos yeux sur les registres.

EIP contient maintenant 0x15. Ce n'est pas une adresse légale pour du code—au moins pour du code win32! Nous sommes arrivés ici contre notre volonté. Il est aussi intéressant de voir que le registre EBP contient 0x14, ECX et EDX contiennent 0x1D.

Étudions un peu plus la structure de la pile.

Après que le contrôle du flux a été passé à `main()`, la valeur du registre EBP a été sauvée sur la pile. Puis, 84 octets ont été alloués pour le tableau et la variable i. C'est `(20+1)*sizeof(int)`. ESP pointe maintenant sur la variable `_i` dans la pile locale et après l'exécution du PUSH `quelquechose` suivant, *quelquechose* apparaît à côté de `_i`.

C'est la structure de la pile pendant que le contrôle est dans `main()` :

ESP	4 octets alloués pour la variable i
ESP+4	80 octets alloués pour le tableau `a[20]`
ESP+84	valeur sauvegardée de EBP
ESP+88	adresse de retour

L'expression `a[19]=quelquechose` écrit le dernier *int* dans des bornes du tableau (dans les limites jusqu'ici!)

L'expression `a[20]=quelquechose` écrit *quelquechose* à l'endroit où la valeur sauvegardée de EBP se trouve.

S'il vous plaît, regardez l'état du registre lors du plantage. Dans notre cas, 20 a été écrit dans le 20ème élément. À la fin de la fonction, l'épilogue restaure la valeur d'origine de EBP. (20 en décimal est 0x14 en hexadécimal). Ensuite RET est exécuté, qui est équivalent à l'instruction POP EIP.

L'instruction RET prend la valeur de retour sur la pile (c'est l'adresse dans CRT), qui a appelé main()), et 21 est stocké ici (0x15 en hexadécimal). Le CPU trape à l'adresse 0x15, mais il n'y a pas de code exécutable ici, donc une exception est levée.

Bienvenu! Ça s'appelle un *buffer overflow (débordement de tampon)*[134].

Remplacez la tableau de *int* avec une chaîne (*char* array), créez délibérément une longue chaîne et passez-là au programme, à la fonction, qui ne teste pas la longueur de la chaîne et la copie dans un petit buffer et vous serez capable de faire pointer le programme à une adresse où il devra sauter. C'est pas aussi simple dans la réalité, mais c'est comme cela que ça a apparu. L'article classique à propos de ça: [Aleph One, *Smashing The Stack For Fun And Profit*, (1996)][135].

GCC

Essayons le même code avec GCC 4.4.1. Nous obtenons:

```
                public main
main            proc near

a               = dword ptr -54h
i               = dword ptr -4

                push    ebp
                mov     ebp, esp
                sub     esp, 60h ; 96
                mov     [ebp+i], 0
                jmp     short loc_80483D1
loc_80483C3 :
                mov     eax, [ebp+i]
                mov     edx, [ebp+i]
                mov     [ebp+eax*4+a], edx
                add     [ebp+i], 1
loc_80483D1 :
                cmp     [ebp+i], 1Dh
                jle     short loc_80483C3
                mov     eax, 0
                leave
                retn
main            endp
```

Lancer ce programme sous Linux donnera: Segmentation fault.

Si nous le lançons dans le débogueur GDB, nous obtenons ceci:

```
(gdb) r
Starting program : /home/dennis/RE/1

Program received signal SIGSEGV, Segmentation fault.
0x00000016 in ?? ()
(gdb) info registers
eax            0x0      0
ecx            0xd2f96388       -755407992
edx            0x1d     29
ebx            0x26eff4 2551796
esp            0xbffff4b0       0xbffff4b0
ebp            0x15     0x15
esi            0x0      0
edi            0x0      0
eip            0x16     0x16
eflags         0x10202  [ IF RF ]
cs             0x73     115
ss             0x7b     123
```

[134] Wikipédia
[135] Aussi disponible en http://go.yurichev.com/17266

```
ds             0x7b     123
es             0x7b     123
fs             0x0      0
gs             0x33     51
(gdb)
```

Les valeurs des registres sont légèrement différentes de l'exemple win32, puisque la structure de la pile est également légèrement différente.

1.20.3 Méthodes de protection contre les débordements de tampon

Il existe quelques méthodes pour protéger contre ce fléau, indépendamment de la négligence des programmeurs C/C++. MSVC possède des options comme[136] :

```
/RTCs Stack Frame runtime checking
/GZ Enable stack checks (/RTCs)
```

Une des méthodes est d'écrire une valeur aléatoire entre les variables locales sur la pile dans le prologue de la fonction et de la vérifier dans l'épilogue, avant de sortir de la fonction. Si la valeur n'est pas la même, ne pas exécuter la dernière instruction RET, mais stopper (ou bloquer). Le processus va s'arrêter, mais c'est mieux qu'une attaque distante sur votre ordinateur.

Cette valeur aléatoire est parfois appelé un « canari », c'est lié au canari[137] que les mineurs utilisaient dans le passé afin de détecter rapidement les gaz toxiques.

Les canaris sont très sensibles aux gaz, ils deviennent très agités en cas de danger, et même meurent.

Si nous compilons notre exemple de tableau très simple (1.20.1 on page 269) dans MSVC avec les options RTC1 et RTCs, nous voyons un appel à `@_RTC_CheckStackVars@8` une fonction à la fin de la fonction qui vérifie si le « canari » est correct.

Voyons comment GCC gère ceci. Prenons un exemple `alloca()` (1.7.2 on page 35):

```
#ifdef __GNUC__
#include <alloca.h> // GCC
#else
#include <malloc.h> // MSVC
#endif
#include <stdio.h>

void f()
{
    char *buf=(char*)alloca (600);
#ifdef __GNUC__
    snprintf (buf, 600, "hi! %d, %d, %d\n", 1, 2, 3); // GCC
#else
    _snprintf (buf, 600, "hi! %d, %d, %d\n", 1, 2, 3); // MSVC
#endif

    puts (buf);
};
```

Par défaut, sans option supplémentaire, GCC 4.7.3 insère un test de « canari » dans le code:

Listing 1.229: GCC 4.7.3

```
.LC0 :
        .string "hi! %d, %d, %d\n"
f :
        push    ebp
        mov     ebp, esp
        push    ebx
        sub     esp, 676
        lea     ebx, [esp+39]
        and     ebx, -16
        mov     DWORD PTR [esp+20], 3
```

[136] méthode de protection contre les débordements de tampons côté compilateur:wikipedia.org/wiki/Buffer_overflow_protection
[137] wikipedia.org/wiki/Domestic_canary#Miner.27s_canary

```
        mov     DWORD PTR [esp+16], 2
        mov     DWORD PTR [esp+12], 1
        mov     DWORD PTR [esp+8], OFFSET FLAT :.LC0  ; "hi! %d, %d, %d\n"
        mov     DWORD PTR [esp+4], 600
        mov     DWORD PTR [esp], ebx
        mov     eax, DWORD PTR gs :20     ; canari
        mov     DWORD PTR [ebp-12], eax
        xor     eax, eax
        call    _snprintf
        mov     DWORD PTR [esp], ebx
        call    puts
        mov     eax, DWORD PTR [ebp-12]
        xor     eax, DWORD PTR gs :20     ; teste le canari
        jne     .L5
        mov     ebx, DWORD PTR [ebp-4]
        leave
        ret
.L5 :
        call    __stack_chk_fail
```

La valeur aléatoire se trouve en `gs:20`. Elle est écrite sur la pile et à la fin de la fonction, la valeur sur la pile est comparée avec le « canari » correct dans `gs:20`. Si les valeurs ne sont pas égales, la fonction `__stack_chk_fail` est appelée et nous voyons dans la console quelque chose comme ça (Ubuntu 13.04 x86):

```
*** buffer overflow detected *** : ./2_1 terminated
======= Backtrace : =========
/lib/i386-linux-gnu/libc.so.6(__fortify_fail+0x63)[0xb7699bc3]
/lib/i386-linux-gnu/libc.so.6(+0x10593a)[0xb769893a]
/lib/i386-linux-gnu/libc.so.6(+0x105008)[0xb7698008]
/lib/i386-linux-gnu/libc.so.6(_IO_default_xsputn+0x8c)[0xb7606e5c]
/lib/i386-linux-gnu/libc.so.6(_IO_vfprintf+0x165)[0xb75d7a45]
/lib/i386-linux-gnu/libc.so.6(__vsprintf_chk+0xc9)[0xb76980d9]
/lib/i386-linux-gnu/libc.so.6(__sprintf_chk+0x2f)[0xb7697fef]
./2_1[0x8048404]
/lib/i386-linux-gnu/libc.so.6(__libc_start_main+0xf5)[0xb75ac935]
======= Memory map : ========
08048000-08049000 r-xp 00000000 08:01 2097586    /home/dennis/2_1
08049000-0804a000 r--p 00000000 08:01 2097586    /home/dennis/2_1
0804a000-0804b000 rw-p 00001000 08:01 2097586    /home/dennis/2_1
094d1000-094f2000 rw-p 00000000 00:00 0          [heap]
b7560000-b757b000 r-xp 00000000 08:01 1048602    /lib/i386-linux-gnu/libgcc_s.so.1
b757b000-b757c000 r--p 0001a000 08:01 1048602    /lib/i386-linux-gnu/libgcc_s.so.1
b757c000-b757d000 rw-p 0001b000 08:01 1048602    /lib/i386-linux-gnu/libgcc_s.so.1
b7592000-b7593000 rw-p 00000000 00:00 0
b7593000-b7740000 r-xp 00000000 08:01 1050781    /lib/i386-linux-gnu/libc-2.17.so
b7740000-b7742000 r--p 001ad000 08:01 1050781    /lib/i386-linux-gnu/libc-2.17.so
b7742000-b7743000 rw-p 001af000 08:01 1050781    /lib/i386-linux-gnu/libc-2.17.so
b7743000-b7746000 rw-p 00000000 00:00 0
b775a000-b775d000 rw-p 00000000 00:00 0
b775d000-b775e000 r-xp 00000000 00:00 0          [vdso]
b775e000-b777e000 r-xp 00000000 08:01 1050794    /lib/i386-linux-gnu/ld-2.17.so
b777e000-b777f000 r--p 0001f000 08:01 1050794    /lib/i386-linux-gnu/ld-2.17.so
b777f000-b7780000 rw-p 00020000 08:01 1050794    /lib/i386-linux-gnu/ld-2.17.so
bff35000-bff56000 rw-p 00000000 00:00 0          [stack]
Aborted (core dumped)
```

gs est ainsi appelé registre de segment. Ces registres étaient beaucoup utilisés du temps de MS-DOS et des extensions de DOS. Aujourd'hui, sa fonction est différente.

Dit brièvement, le registre gs dans Linux pointe toujours sur le TLS[138] (4.2 on page 552)—des informations spécifiques au thread sont stockées là. À propos, en win32 le registre `fs` joue le même rôle, pointant sur TIB[139] [140].

Il y a plus d'information dans le code source du noyau Linux (au moins dans la version 3.11), dans *arch/x86/include/asm/stackprotector.h* cette variable est décrite dans les commentaires.

[138] Thread Local Storage
[139] Thread Information Block
[140] wikipedia.org/wiki/Win32_Thread_Information_Block

avec optimisation Xcode 4.6.3 (LLVM) (Mode Thumb-2)

Reprenons notre exemple de simple tableau (1.20.1 on page 269),

à nouveau, nous pouvons voir comment LLVM teste si le « canari » est correct:

```
_main

var_64          = -0x64
var_60          = -0x60
var_5C          = -0x5C
var_58          = -0x58
var_54          = -0x54
var_50          = -0x50
var_4C          = -0x4C
var_48          = -0x48
var_44          = -0x44
var_40          = -0x40
var_3C          = -0x3C
var_38          = -0x38
var_34          = -0x34
var_30          = -0x30
var_2C          = -0x2C
var_28          = -0x28
var_24          = -0x24
var_20          = -0x20
var_1C          = -0x1C
var_18          = -0x18
canary          = -0x14
var_10          = -0x10

    PUSH     {R4-R7,LR}
    ADD      R7, SP, #0xC
    STR.W    R8, [SP,#0xC+var_10]!
    SUB      SP, SP, #0x54
    MOVW     R0, #aObjc_methtype ; "objc_methtype"
    MOVS     R2, #0
    MOVT.W   R0, #0
    MOVS     R5, #0
    ADD      R0, PC
    LDR.W    R8, [R0]
    LDR.W    R0, [R8]
    STR      R0, [SP,#0x64+canari]
    MOVS     R0, #2
    STR      R2, [SP,#0x64+var_64]
    STR      R0, [SP,#0x64+var_60]
    MOVS     R0, #4
    STR      R0, [SP,#0x64+var_5C]
    MOVS     R0, #6
    STR      R0, [SP,#0x64+var_58]
    MOVS     R0, #8
    STR      R0, [SP,#0x64+var_54]
    MOVS     R0, #0xA
    STR      R0, [SP,#0x64+var_50]
    MOVS     R0, #0xC
    STR      R0, [SP,#0x64+var_4C]
    MOVS     R0, #0xE
    STR      R0, [SP,#0x64+var_48]
    MOVS     R0, #0x10
    STR      R0, [SP,#0x64+var_44]
    MOVS     R0, #0x12
    STR      R0, [SP,#0x64+var_40]
    MOVS     R0, #0x14
    STR      R0, [SP,#0x64+var_3C]
    MOVS     R0, #0x16
    STR      R0, [SP,#0x64+var_38]
    MOVS     R0, #0x18
    STR      R0, [SP,#0x64+var_34]
    MOVS     R0, #0x1A
    STR      R0, [SP,#0x64+var_30]
```

```
    MOVS     R0, #0x1C
    STR      R0, [SP,#0x64+var_2C]
    MOVS     R0, #0x1E
    STR      R0, [SP,#0x64+var_28]
    MOVS     R0, #0x20
    STR      R0, [SP,#0x64+var_24]
    MOVS     R0, #0x22
    STR      R0, [SP,#0x64+var_20]
    MOVS     R0, #0x24
    STR      R0, [SP,#0x64+var_1C]
    MOVS     R0, #0x26
    STR      R0, [SP,#0x64+var_18]
    MOV      R4, 0xFDA ; "a[%d]=%d\n"
    MOV      R0, SP
    ADDS     R6, R0, #4
    ADD      R4, PC
    B        loc_2F1C

; début de la seconde boucle

loc_2F14
    ADDS     R0, R5, #1
    LDR.W    R2, [R6,R5,LSL#2]
    MOV      R5, R0

loc_2F1C
    MOV      R0, R4
    MOV      R1, R5
    BLX      _printf
    CMP      R5, #0x13
    BNE      loc_2F14
    LDR.W    R0, [R8]
    LDR      R1, [SP,#0x64+canari]
    CMP      R0, R1
    ITTTT EQ             ; est-ce que le canari est toujours correct?
    MOVEQ    R0, #0
    ADDEQ    SP, SP, #0x54
    LDREQ.W  R8, [SP+0x64+var_64],#4
    POPEQ    {R4-R7,PC}
    BLX      ___stack_chk_fail
```

Tout d'abord, on voit que LLVM a « déroulé » la boucle et que toutes les valeurs sont écrites une par une, pré-calculée, car LLVM a conclu que c'est plus rapide. À propos, des instructions en mode ARM peuvent aider à rendre cela encore plus rapide, et les trouver peut être un exercice pour vous.

À la fin de la fonction, nous voyons la comparaison des « canaris »—celui sur la pile locale et le correct.

S'ils sont égaux, un bloc de 4 instructions est exécuté par `ITTTT EQ`, qui contient l'écriture de 0 dans `R0`, l'épilogue de la fonction et la sortie. Si les « canaris » ne sont pas égaux, le bloc est passé, et la fonction saute en `___stack_chk_fail`, qui, peut-être, stoppe l'exécution.

1.20.4 Encore un mot sur les tableaux

Maintenant nous comprenons pourquoi il est impossible d'écrire quelque chose comme ceci en code C/C++ :

```
void f(int size)
{
    int a[size];
...
};
```

C'est simplement parce que le compilateur doit connaître la taille exacte du tableau pour lui allouer de l'espace sur la pile locale lors de l'étape de compilation.

Si vous avez besoin d'un tableau de taille arbitraire, il faut l'allouer en utilisant `malloc()`, puis en accédant aux blocs de mémoire allouée comme un tableau de variables du type dont vous avez besoin.

Ou utiliser la caractéristique du standart C99 [*ISO/IEC 9899:TC3 (C C99 standard)*, (2007)6.7.5/2], et qui fonctionne comme *alloca()* (1.7.2 on page 35) en interne.

Il est aussi possible d'utiliser des bibliothèques de ramasse-miettes pour C.

Et il y a aussi des bibliothèques supportant les pointeurs intelligents pour C++.

1.20.5 Tableau de pointeurs sur des chaînes

Voici un exemple de tableau de pointeurs.[141]

Listing 1.230: Prendre le nom du mois

```
#include <stdio.h>

const char* month1[]=
{
        "janvier", "fevrier", "mars", "avril",
        "mai", "juin", "juillet", "aout",
        "septembre", "octobre", "novembre", "decembre"
};

// dans l'intervalle 0..11
const char* get_month1 (int month)
{
        return month1[month];
};
```

x64

Listing 1.231: MSVC 2013 avec optimisation x64

```
_DATA   SEGMENT
month1  DQ      FLAT :$SG3122
        DQ      FLAT :$SG3123
        DQ      FLAT :$SG3124
        DQ      FLAT :$SG3125
        DQ      FLAT :$SG3126
        DQ      FLAT :$SG3127
        DQ      FLAT :$SG3128
        DQ      FLAT :$SG3129
        DQ      FLAT :$SG3130
        DQ      FLAT :$SG3131
        DQ      FLAT :$SG3132
        DQ      FLAT :$SG3133
$SG3122 DB      'January', 00H
$SG3123 DB      'February', 00H
$SG3124 DB      'March', 00H
$SG3125 DB      'April', 00H
$SG3126 DB      'May', 00H
$SG3127 DB      'June', 00H
$SG3128 DB      'July', 00H
$SG3129 DB      'August', 00H
$SG3130 DB      'September', 00H
$SG3156 DB      '%s', 0aH, 00H
$SG3131 DB      'October', 00H
$SG3132 DB      'November', 00H
$SG3133 DB      'December', 00H
_DATA   ENDS

month$ = 8
get_month1 PROC
        movsxd  rax, ecx
        lea     rcx, OFFSET FLAT :month1
```

[141] NDT: attention à l'encodage des fichiers, en ASCII ou en ISO-8859, un caractère occupe un octet, alors qu'en UTF-8, notamment, il peut en occuper plusieurs. Par exemple, 'û' est codé $fb (1 octet) en ISO-8859 et $c3$bb (2 octets) en UTF-8. J'ai donc volontairement mis des caractères non accentués dans le code.

```
        mov     rax, QWORD PTR [rcx+rax*8]
        ret     0
get_month1 ENDP
```

Le code est très simple:

- La première instruction MOVSXD copie une valeur 32-bit depuis ECX (où l'argument $month$ est passé) dans RAX avec extension du signe (car l'argument $month$ est de type *int*).

  La raison de l'extension du signe est que cette valeur 32-bit va être utilisée dans des calculs avec d'autres valeurs 64-bit.

  C'est pourquoi il doit être étendu à 64-bit[142].
- Ensuite l'adresse du pointeur de la table est chargée dans RCX.
- Enfin, la valeur d'entrée ($month$) est multipliée par 8 et ajoutée à l'adresse. Effectivement: nous sommes dans un environnement 64-bit et toutes les adresses (ou pointeurs) nécessitent exactement 64 bits (ou 8 octets) pour être stockées. C'est pourquoi chaque élément de la table a une taille de 8 octets. Et c'est pourquoi pour prendre un élément spécifique, $month * 8$ octets doivent être passés depuis le début. C'est ce que fait MOV. De plus, cette instruction charge également l'élément à cette adresse. Pour 1, l'élément sera un pointeur sur la chaîne qui contient « février », etc.

GCC 4.9 avec optimisation peut faire encore mieux[143] :

Listing 1.232: GCC 4.9 avec optimisation x64

```
        movsx   rdi, edi
        mov     rax, QWORD PTR month1[0+rdi*8]
        ret
```

MSVC 32-bit

Compilons-le aussi avec le compilateur MSVC 32-bit:

Listing 1.233: MSVC 2013 avec optimisation x86

```
_month$ = 8
_get_month1 PROC
        mov     eax, DWORD PTR _month$[esp-4]
        mov     eax, DWORD PTR _month1[eax*4]
        ret     0
_get_month1 ENDP
```

La valeur en entrée n'a pas besoin d'être étendue sur 64-bit, donc elle est utilisée telle quelle.

Et elle est multipliée par 4, car les éléments de la table sont larges de 32-bit (ou 4 octets).

ARM 32-bit

ARM en mode ARM

Listing 1.234: avec optimisation Keil 6/2013 (Mode ARM)

```
get_month1 PROC
        LDR      r1,|L0.100|
        LDR      r0,[r1,r0,LSL #2]
        BX       lr
        ENDP

|L0.100|
        DCD      ||.data||
```

[142] C'est parfois bizarre, mais des indices négatifs de tableau peuvent être passés par $month$ (les indices négatifs de tableaux sont expliqués plus loin: **??** on page ??). Et si cela arrive, la valeur entrée négative de type *int* est étendue correctement et l'élément correspondant avant le tableau est sélectionné. Ça ne fonctionnera pas correctement sans l'extension du signe.

[143] « 0+ » a été laissé dans le listing car la sortie de l'assembleur GCC n'est pas assez soignée pour l'éliminer. C'est un *déplacement*, et il vaut zéro ici.

```
        DCB      "January",0
        DCB      "February",0
        DCB      "March",0
        DCB      "April",0
        DCB      "May",0
        DCB      "June",0
        DCB      "July",0
        DCB      "August",0
        DCB      "September",0
        DCB      "October",0
        DCB      "November",0
        DCB      "December",0

        AREA ||.data||, DATA, ALIGN=2
month1
        DCD      ||.conststring||
        DCD      ||.conststring||+0x8
        DCD      ||.conststring||+0x11
        DCD      ||.conststring||+0x17
        DCD      ||.conststring||+0x1d
        DCD      ||.conststring||+0x21
        DCD      ||.conststring||+0x26
        DCD      ||.conststring||+0x2b
        DCD      ||.conststring||+0x32
        DCD      ||.conststring||+0x3c
        DCD      ||.conststring||+0x44
        DCD      ||.conststring||+0x4d
```

L'adresse de la table est chargée en R1.

Tout le reste est effectué en utilisant juste une instruction LDR.

Puis la valeur en entrée est décalée de 2 vers la gauche (ce qui est la même chose que multiplier par 4), puis ajoutée à R1 (où se trouve l'adresse de la table) et enfin un élément de la table est chargé depuis cette adresse.

L'élément 32-bit de la table est chargé dans R1 depuis la table.

ARM en mode Thumb

Le code est essentiellement le même, mais moins dense, car le suffixe LSL ne peut pas être spécifié dans l'instruction LDR ici:

```
get_month1 PROC
        LSLS     r0,r0,#2
        LDR      r1,|L0.64|
        LDR      r0,[r1,r0]
        BX       lr
        ENDP
```

ARM64

Listing 1.235: GCC 4.9 avec optimisation ARM64

```
get_month1 :
        adrp    x1, .LANCHOR0
        add     x1, x1,  :lo12 :.LANCHOR0
        ldr     x0, [x1,w0,sxtw 3]
        ret

.LANCHOR0 = . + 0
        .type   month1, %object
        .size   month1, 96
month1 :
        .xword  .LC2
        .xword  .LC3
```

```
        .xword  .LC4
        .xword  .LC5
        .xword  .LC6
        .xword  .LC7
        .xword  .LC8
        .xword  .LC9
        .xword  .LC10
        .xword  .LC11
        .xword  .LC12
        .xword  .LC13
.LC2 :
        .string "January"
.LC3 :
        .string "February"
.LC4 :
        .string "March"
.LC5 :
        .string "April"
.LC6 :
        .string "May"
.LC7 :
        .string "June"
.LC8 :
        .string "July"
.LC9 :
        .string "August"
.LC10 :
        .string "September"
.LC11 :
        .string "October"
.LC12 :
        .string "November"
.LC13 :
        .string "December"
```

L'adresse de la table est chargée dans X1 en utilisant la paire ADRP/ADD.

Puis l'élément correspondant est choisi dans la table en utilisant seulement un LDR, qui prend W0 (le registre où l'argument d'entrée $month$ se trouve), le décale de 3 bits vers la gauche (ce qui est la même chose que de le multiplier par 8), étend son signe (c'est ce que le suffixe « sxtw » implique) et l'ajoute à X0. Enfin la valeur 64-bit est chargée depuis la table dans X0.

MIPS

Listing 1.236: GCC 4.4.5 avec optimisation (IDA)

```
get_month1 :
; charger l'adresse de la table dans $v0 :
                la      $v0, month1
; prendre la valeur en entrée et la multiplier par 4:
                sll     $a0, 2
; ajouter l'adresse de la table et la valeur multipliée :
                addu    $a0, $v0
; charger l'élément de la table à cette adresse dans $v0 :
                lw      $v0, 0($a0)
; sortir
                jr      $ra
                or      $at, $zero ; slot de délai de branchement, NOP

                .data # .data.rel.local
                .globl month1
month1 :         .word aJanuary           # "janvier"
                .word aFebruary          # "fevrier"
                .word aMarch             # "mars"
                .word aApril             # "avril"
                .word aMay               # "mai"
                .word aJune              # "juin"
                .word aJuly              # "juillet"
```

```
                .word aAugust           # "aout"
                .word aSeptember        # "septembre"
                .word aOctober          # "octobre"
                .word aNovember         # "novembre"
                .word aDecember         # "decembre"

                .data # .rodata.str1.4
aJanuary :       .ascii "janvier"<0>
aFebruary :      .ascii "fevrier"<0>
aMarch :         .ascii "mars"<0>
aApril :         .ascii "avril"<0>
aMay :           .ascii "mai"<0>
aJune :          .ascii "juin"<0>
aJuly :          .ascii "juillet"<0>
aAugust :        .ascii "aout"<0>
aSeptember :     .ascii "septembre"<0>
aOctober :       .ascii "octobre"<0>
aNovember :      .ascii "novembre"<0>
aDecember :      .ascii "decembre"<0>
```

Débordement de tableau

Notre fonction accepte des valeurs dans l'intervalle 0..11, mais que se passe-t-il si 12 est passé? Il n'y a pas d'élément dans la table à cet endroit.

Donc la fonction va charger la valeur qui se trouve là, et la renvoyer.

Peu après, une autre fonction pourrait essayer de lire une chaîne de texte depuis cette adresse et pourrait planter.

Compilons l'exemple dans MSVC pour win64 et ouvrons le dans IDA pour voir ce que l'éditeur de lien à stocker après la table:

Listing 1.237: Fichier exécutable dans IDA

```
off_140011000   dq offset aJanuary_1    ; DATA XREF : .text :0000000140001003
                                        ; "January"
                dq offset aFebruary_1   ; "February"
                dq offset aMarch_1      ; "March"
                dq offset aApril_1      ; "April"
                dq offset aMay_1        ; "May"
                dq offset aJune_1       ; "June"
                dq offset aJuly_1       ; "July"
                dq offset aAugust_1     ; "August"
                dq offset aSeptember_1  ; "September"
                dq offset aOctober_1    ; "October"
                dq offset aNovember_1   ; "November"
                dq offset aDecember_1   ; "December"
aJanuary_1      db 'January',0          ; DATA XREF : sub_140001020+4
                                        ; .data :off_140011000
aFebruary_1     db 'February',0         ; DATA XREF : .data :0000000140011008
                align 4
aMarch_1        db 'March',0            ; DATA XREF : .data :0000000140011010
                align 4
aApril_1        db 'April',0            ; DATA XREF : .data :0000000140011018
```

Les noms des mois se trouvent juste après.

Notre programme est minuscule, il n'y a donc pas beaucoup de données à mettre dans le segment de données, juste les noms des mois. Mais il faut noter qu'il peut y avoir ici vraiment *n'importe quoi* que l'éditeur de lien aurait décidé d'y mettre.

Donc, que se passe-t-il si nous passons 12 à la fonction? Le 13ème élément va être renvoyé.

Voyons comment le CPU traite les octets en une valeur 64-bit:

Listing 1.238: Fichier exécutable dans IDA

```
off_140011000   dq offset qword_140011060
                                        ; DATA XREF : .text :0000000140001003
```

```
                dq offset aFebruary_1   ; "February"
                dq offset aMarch_1      ; "March"
                dq offset aApril_1      ; "April"
                dq offset aMay_1        ; "May"
                dq offset aJune_1       ; "June"
                dq offset aJuly_1       ; "July"
                dq offset aAugust_1     ; "August"
                dq offset aSeptember_1  ; "September"
                dq offset aOctober_1    ; "October"
                dq offset aNovember_1   ; "November"
                dq offset aDecember_1   ; "December"
qword_140011060 dq 797261756E614Ah      ; DATA XREF : sub_140001020+4
                                        ; .data :off_140011000
aFebruary_1     db 'February',0         ; DATA XREF : .data :0000000140011008
                align 4
aMarch_1        db 'March',0            ; DATA XREF : .data :0000000140011010
```

Et c'est 0x797261756E614A.

Peu après, une autre fonction (supposons, une qui traite des chaînes) pourrait essayer de lire des octets à cette adresse, y attendant une chaîne-C.

Plus probablement, ça planterait, car cette valeur ne ressemble pas à une adresse valide.

Protection contre les débordements de tampon

> Si quelque chose peut mal tourner, ça tournera mal
>
> Loi de Murphy

Il est un peu naïf de s'attendre à ce que chaque programmeur qui utilisera votre fonction ou votre bibliothèque ne passera jamais un argument plus grand que 11.

Il existe une philosophie qui dit « échouer tôt et échouer bruyamment » ou « échouer rapidement », qui enseigne de remonter les problèmes le plus tôt possible et d'arrêter.

Une telle méthode en C/C++ est les assertions.

Nous pouvons modifier notre programme pour qu'il échoue si une valeur incorrecte est passée:

Listing 1.239: assert() ajoutée

```
const char* get_month1_checked (int month)
{
        assert (month<12);
        return month1[month];
};
```

La macro assertion vérifie que la validité des valeurs à chaque démarrage de fonction et échoue si l'expression est fausse.

Listing 1.240: MSVC 2013 x64 avec optimisation

```
$SG3143 DB      'm', 00H, 'o', 00H, 'n', 00H, 't', 00H, 'h', 00H, '.', 00H
        DB      'c', 00H, 00H, 00H
$SG3144 DB      'm', 00H, 'o', 00H, 'n', 00H, 't', 00H, 'h', 00H, '<', 00H
        DB      '1', 00H, '2', 00H, 00H, 00H

month$ = 48
get_month1_checked PROC
$LN5 :
        push    rbx
        sub     rsp, 32
        movsxd  rbx, ecx
        cmp     ebx, 12
        jl      SHORT $LN3@get_month1
        lea     rdx, OFFSET FLAT :$SG3143
        lea     rcx, OFFSET FLAT :$SG3144
        mov     r8d, 29
```

```
        call    _wassert
$LN3@get_month1 :
        lea     rcx, OFFSET FLAT :month1
        mov     rax, QWORD PTR [rcx+rbx*8]
        add     rsp, 32
        pop     rbx
        ret     0
get_month1_checked ENDP
```

En fait, assert() n'est pas une fonction, mais une macro. Elle teste une condition, puis passe le numéro de ligne et le nom du fichier à une autre fonction qui rapporte cette information à l'utilisateur.

Ici nous voyons qu'à la fois le nom du fichier et la condition sont encodés en UTF-16. Le numéro de ligne est aussi passé (c'est 29).

Le mécanisme est sans doute le même dans tous les compilateurs. Voici ce que fait GCC:

Listing 1.241: GCC 4.9 x64 avec optimisation

```
.LC1 :
        .string "month.c"
.LC2 :
        .string "month<12"

get_month1_checked :
        cmp     edi, 11
        jg      .L6
        movsx   rdi, edi
        mov     rax, QWORD PTR month1[0+rdi*8]
        ret
.L6 :
        push    rax
        mov     ecx, OFFSET FLAT :__PRETTY_FUNCTION__.2423
        mov     edx, 29
        mov     esi, OFFSET FLAT :.LC1
        mov     edi, OFFSET FLAT :.LC2
        call    __assert_fail

__PRETTY_FUNCTION__.2423:
        .string "get_month1_checked"
```

Donc la macro dans GCC passe aussi le nom de la fonction par commodité.

Rien n'est vraiment gratuit, et c'est également vrai pour les tests de validité.

Ils rendent votre programme plus lent, en particulier si la macro assert() est utilisée dans des petites fonctions à durée critique.

Donc MSCV, par exemple, laisse les tests dans les compilations debug, mais ils disparaissent dans celles de release.

Les noyaux de Microsoft Windows NT existent en versions « checked » et « free ». [144].

Le premier a des tests de validation (d'où, « checked »), le second n'en a pas (d'où, « free/libre » de tests).

Bien sûr, le noyau « checked » fonctionne plus lentement à cause de ces tests, donc il n'est utilisé que pour des sessions de debug.

Accéder à un caractère spécifique

Un tableau de pointeurs sur des chaînes peut être accédé comme ceci[145] :

```
#include <stdio.h>

const char* month[]=
{
        "janvier", "fevrier", "mars", "avril",
        "mai", "juin", "juillet", "aout",
```

[144] msdn.microsoft.com/en-us/library/windows/hardware/ff543450(v=vs.85).aspx
[145] Lisez l'avertissement dans la NDT ici 1.20.5 on page 288

```
        "septembre", "octobre", "novembre", "decembre"
};

int main()
{
        // 4ème mois, 5ème caractère :
        printf ("%c\n", month[3][4]);
};
```

...puisque l'expression *month[3]* a un type *const char**. Et donc, le 5ème caractère est extrait de cette expression en ajoutant 4 octets à cette adresse.

À propos, la liste d'arguments passée à la fonction *main()* a le même type de données:

```
#include <stdio.h>

int main(int argc, char *argv[])
{
        printf ("3ème argument, 2ème caractère : %c\n", argv[3][1]);
};
```

Il est très important de comprendre, que, malgré la syntaxe similaire, c'est différent d'un tableau à deux dimensions, dont nous allons parler plus tard.

Une autre chose importante à noter: les chaînes considérées doivent être encodées dans un système où chaque caractère occupe un seul octet, comme l'ASCII[146] ou l'ASCII étendu. UTF-8 ne fonctionnera pas ici.

1.20.6 Tableaux multidimensionnels

En interne, un tableau multidimensionnel est pratiquement la même chose qu'un tableau linéaire.

Puisque la mémoire d'un ordinateur est linéaire, c'est un tableau uni-dimensionnel. Par commodité, ce tableau multidimensionnel peut facilement être représenté comme un uni-dimensionnel.

Par exemple, voici comment les éléments du tableau 3*4 sont placés dans un tableau uni-dimensionnel de 12 éléments:

Offset en mémoire	élément du tableau
0	[0][0]
1	[0][1]
2	[0][2]
3	[0][3]
4	[1][0]
5	[1][1]
6	[1][2]
7	[1][3]
8	[2][0]
9	[2][1]
10	[2][2]
11	[2][3]

Tab. 1.3: Tableau en deux dimensions représenté en mémoire en une dimension

Voici comment chacun des éléments du tableau 3*4 sont placés en mémoire:

0	1	2	3
4	5	6	7
8	9	10	11

Tab. 1.4: Adresse mémoire de chaque élément d'un tableau à deux dimensions

Donc, afin de calculer l'adresse de l'élément voulu, nous devons d'abord multiplier le premier index par 4 (largeur du tableau) et puis ajouter le second index. Ceci est appelé *row-major order* (ordre ligne d'abord),

[146] American Standard Code for Information Interchange

et c'est la méthode de représentation des tableaux et des matrices au moins en C/C++ et Python. Le terme *row-major order* est de l'anglais signifiant: « d'abord, écrire les éléments de la première ligne, puis ceux de la seconde ligne ...et enfin les éléments de la dernière ligne ».

Une autre méthode de représentation est appelée *column-major order* (ordre colonne d'abord) (les indices du tableau sont utilisés dans l'ordre inverse) et est utilisé au moins en ForTran, MATLAB et R. Le terme *column-major order* est de l'anglais signifiant: « d'abord, écrire les éléments de la première colonne, puis ceux de la seconde colonne ...et enfin les éléments de la dernière colonne ».

Quelle méthode est la meilleure?

En général, en termes de performance et de mémoire cache, le meilleur schéma pour l'organisation des données est celui dans lequel les éléments sont accédés séquentiellement.

Donc si votre fonction accède les données par ligne, *row-major order* est meilleur, et vice-versa.

Exemple de tableau à 2 dimensions

Nous allons travailler avec un tableau de type *char*, qui implique que chaque élément n'a besoin que d'un octet en mémoire.

Exemple de remplissage d'une ligne

Remplissons la seconde ligne avec les valeurs 0..3:

Listing 1.242: Exemple de remplissage d'une ligne

```
#include <stdio.h>

char a[3][4];

int main()
{
        int x, y;

        // effacer le tableau
        for (x=0; x<3; x++)
                for (y=0; y<4; y++)
                        a[x][y]=0;

        // remplir la 2ème ligne avec 0..3
        for (y=0; y<4; y++)
                a[1][y]=y;
};
```

Les trois lignes sont entourées en rouge. Nous voyons que la seconde ligne a maintenant les valeurs 0, 1, 2 et 3:

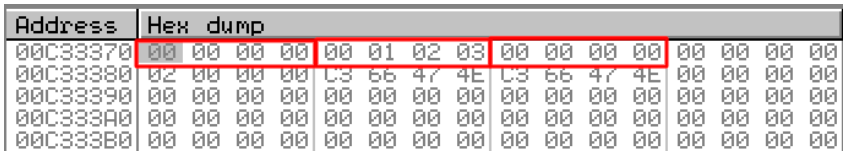


Fig. 1.92: OllyDbg : le tableau est rempli

Exemple de remplissage d'une colonne

Remplissons la troisième colonne avec les valeurs: 0..2:

Listing 1.243: Exemple de remplissage d'une colonne

```
#include <stdio.h>

char a[3][4];
```

```
int main()
{
        int x, y;

        // effacer le tableau
        for (x=0; x<3; x++)
                for (y=0; y<4; y++)
                        a[x][y]=0;

        // remplir la troisième colonne avec 0..2:
        for (x=0; x<3; x++)
                a[x][2]=x;
};
```

Les trois lignes sont entourées en rouge ici.

Nous voyons que dans chaque ligne, à la troisième position, ces valeurs sont écrites: 0, 1 et 2.

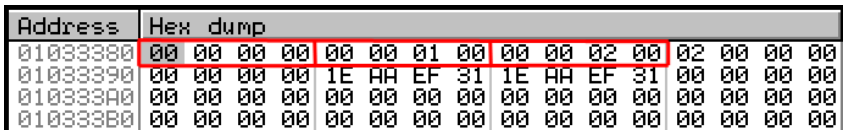


Fig. 1.93: OllyDbg : le tableau est rempli

Accéder à un tableau en deux dimensions comme un à une dimension

Nous pouvons facilement nous assurer qu'il est possible d'accéder à un tableau en deux dimensions d'au moins deux façons:

```
#include <stdio.h>

char a[3][4];

char get_by_coordinates1 (char array[3][4], int a, int b)
{
        return array[a][b];
};

char get_by_coordinates2 (char *array, int a, int b)
{
        // traiter le tableau en entrée comme uni-dimensionnel
        // 4 est ici la largeur du tableau
        return array[a*4+b];
};

char get_by_coordinates3 (char *array, int a, int b)
{
        // traiter le tableau en entrée comme un pointeur
        // calculer l'adresse, y prendre une valeur
        // 4 est ici la largeur du tableau
        return *(array+a*4+b);
};

int main()
{
        a[2][3]=123;
        printf ("%d\n", get_by_coordinates1(a, 2, 3));
        printf ("%d\n", get_by_coordinates2(a, 2, 3));
        printf ("%d\n", get_by_coordinates3(a, 2, 3));
};
```

Compilons[147] le et lançons le: il montre des valeurs correctes.

Ce que MSVC 2013 a généré est fascinant, les trois routines sont les mêmes!

[147] Ce programme doit être compilé comme un programme C, pas C++, sauvegardez-le dans un fichier avecl'extention .c pour le compiler avec MSVC

Listing 1.244: MSVC 2013 avec optimisation x64

```
array$ = 8
a$ = 16
b$ = 24
get_by_coordinates3 PROC
; RCX=adresse du tableau
; RDX=a
; R8=b
        movsxd  rax, r8d
; EAX=b
        movsxd  r9, edx
; R9=a
        add     rax, rcx
; RAX=b+adresse du tableau
        movzx   eax, BYTE PTR [rax+r9*4]
; AL=charger l'octet à l'adresse RAX+R9*4=b+adresse du tableau+a*4=adresse du tableau+a*4+b
        ret     0
get_by_coordinates3 ENDP

array$ = 8
a$ = 16
b$ = 24
get_by_coordinates2 PROC
        movsxd  rax, r8d
        movsxd  r9, edx
        add     rax, rcx
        movzx   eax, BYTE PTR [rax+r9*4]
        ret     0
get_by_coordinates2 ENDP

array$ = 8
a$ = 16
b$ = 24
get_by_coordinates1 PROC
        movsxd  rax, r8d
        movsxd  r9, edx
        add     rax, rcx
        movzx   eax, BYTE PTR [rax+r9*4]
        ret     0
get_by_coordinates1 ENDP
```

GCC génère des routines équivalentes, mais légèrement différentes:

Listing 1.245: GCC 4.9 x64 avec optimisation

```
; RDI=adresse du tableau
; RSI=a
; RDX=b

get_by_coordinates1 :
; étendre le signe sur 64-bit des valeurs 32-bit en entrée "a" et "b"
        movsx   rsi, esi
        movsx   rdx, edx
        lea     rax, [rdi+rsi*4]
; RAX=RDI+RSI*4=adresse du tableau+a*4
        movzx   eax, BYTE PTR [rax+rdx]
; AL=charger l'octet à l'adresse RAX+RDX=adresse du tableau+a*4+b
        ret

get_by_coordinates2 :
        lea     eax, [rdx+rsi*4]
; RAX=RDX+RSI*4=b+a*4
        cdqe
        movzx   eax, BYTE PTR [rdi+rax]
; AL=charger l'octet à l'adresse RDI+RAX=adresse du tableau+b+a*4
        ret

get_by_coordinates3 :
        sal     esi, 2
; ESI=a<<2=a*4
```

```
; étendre le signe sur 64-bit des valeurs 32-bit en entrée "a*4" et "b"
        movsx   rdx, edx
        movsx   rsi, esi
        add     rdi, rsi
; RDI=RDI+RSI=adresse du tableau+a*4
        movzx   eax, BYTE PTR [rdi+rdx]
; AL=charger l'octet à l'adresse RDI+RAX=adresse du tableau+a*4+b
        ret
```

Exemple de tableau à trois dimensions

C'est la même chose pour des tableaux multidimensionnels.

Nous allons travailler avec des tableaux de type *int* : chaque élément nécessite 4 octets en mémoire.

Voyons ceci:

Listing 1.246: simple exemple

```
#include <stdio.h>

int a[10][20][30];

void insert(int x, int y, int z, int value)
{
        a[x][y][z]=value;
};
```

x86

Nous obtenons (MSVC 2010):

Listing 1.247: MSVC 2010

```
_DATA    SEGMENT
COMM     _a :DWORD :01770H
_DATA    ENDS
PUBLIC   _insert
_TEXT    SEGMENT
_x$ = 8                 ; taille = 4
_y$ = 12                ; taille = 4
_z$ = 16                ; taille = 4
_value$ = 20            ; taille = 4
_insert    PROC
    push   ebp
    mov    ebp, esp
    mov    eax, DWORD PTR _x$[ebp]
    imul   eax, 2400                    ; eax=600*4*x
    mov    ecx, DWORD PTR _y$[ebp]
    imul   ecx, 120                     ; ecx=30*4*y
    lea    edx, DWORD PTR _a[eax+ecx]   ; edx=a + 600*4*x + 30*4*y
    mov    eax, DWORD PTR _z$[ebp]
    mov    ecx, DWORD PTR _value$[ebp]
    mov    DWORD PTR [edx+eax*4], ecx   ; *(edx+z*4)=valeur
    pop    ebp
    ret    0
_insert    ENDP
_TEXT      ENDS
```

Rien de particulier. Pour le calcul de l'index, trois arguments en entrée sont utilisés dans la formule $address = 600 \cdot 4 \cdot x + 30 \cdot 4 \cdot y + 4z$, pour représenter le tableau comme multidimensionnel. N'oubliez pas que le type *int* est 32-bit (4 octets), donc tous les coefficients doivent être multipliés par 4.

Listing 1.248: GCC 4.4.1

```
        public insert
insert  proc near
```

```
x        = dword ptr  8
y        = dword ptr  0Ch
z        = dword ptr  10h
value    = dword ptr  14h

         push    ebp
         mov     ebp, esp
         push    ebx
         mov     ebx, [ebp+x]
         mov     eax, [ebp+y]
         mov     ecx, [ebp+z]
         lea     edx, [eax+eax]    ; edx=y*2
         mov     eax, edx          ; eax=y*2
         shl     eax, 4            ; eax=(y*2)<<4 = y*2*16 = y*32
         sub     eax, edx          ; eax=y*32 - y*2=y*30
         imul    edx, ebx, 600     ; edx=x*600
         add     eax, edx          ; eax=eax+edx=y*30 + x*600
         lea     edx, [eax+ecx]    ; edx=y*30 + x*600 + z
         mov     eax, [ebp+value]
         mov     dword ptr ds :a[edx*4], eax  ; *(a+edx*4)=valeur
         pop     ebx
         pop     ebp
         retn
insert   endp
```

Le compilateur GCC fait cela différemment.

Pour une des opérations du calcul ($30y$), GCC produit un code sans instruction de multiplication. Voici comment il fait: $(y+y) \ll 4 - (y+y) = (2y) \ll 4 - 2y = 2 \cdot 16 \cdot y - 2y = 32y - 2y = 30y$. Ainsi, pour le calcul de $30y$, seulement une addition, un décalage de bit et une soustraction sont utilisés. Ceci fonctionne plus vite.

ARM + sans optimisation Xcode 4.6.3 (LLVM) (Mode Thumb)

Listing 1.249: sans optimisation Xcode 4.6.3 (LLVM) (Mode Thumb)

```
_insert

value   = -0x10
z       = -0xC
y       = -8
x       = -4

; allouer de l'espace sur la pile locale pour 4 valeurs de type int
SUB     SP, SP, #0x10
MOV     R9, 0xFC2 ; a
ADD     R9, PC
LDR.W   R9, [R9] ; prendre le pointeur sur le tableau
STR     R0, [SP,#0x10+x]
STR     R1, [SP,#0x10+y]
STR     R2, [SP,#0x10+z]
STR     R3, [SP,#0x10+value]
LDR     R0, [SP,#0x10+value]
LDR     R1, [SP,#0x10+z]
LDR     R2, [SP,#0x10+y]
LDR     R3, [SP,#0x10+x]
MOV     R12, 2400
MUL.W   R3, R3, R12
ADD     R3, R9
MOV     R9, 120
MUL.W   R2, R2, R9
ADD     R2, R3
LSLS    R1, R1, #2 ; R1=R1<<2
ADD     R1, R2
STR     R0, [R1]   ; R1 - adresse de l'élément du tableau
; libérer le chunk sur la pile locale, alloué pour 4 valeurs de type int
ADD     SP, SP, #0x10
BX      LR
```

LLVM sans optimisation sauve toutes les variables dans la pile locale, ce qui est redondant.

L'adresse de l'élément du tableau est calculée par la formule vue précédemment.

ARM + avec optimisation Xcode 4.6.3 (LLVM) (Mode Thumb)

Listing 1.250: avec optimisation Xcode 4.6.3 (LLVM) (Mode Thumb)

```
_insert
MOVW    R9, #0x10FC
MOV.W   R12, #2400
MOVT.W  R9, #0
RSB.W   R1, R1, R1,LSL#4   ; R1 - y. R1=y<<4 - y = y*16 - y = y*15
ADD     R9, PC
LDR.W   R9, [R9]           ; R9 = pointeur sur la tableau a
MLA.W   R0, R0, R12, R9    ; R0 - x, R12 - 2400, R9 - pointeur sur a. R0=x*2400 + ptr sur a
ADD.W   R0, R0, R1,LSL#3   ; R0 = R0+R1<<3 = R0+R1*8 = x*2400 + ptr sur a + y*15*8 =
                           ; ptr sur a + y*30*4 + x*600*4
STR.W   R3, [R0,R2,LSL#2] ; R2 - z, R3 - valeur. adresse=R0+z*4 =
                           ; ptr sur a + y*30*4 + x*600*4 + z*4
BX      LR
```

L'astuce de remplacer la multiplication par des décalage, addition et soustraction que nous avons déjà vue est aussi utilisée ici.

Ici, nous voyons aussi une nouvelle instruction: RSB (*Reverse Subtract*).

Elle fonctionne comme SUB, mais échange ses opérandes l'un avec l'autre avant l'exécution. Pourquoi? SUB et RSB sont des instructions auxquelles un coefficient de décalage peut être appliqué au second opérande: (LSL#4).

Mais ce coefficient ne peut être appliqué qu'au second opérande.

C'est bien pour des opérations commutatives comme l'addition ou la multiplication (les opérandes peuvent être échangés sans changer le résultat).

Mais la soustraction est une opération non commutative, donc RSB existe pour ces cas.

MIPS

Mon exemple est minuscule, donc le compilateur GCC a décidé de mettre le tableau a dans la zone de 64KiB adressable par le Global Pointer.

Listing 1.251: GCC 4.4.5 avec optimisation (IDA)

```
insert :
; $a0=x
; $a1=y
; $a2=z
; $a3=valeur
                sll     $v0, $a0, 5
; $v0 = $a0<<5 = x*32
                sll     $a0, 3
; $a0 = $a0<<3 = x*8
                addu    $a0, $v0
; $a0 = $a0+$v0 = x*8+x*32 = x*40
                sll     $v1, $a1, 5
; $v1 = $a1<<5 = y*32
                sll     $v0, $a0, 4
; $v0 = $a0<<4 = x*40*16 = x*640
                sll     $a1, 1
; $a1 = $a1<<1 = y*2
                subu    $a1, $v1, $a1
; $a1 = $v1-$a1 = y*32-y*2 = y*30
                subu    $a0, $v0, $a0
; $a0 = $v0-$a0 = x*640-x*40 = x*600
                la      $gp, __gnu_local_gp
                addu    $a0, $a1, $a0
```

```
; $a0 = $a1+$a0 = y*30+x*600
                addu    $a0, $a2
; $a0 = $a0+$a2 = y*30+x*600+z
; charger l'adresse de la table :
                lw      $v0, (a & 0xFFFF)($gp)
; multiplier l'index par 4 pour avancer d'un élément du tableau :
                sll     $a0, 2
; ajouter l'index multiplié et l'adresse de la table :
                addu    $a0, $v0, $a0
; stocker la valeur dans la table et retourner :
                jr      $ra
                sw      $a3, 0($a0)

                .comm a :0x1770
```

Plus d'exemples

L'écran de l'ordinateur est représenté comme un tableau 2D, mais le buffer vidéo est un tableau linéaire 1D. Nous en parlons ici: **??** on page ??.

Un autre exemple dans ce livre est le jeu Minesweeper: son champ est aussi un tableau à deux dimensions: **??**.

1.20.7 Ensemble de chaînes comme un tableau à deux dimensions

Retravaillons la fonction qui renvoie le nom d'un mois: liste.1.230.

Comme vous le voyez, au moins une opération de chargement en mémoire est nécessaire pour préparer le pointeur sur la chaîne représentant le nom du mois.

Est-il possible de se passer de cette opération de chargement en mémoire?

En fait oui, si vous représentez la liste de chaînes comme un tableau à deux dimensions:

```
#include <stdio.h>
#include <assert.h>

const char month2[12][10]=
{
        { 'j','a','n','v','i','e','r',  0,  0,  0 },
        { 'f','e','b','v','r','i','e','r',  0,  0 },
        { 'm','a','s',  0,  0,  0,  0,  0,  0,  0 },
        { 'a','v','r','i','l',  0,  0,  0,  0,  0 },
        { 'm','a','i',  0,  0,  0,  0,  0,  0,  0 },
        { 'j','u','i','n',  0,  0,  0,  0,  0,  0 },
        { 'j','u','i','l','l','e','t',  0,  0,  0 },
        { 'a','o','u','t',  0,  0,  0,  0,  0,  0 },
        { 's','e','p','t','e','m','b','r','e',  0 },
        { 'o','c','t','o','b','r','e',  0,  0,  0 },
        { 'n','o','v','e','m','b','r','e',  0,  0 },
        { 'd','e','c','e','m','b','r','e',  0,  0 }
};

// dans l'intervalle 0..11
const char* get_month2 (int month)
{
        return &month2[month][0];
};
```

Voici ce que nous obtenons:

Listing 1.252: MSVC 2013 x64 avec optimisation

```
month2  DB      04aH
        DB      061H
        DB      06eH
        DB      075H
        DB      061H
```

```
        DB      072H
        DB      079H
        DB      00H
        DB      00H
        DB      00H
...

get_month2 PROC
; étendre le signe de l'argument en entrée sur 64-bit
        movsxd  rax, ecx
        lea     rcx, QWORD PTR [rax+rax*4]
; RCX=mois+mois*4=mois*5
        lea     rax, OFFSET FLAT :month2
; RAX=pointeur sur la table
        lea     rax, QWORD PTR [rax+rcx*2]
; RAX=pointeur sur la table + RCX*2=pointeur sur la table + mois*5*2=pointeur sur la table + ↙
    ↳ mois*10
        ret     0
get_month2 ENDP
```

Il n'y a pas du tout d'accès à la mémoire.

Tout ce que fait cette fonction, c'est de calculer le point où le premier caractère du nom du mois se trouve: $pointeur\_sur\_la\_table + mois * 10$.

Il y a deux instructions LEA, qui fonctionnent en fait comme plusieurs instructions MUL et MOV.

La largeur du tableau est de 10 octets.

En effet, la chaîne la plus longue ici—« septembre »—fait 9 octets, plus l'indicateur de fin de chaîne 0, ça fait 10 octets

Le reste du nom de chaque mois est complété par des zéros, afin d'occuper le même espace (10 octets).

Donc, notre fonction fonctionne même plus vite, car toutes les chaînes débutent à une adresse qui peut être facilement calculée.

GCC 4.9 avec optimisation fait encore plus court:

Listing 1.253: GCC 4.9 x64 avec optimisation

```
        movsx   rdi, edi
        lea     rax, [rdi+rdi*4]
        lea     rax, month2[rax+rax]
        ret
```

LEA est aussi utilisé ici pour la multiplication par 10.

Les compilateurs sans optimisations génèrent la multiplication différemment.

Listing 1.254: GCC 4.9 x64 sans optimisation

```
get_month2 :
        push    rbp
        mov     rbp, rsp
        mov     DWORD PTR [rbp-4], edi
        mov     eax, DWORD PTR [rbp-4]
        movsx   rdx, eax
; RDX = valeur entrée avec signe étendu
        mov     rax, rdx
; RAX = mois
        sal     rax, 2
; RAX = mois<<2 = mois*4
        add     rax, rdx
; RAX = RAX+RDX = mois*4+mois = mois*5
        add     rax, rax
; RAX = RAX*2 = mois*5*2 = mois*10
        add     rax, OFFSET FLAT :month2
; RAX = mois*10 + pointeur sur la table
        pop     rbp
        ret
```

MSVC sans optimisation utilise simplement l'instruction IMUL :

Listing 1.255: MSVC 2013 x64 sans optimisation

```
month$ = 8
get_month2 PROC
        mov     DWORD PTR [rsp+8], ecx
        movsxd  rax, DWORD PTR month$[rsp]
; RAX = étendre le signe de la valeur entrée sur 64-bit
        imul    rax, rax, 10
; RAX = RAX*10
        lea     rcx, OFFSET FLAT :month2
; RCX = pointeur sur la table
        add     rcx, rax
; RCX = RCX+RAX = pointeur sur la table+mois*10
        mov     rax, rcx
; RAX = pointeur sur la table+mois*10
        mov     ecx, 1
; RCX = 1
        imul    rcx, rcx, 0
; RCX = 1*0 = 0
        add     rax, rcx
; RAX = pointeur sur la table+mois*10 + 0 = pointeur sur la table+mois*10
        ret     0
get_month2 ENDP
```

Mais une chose est est curieuse: pourquoi ajouter une multiplication par zéro et ajouter zéro au résultat final?

Ceci ressemble à une bizarrerie du générateur de code du compilateur, qui n'a pas été détectée par les tests du compilateur (le code résultant fonctionne correctement après tout). Nous examinons volontairement de tels morceaux de code, afin que le lecteur prenne conscience qu'il ne doit parfois pas se casser la tête sur des artefacts de compilateur.

32-bit ARM

Keil avec optimisation pour le mode Thumb utilise l'instruction de multiplication `MULS` :

Listing 1.256: avec optimisation Keil 6/2013 (Mode Thumb)

```
; R0 = mois
        MOVS     r1,#0xa
; R1 = 10
        MULS     r0,r1,r0
; R0 = R1*R0 = 10*mois
        LDR      r1,|L0.68|
; R1 = pointer sur la table
        ADDS     r0,r0,r1
; R0 = R0+R1 = 10*mois + pointer sur la table
        BX       lr
```

Keil avec optimisation pour mode ARM utilise des instructions d'addition et de décalage:

Listing 1.257: avec optimisation Keil 6/2013 (Mode ARM)

```
; R0 = mois
        LDR      r1,|L0.104|
; R1 = pointeur sur la table
        ADD      r0,r0,r0,LSL #2
; R0 = R0+R0<<2 = R0+R0*4 = mois*5
        ADD      r0,r1,r0,LSL #1
; R0 = R1+R0<<2 = pointeur sur la table + mois*5*2 = pointeur sur la table + mois*10
        BX       lr
```

ARM64

Listing 1.258: GCC 4.9 ARM64 avec optimisation

```
; W0 = mois
        sxtw    x0, w0
```

```
; X0 = valeur entrée avec signe étendu
        adrp    x1, .LANCHOR1
        add     x1, x1,  :lo12 :.LANCHOR1
; X1 = pointeur sur la table
        add     x0, x0, x0, lsl 2
; X0 = X0+X0<<2 = X0+X0*4 = X0*5
        add     x0, x1, x0, lsl 1
; X0 = X1+X0<<1 = X1+X0*2 = pointeur sur la table + X0*10
        ret
```

SXTW est utilisée pour étendre le signe, convertir l'entrée 32-bit en 64-bit et stocker le résultat dans X0.

La paire ADRP/ADD est utilisée pour charger l'adresse de la table.

L'instruction ADD a aussi un suffixe LSL, qui aide avec les multiplications.

MIPS

Listing 1.259: GCC 4.4.5 avec optimisation (IDA)

```
                .globl get_mois2
get_mois2 :
; $a0=mois
                sll     $v0, $a0, 3
; $v0 = $a0<<3 = mois*8
                sll     $a0, 1
; $a0 = $a0<<1 = mois*2
                addu    $a0, $v0
; $a0 = mois*2+mois*8 = mois*10
; charger l'adresse de la table :
                la      $v0, mois2
; ajouter l'adresse de la table et l'index que nous avons calculé et sortir :
                jr      $ra
                addu    $v0, $a0

mois2 :         .ascii "janvier"<0>
                .byte 0, 0
aFebruary :      .ascii "fevrier"<0>
                .byte    0
aMarch :         .ascii "mars"<0>
                .byte 0, 0, 0, 0
aApril :         .ascii "avril"<0>
                .byte 0, 0, 0, 0
aMay :           .ascii "mai"<0>
                .byte 0, 0, 0, 0, 0, 0
aJune :          .ascii "juin"<0>
                .byte 0, 0, 0, 0, 0
aJuly :          .ascii "juillet"<0>
                .byte 0, 0, 0, 0, 0
aAugust :        .ascii "aout"<0>
                .byte 0, 0, 0
aSeptember :     .ascii "septembre"<0>
aOctober :       .ascii "octobre"<0>
                .byte 0, 0
aNovember :      .ascii "novembre"<0>
                .byte    0
aDecember :      .ascii "decembre"<0>
                .byte 0, 0, 0, 0, 0, 0, 0, 0, 0
```

Conclusion

C'est une technique surannée de stocker des chaînes de texte. Vous pouvez en trouver beaucoup dans Oracle RDBMS, par exemple. Il est difficile de dire si ça vaut la peine de le faire sur des ordinateurs modernes. Néanmoins, c'est un bon exemple de tableaux, donc il a été ajouté à ce livre.

1.20.8 Conclusion

Un tableau est un ensemble de données adjacentes en mémoire.

C'est vrai pour tout type d'élément, structures incluses.

Pour accéder à un élément spécifique d'un tableau, il suffit de calculer son adresse.

1.21 À propos

Donc, un pointeur sur un tableau et l'adresse de son premier élément—sont la même chose. C'est pourquoi les expressions `ptr[0]` et `*ptr` sont équivalentes en C/C++. Il est intéressant de noter que Hex-Rays remplace souvent la première par la seconde. Il procède ainsi lorsqu'il n'a aucune idée qu'il travaille avec un pointeur sur le tableau complet et pense que c'est un pointeur sur une seule variable.

1.21.1 Exercices

- http://challenges.re/62
- http://challenges.re/63
- http://challenges.re/64
- http://challenges.re/65
- http://challenges.re/66

1.22 Manipulation de bits spécifiques

Beaucoup de fonctions définissent leurs arguments comme des flags dans un champ de bits.

Bien sûr, ils pourraient être substitués par un ensemble de variables de type *bool*, mais ce n'est pas frugal.

1.22.1 Test d'un bit spécifique

x86

Exemple avec l'API win32:

```
        HANDLE fh;

        fh=CreateFile ("file", GENERIC_WRITE | GENERIC_READ, FILE_SHARE_READ, NULL, OPEN_ALWAYS⇙
    ↳ , FILE_ATTRIBUTE_NORMAL, NULL);
```

Nous obtenons (MSVC 2010):

Listing 1.260: MSVC 2010

```
        push    0
        push    128             ; 00000080H
        push    4
        push    0
        push    1
        push    -1073741824     ; c0000000H
        push    OFFSET $SG78813
        call    DWORD PTR __imp__CreateFileA@28
        mov     DWORD PTR _fh$[ebp], eax
```

Regardons dans WinNT.h:

Listing 1.261: WinNT.h

```
#define GENERIC_READ                     (0x80000000L)
#define GENERIC_WRITE                    (0x40000000L)
#define GENERIC_EXECUTE                  (0x20000000L)
#define GENERIC_ALL                      (0x10000000L)
```

Tout est clair, GENERIC_READ | GENERIC_WRITE = 0x80000000 | 0x40000000 = 0xC0000000, et c'est la valeur utilisée comme second argument pour la fonction `CreateFile()`[148].

Comment `CreateFile()` va tester ces flags?

Si nous regardons dans KERNEL32.DLL de Windows XP SP3 x86, nous trouverons ce morceau de code dans `CreateFileW` :

Listing 1.262: KERNEL32.DLL (Windows XP SP3 x86)

```
.text :7C83D429     test    byte ptr [ebp+dwDesiredAccess+3], 40h
.text :7C83D42D     mov     [ebp+var_8], 1
.text :7C83D434     jz      short loc_7C83D417
.text :7C83D436     jmp     loc_7C810817
```

Ici nous voyons l'instruction TEST, toutefois elle n'utilise pas complètement le second argument, mais seulement l'octet le plus significatif et le teste avec le flag `0x40` (ce qui implique le flag `GENERIC_WRITE` ici).

TEST est essentiellement la même chose que AND, mais sans sauver le résultat (rappelez vous le cas de CMP qui est la même chose que SUB, mais sans sauver le résultat (1.9.4 on page 86)).

La logique de ce bout de code est la suivante:

```
if ((dwDesiredAccess&0x40000000) == 0) goto loc_7C83D417
```

Si l'instruction AND laisse ce bit, le flag ZF sera mis à zéro et le saut conditionnel JZ ne sera pas effectué. Le saut conditionnel est effectué uniquement su la bit `0x40000000` est absent dans la variable `dwDesiredAccess` —auquel cas le résultat du AND est 0, ZF est mis à 1 et le saut conditionnel est effectué.

Essayons avec GCC 4.4.1 et Linux:

```
#include <stdio.h>
#include <fcntl.h>

void main()
{
        int handle;

        handle=open ("file", O_RDWR | O_CREAT);
};
```

Nous obtenons:

Listing 1.263: GCC 4.4.1

```
                public main
main            proc near

var_20          = dword ptr -20h
var_1C          = dword ptr -1Ch
var_4           = dword ptr -4

                push    ebp
                mov     ebp, esp
                and     esp, 0FFFFFFF0h
                sub     esp, 20h
                mov     [esp+20h+var_1C], 42h
                mov     [esp+20h+var_20], offset aFile ; "file"
                call    _open
                mov     [esp+20h+var_4], eax
                leave
                retn
main            endp
```

[148] msdn.microsoft.com/en-us/library/aa363858(VS.85).aspx

Si nous regardons dans la fonction open() de la bibliothèque libc.so.6, c'est seulement un appel système:

Listing 1.264: open() (libc.so.6)

```
.text :000BE69B     mov     edx, [esp+4+mode] ; mode
.text :000BE69F     mov     ecx, [esp+4+flags] ; flags
.text :000BE6A3     mov     ebx, [esp+4+filename] ; filename
.text :000BE6A7     mov     eax, 5
.text :000BE6AC     int     80h             ; LINUX - sys_open
```

Donc, le champ de bits pour open() est apparemment testé quelque part dans le noyau Linux.

Bien sûr, il est facile de télécharger le code source de la Glibc et du noyau Linux, mais nous voulons comprendre ce qui se passe sans cela.

Donc, à partir de Linux 2.6, lorsque l'appel système sys_open est appelé, le contrôle passe finalement à do_sys_open, et à partir de là—à la fonction do_filp_open() (elle est située ici fs/namei.c dans l'arborescence des sources du noyau).

N.B. Outre le passage des arguments par la pile, il y a aussi une méthode consistant à passer certains d'entre eux par des registres. Ceci est aussi appelé fastcall (4.1.3 on page 545). Ceci fonctionne plus vite puisque le CPU ne doit pas faire d'accès à la pile en mémoire pour lire la valeur des arguments. GCC a l'option *regparm*[149], avec laquelle il est possible de définir le nombre d'arguments qui peuvent être passés par des registres.

Le noyau Linux 2.6 est compilé avec l'option -mregparm=3 [150] [151].

Cela signifie que les 3 premiers arguments sont passés par les registres EAX, EDX et ECX, et le reste via la pile. Bien sûr, si le nombre d'arguments est moins que 3, seule une partie de ces registres seront utilisés.

Donc, téléchargeons le noyau Linux 2.6.31, compilons-le dans Ubuntu: make vmlinux, ouvrons-le dans IDA, et cherchons la fonction do_filp_open(). Au début, nous voyons (les commentaires sont les miens):

Listing 1.265: do_filp_open() (noyau Linux kernel 2.6.31)

```
do_filp_open    proc near
...
                push    ebp
                mov     ebp, esp
                push    edi
                push    esi
                push    ebx
                mov     ebx, ecx
                add     ebx, 1
                sub     esp, 98h
                mov     esi, [ebp+arg_4] ; acc_mode (5ème argument)
                test    bl, 3
                mov     [ebp+var_80], eax ; dfd (1er argument)
                mov     [ebp+var_7C], edx ; pathname (2ème argument)
                mov     [ebp+var_78], ecx ; open_flag (3ème argument)
                jnz     short loc_C01EF684
                mov     ebx, ecx        ; ebx <- open_flag
```

GCC sauve les valeurs des 3 premiers arguments dans la pile locale. Si cela n'était pas fait, le compilateur ne toucherait pas ces registres, et ça serait un environnement trop étroit pour l'allocateur de registres du compilateur.

Cherchons ce morceau de code:

Listing 1.266: do_filp_open() (noyau Linux 2.6.31)

```
loc_C01EF6B4 :       ; CODE XREF : do_filp_open+4F
                test    bl, 40h         ; O_CREAT
                jnz     loc_C01EF810
                mov     edi, ebx
                shr     edi, 11h
                xor     edi, 1
                and     edi, 1
```

[149] ohse.de/uwe/articles/gcc-attributes.html#func-regparm
[150] kernelnewbies.org/Linux_2_6_20#head-042c62f290834eb1fe0a1942bbf5bb9a4accbc8f
[151] Voir aussi le fichier arch/x86/include/asm/calling.h dans l'arborescence du noyau

```
                test    ebx, 10000h
                jz      short loc_C01EF6D3
                or      edi, 2
```

0x40—c'est ce à quoi est égale la macro O_CREAT. Le bit 0x40 de open_flag est testé, et si il est à 1, le saut de l'instruction JNZ suivante est effectué.

ARM

Le bit O_CREAT est testé différemment dans le noyau Linux 3.8.0.

Listing 1.267: noyau Linux 3.8.0

```
struct file *do_filp_open(int dfd, struct filename *pathname,
                const struct open_flags *op)
{
...
        filp = path_openat(dfd, pathname, &nd, op, flags | LOOKUP_RCU);
...
}

static struct file *path_openat(int dfd, struct filename *pathname,
                struct nameidata *nd, const struct open_flags *op, int flags)
{
...
        error = do_last(nd, &path, file, op, &opened, pathname);
...
}


static int do_last(struct nameidata *nd, struct path *path,
                   struct file *file, const struct open_flags *op,
                   int *opened, struct filename *name)
{
...
        if (!(open_flag & O_CREAT)) {
    ...
                error = lookup_fast(nd, path, &inode);
    ...
        } else {
    ...
                error = complete_walk(nd);
        }
...
}
```

Voici à quoi ressemble le noyau compilé pour le mode ARM dans IDA :

Listing 1.268: do_last() dans vmlinux (IDA)

```
...
.text :C0169EA8      MOV       R9, R3  ; R3 - (4th argument) open_flag
...
.text :C0169ED4      LDR       R6, [R9] ; R6 - open_flag
...
.text :C0169F68      TST       R6, #0x40 ; jumptable C0169F00 default case
.text :C0169F6C      BNE       loc_C016A128
.text :C0169F70      LDR       R2, [R4,#0x10]
.text :C0169F74      ADD       R12, R4, #8
.text :C0169F78      LDR       R3, [R4,#0xC]
.text :C0169F7C      MOV       R0, R4
.text :C0169F80      STR       R12, [R11,#var_50]
.text :C0169F84      LDRB      R3, [R2,R3]
.text :C0169F88      MOV       R2, R8
.text :C0169F8C      CMP       R3, #0
.text :C0169F90      ORRNE     R1, R1, #3
.text :C0169F94      STRNE     R1, [R4,#0x24]
.text :C0169F98      ANDS      R3, R6, #0x200000
.text :C0169F9C      MOV       R1, R12
```

```
.text :C0169FA0      LDRNE     R3, [R4,#0x24]
.text :C0169FA4      ANDNE     R3, R3, #1
.text :C0169FA8      EORNE     R3, R3, #1
.text :C0169FAC      STR       R3, [R11,#var_54]
.text :C0169FB0      SUB       R3, R11, #-var_38
.text :C0169FB4      BL        lookup_fast
...
.text :C016A128 loc_C016A128   ; CODE XREF : do_last.isra.14+DC
.text :C016A128      MOV       R0, R4
.text :C016A12C      BL        complete_walk
...
```

TST est analogue à l'instruction TEST en x86. Nous pouvons « pointer » visuellement ce morceau de code grâce au fait que la fonction `lookup_fast()` doit être exécutée dans un cas et `complete_walk()` dans l'autre. Ceci correspond au code source de la fonction `do_last()`. La macro `O_CREAT` vaut `0x40` ici aussi.

1.22.2 Mettre (à 1) et effacer (à 0) des bits spécifiques

Par exemple:

```
#include <stdio.h>

#define IS_SET(flag, bit)       ((flag) & (bit))
#define SET_BIT(var, bit)       ((var) |= (bit))
#define REMOVE_BIT(var, bit)    ((var) &= ~(bit))

int f(int a)
{
    int rt=a;

    SET_BIT (rt, 0x4000);
    REMOVE_BIT (rt, 0x200);

    return rt;
};

int main()
{
    f(0x12340678);
};
```

x86

MSVC sans optimisation

Nous obtenons (MSVC 2010):

Listing 1.269: MSVC 2010

```
_rt$ = -4         ; size = 4
_a$ = 8           ; size = 4
_f  PROC
    push   ebp
    mov    ebp, esp
    push   ecx
    mov    eax, DWORD PTR _a$[ebp]
    mov    DWORD PTR _rt$[ebp], eax
    mov    ecx, DWORD PTR _rt$[ebp]
    or     ecx, 16384                ; 00004000H
    mov    DWORD PTR _rt$[ebp], ecx
    mov    edx, DWORD PTR _rt$[ebp]
    and    edx, -513                 ; fffffdffH
    mov    DWORD PTR _rt$[ebp], edx
    mov    eax, DWORD PTR _rt$[ebp]
    mov    esp, ebp
    pop    ebp
```

```
    ret    0
_f  ENDP
```

L'instruction OR met un bit à la valeur 1 tout en ignorant les autres bits.

AND annule un bit. On peut dire que AND copie simplement tous les bits sauf un. En effet, dans le second opérande du AND seuls les bits qui doivent être sauvés sont mis (à 1), seul celui qu'on ne veut pas copier ne l'est pas (il est à 0 dans le bitmask). C'est la manière la plus facile de mémoriser la logique.

OllyDbg

Essayons cet exemple dans OllyDbg.

Tout d'abord, regardons la forme binaire de la constante que nous allons utiliser:

`0x200` (0b00000000000000000001000000000) (i.e., le 10ème bit (en comptant depuis le 1er)).

`0x200` inversé est `0xFFFFFDFF` (0b11111111111111111110111111111).

`0x4000` (0b00000000000000100000000000000) (i.e., le 15ème bit).

La valeur d'entrée est: `0x12340678` (0b10010001101000000011001111000). Nous voyons comment elle est chargée:

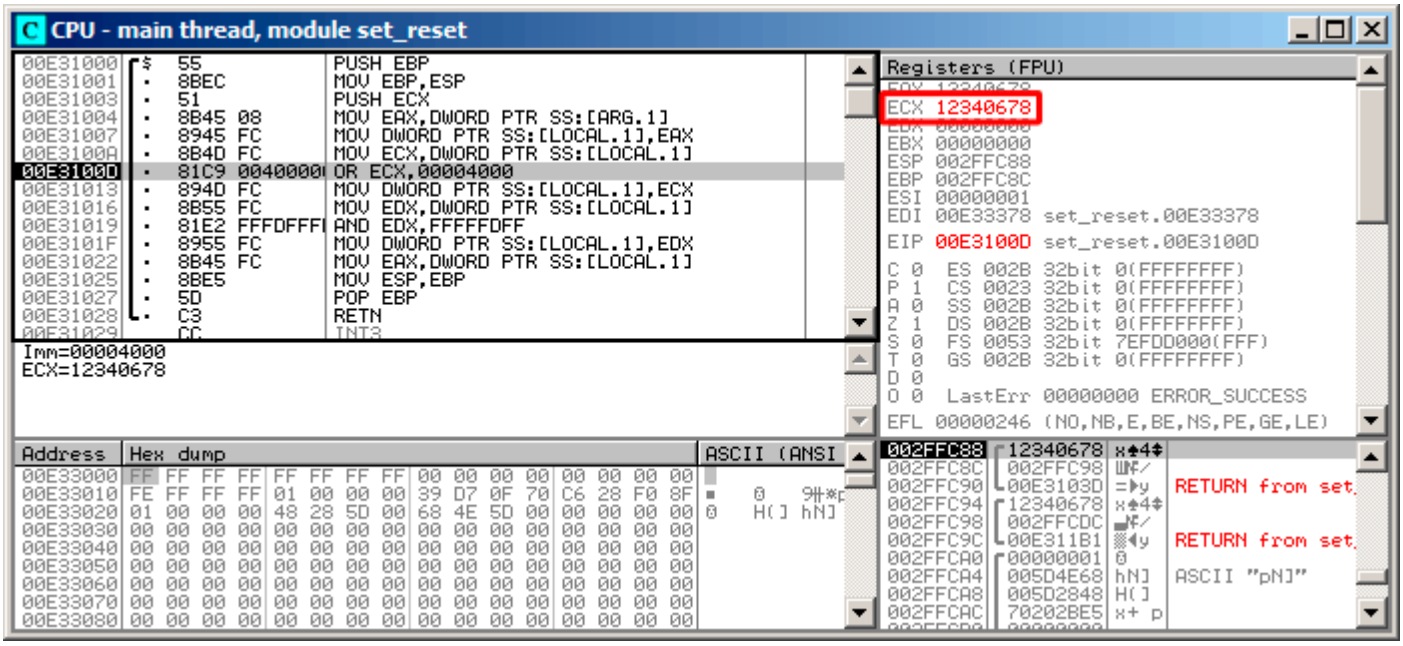


Fig. 1.94: OllyDbg : valeur chargée dans ECX

0R exécuté:

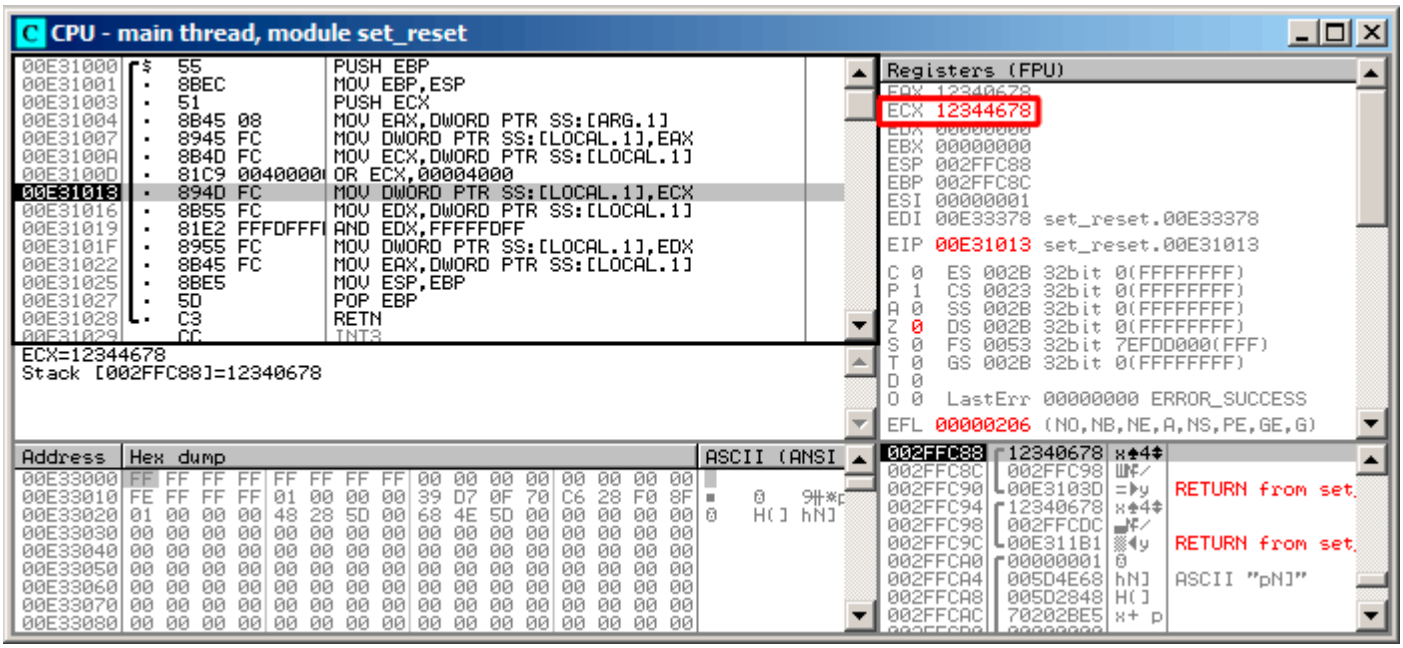


Fig. 1.95: OllyDbg : 0R exécuté

Le 15ème bit est mis: 0x12344678 (0b10010001101000100011001111000).

La valeur est encore rechargée (car le compilateur n'est pas en mode avec optimisation):

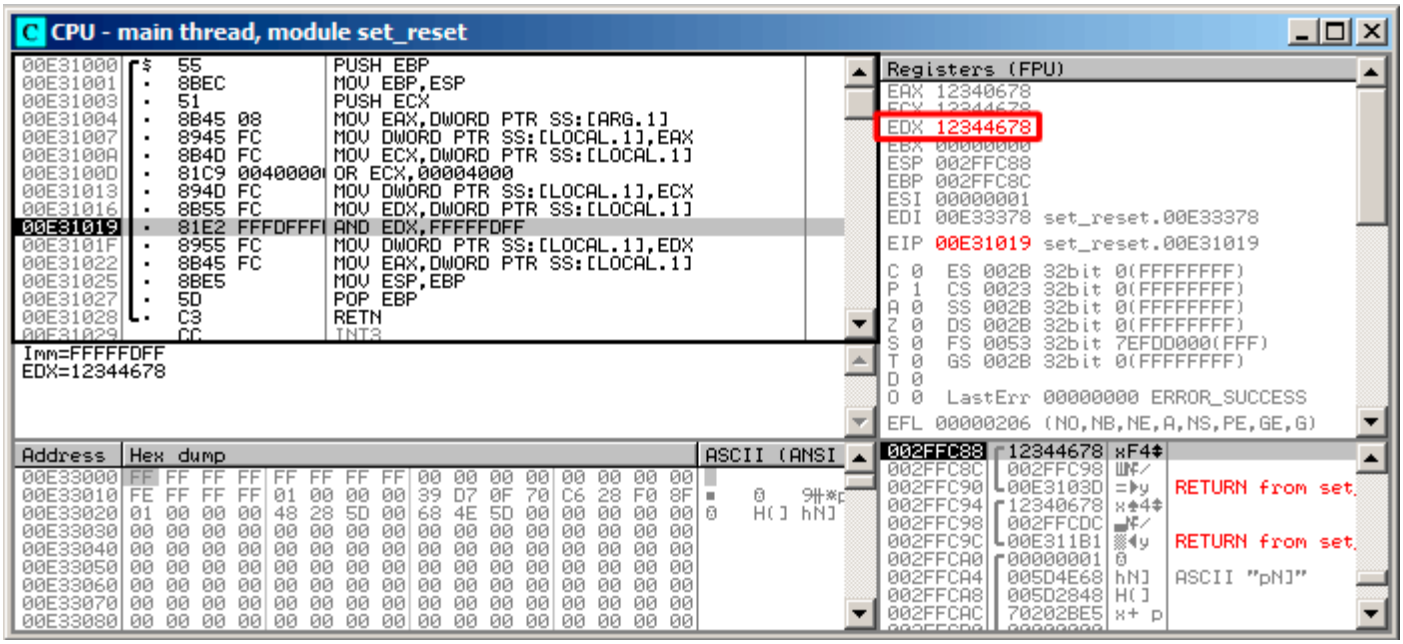


Fig. 1.96: OllyDbg : valeur rechargée dans EDX

AND exécuté:

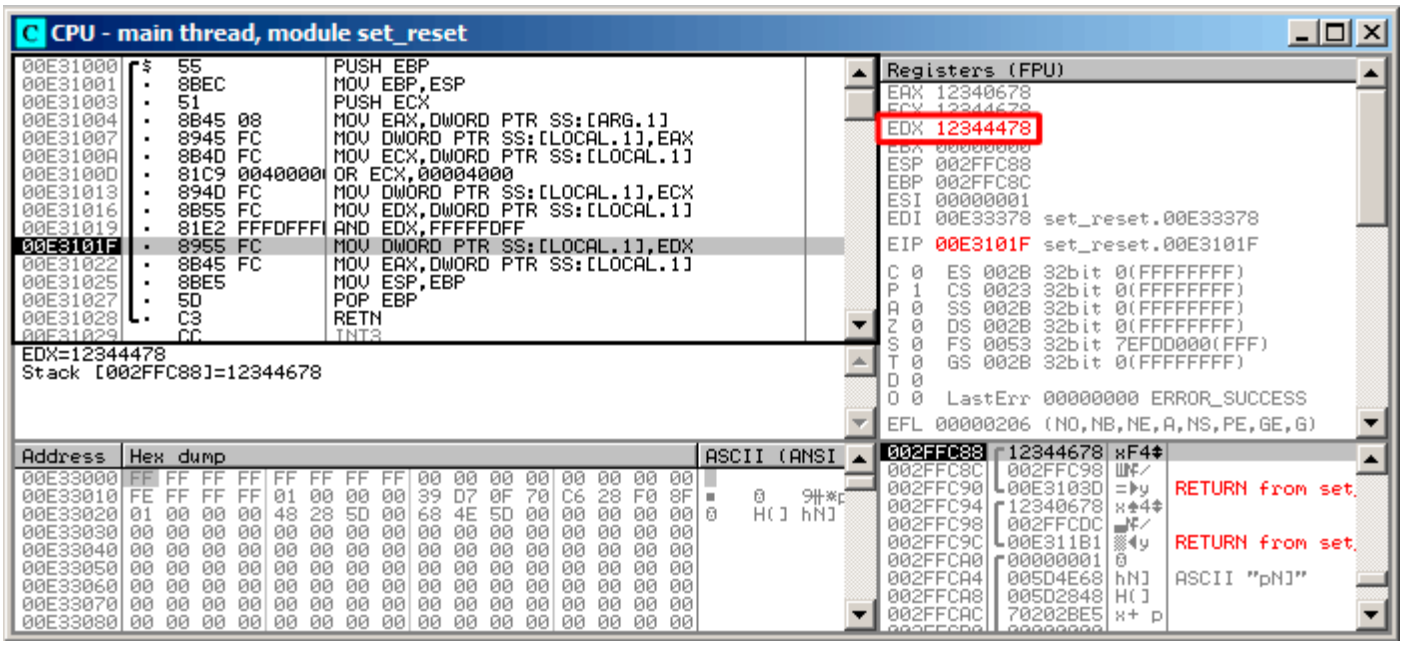


Fig. 1.97: OllyDbg : AND exécuté

Le 10ème bit a été mis à 0 (ou, en d'autres mots, tous les bits ont été laissés sauf le 10ème) et la valeur finale est maintenant 0x12344478 (0b10010001101000100010001111000).

MSVC avec optimisation

Si nous le compilons dans MSVC avec l'option d'optimisation (/0x), le code est même plus court:

Listing 1.270: MSVC avec optimisation

```
_a$ = 8                         ; size = 4
_f    PROC
    mov     eax, DWORD PTR _a$[esp-4]
    and     eax, -513           ; fffffdffH
    or      eax, 16384          ; 00004000H
    ret     0
_f    ENDP
```

GCC sans optimisation

Essayons avec GCC 4.4.1 sans optimisation:

Listing 1.271: GCC sans optimisation

```
                public f
f               proc near

var_4           = dword ptr -4
arg_0           = dword ptr  8

                push    ebp
                mov     ebp, esp
                sub     esp, 10h
                mov     eax, [ebp+arg_0]
                mov     [ebp+var_4], eax
                or      [ebp+var_4], 4000h
                and     [ebp+var_4], 0FFFFFDFFh
                mov     eax, [ebp+var_4]
                leave
```

```
                retn
f               endp
```

Il y a du code redondant, toutefois, c'est plus court que la version MSVC sans optimisation.

Maintenant, essayons GCC avec l'option d'optimisation `-03` :

GCC avec optimisation

Listing 1.272: GCC avec optimisation

```
                public f
f               proc near

arg_0           = dword ptr  8

                push    ebp
                mov     ebp, esp
                mov     eax, [ebp+arg_0]
                pop     ebp
                or      ah, 40h
                and     ah, 0FDh
                retn
f               endp
```

C'est plus court. Il est intéressant de noter que le compilateur travaille avec une partie du registre EAX via le registre AH—qui est la partie du registre EAX située entre les 8ème et 15ème bits inclus.

Octet d'indice							
7	6	5	4	3	2	1	0
RAX^{x64}							
				EAX			
						AX	
						AH	AL

N.B. L'accumulateur du CPU 16-bit 8086 était appelé AX et consistait en deux moitiés de 8-bit—AL (octet bas) et AH (octet haut). Dans le 80386, presque tous les registres ont été étendus à 32-bit, l'accumulateur a été appelé EAX, mais pour des raisons de compatibilité, ses *anciennes parties* peuvent toujours être accédées par AX/AH/AL.

Puisque tous les CPUs x86 sont des descendants du CPU 16-bit 8086, ces *anciens* opcodes 16-bit sont plus courts que les nouveaux sur 32-bit. C'est pourquoi l'instruction `or ah, 40h` occupe seulement 3 octets. Il serait plus logique de générer ici `or eax, 04000h` mais ça fait 5 octets, ou même 6 (dans le cas où le registre du premier opérande n'est pas EAX).

GCC avec optimisation et regparm

Il serait encore plus court en mettant le flag d'optimisation `-03` et aussi `regparm=3`.

Listing 1.273: GCC avec optimisation

```
                public f
f               proc near
                push    ebp
                or      ah, 40h
                mov     ebp, esp
                and     ah, 0FDh
                pop     ebp
                retn
f               endp
```

En effet, le premier argument est déjà chargé dans EAX, donc il est possible de travailler avec directement. Il est intéressant de noter qu'à la fois le prologue (`push ebp / mov ebp,esp`) et l'épilogue (`pop ebp`) de la fonction peuvent être facilement omis ici, mais sans doute que GCC n'est pas assez bon pour effectuer une telle optimisation de la taille du code. Toutefois, il est préférable que de telles petites fonctions soient des *fonctions inlined* (3.11 on page 511).

ARM + avec optimisation Keil 6/2013 (Mode ARM)

Listing 1.274: avec optimisation Keil 6/2013 (Mode ARM)

```
02 0C C0 E3          BIC     R0, R0, #0x200
01 09 80 E3          ORR     R0, R0, #0x4000
1E FF 2F E1          BX      LR
```

L'instruction BIC (*BItwise bit Clear*) est une instruction pour mettre à zéro des bits spécifiques. Ceci est comme l'instruction AND, mais avec un opérande inversé. I.e., c'est analogue à la paire d'instructions NOT +AND.

ORR est le « ou logique », analogue à OR en x86.

Jusqu'ici, c'est facile.

ARM + avec optimisation Keil 6/2013 (Mode Thumb)

Listing 1.275: avec optimisation Keil 6/2013 (Mode Thumb)

```
01 21 89 03          MOVS    R1, 0x4000
08 43                ORRS    R0, R1
49 11                ASRS    R1, R1, #5    ; génère 0x200 et le met dans R1
88 43                BICS    R0, R1
70 47                BX      LR
```

Il semble que Keil a décidé que le code en mode Thumb, pour générer 0x200 à partir de 0x4000, est plus compact que celui pour écrire 0x200 dans un registre arbitraire.

C'est pourquoi, avec l'aide de ASRS (décalage arithmétique vers la droite), cette valeur est calculée comme $0x4000 \gg 5$.

ARM + avec optimisation Xcode 4.6.3 (LLVM) (Mode ARM)

Listing 1.276: avec optimisation Xcode 4.6.3 (LLVM) (Mode ARM)

```
42 0C C0 E3          BIC             R0, R0, #0x4200
01 09 80 E3          ORR             R0, R0, #0x4000
1E FF 2F E1          BX              LR
```

Le code qui a été généré par LLVM, pourrait être quelque chose comme ça sous la forme de code source:

```
    REMOVE_BIT (rt, 0x4200);
    SET_BIT (rt, 0x4000);
```

Et c'est exactement ce dont nous avons besoin. Mais pourquoi 0x4200? Peut-être que c'est un artefact de l'optimiseur de LLVM[152].

Probablement une erreur de l'optimiseur du compilateur, mais le code généré fonctionne malgré tout correctement.

Vous pouvez en savoir plus sur les anomalies de compilateur ici (7.4 on page 641).

avec optimisation Xcode 4.6.3 (LLVM) pour le mode Thumb génère le même code.

ARM: plus d'informations sur l'instruction BIC

Retravaillons légèrement l'exemple:

```
int f(int a)
{
    int rt=a;

    REMOVE_BIT (rt, 0x1234);
```

[152] C'était LLVM build 2410.2.00 fourni avec Apple Xcode 4.6.3

```
    return rt;
};
```

Ensuite Keil 5.03 pour mode ARM avec optimisation fait:

```
f PROC
        BIC      r0,r0,#0x1000
        BIC      r0,r0,#0x234
        BX       lr
        ENDP
```

Il y a deux instructions BIC, i.e., les bits 0x1234 sont mis à zéro en deux temps.

C'est parce qu'il n'est pas possible d'encoder 0x1234 dans une instruction BIC, mais il est possible d'encoder 0x1000 et 0x234.

ARM64: GCC (Linaro) 4.9 avec optimisation

GCC en compilant avec optimisation pour ARM64 peut utiliser l'instruction AND au lieu de BIC :

Listing 1.277: GCC (Linaro) 4.9 avec optimisation

```
f :
        and     w0, w0, -513    ; 0xFFFFFFFFFFFFFDFF
        orr     w0, w0, 16384   ; 0x4000
        ret
```

ARM64: GCC (Linaro) 4.9 sans optimisation

GCC sans optimisation génère plus de code redondant, mais fonctionne comme celui optimisé:

Listing 1.278: GCC (Linaro) 4.9 sans optimisation

```
f :
        sub     sp, sp, #32
        str     w0, [sp,12]
        ldr     w0, [sp,12]
        str     w0, [sp,28]
        ldr     w0, [sp,28]
        orr     w0, w0, 16384   ; 0x4000
        str     w0, [sp,28]
        ldr     w0, [sp,28]
        and     w0, w0, -513    ; 0xFFFFFFFFFFFFFDFF
        str     w0, [sp,28]
        ldr     w0, [sp,28]
        add     sp, sp, 32
        ret
```

MIPS

Listing 1.279: GCC 4.4.5 avec optimisation (IDA)

```
f :
; $a0=a
                ori     $a0, 0x4000
; $a0=a|0x4000
                li      $v0, 0xFFFFFDFF
                jr      $ra
                and     $v0, $a0, $v0
; à la fin : $v0 = $a0&$v0 = a|0x4000 & 0xFFFFFDFF
```

ORI est, bien sûr, l'opération OR. « I » dans l'instruction signifie que la valeur est intégrée dans le code machine.

Mais après ça, nous avons AND. Il n'y a pas moyen d'utiliser ANDI car il n'est pas possible d'intégrer le nombre 0xFFFFFDFF dans une seule instruction, donc le compilateur doit d'abord charger 0xFFFFFDFF dans le registre $V0 et ensuite génère AND qui prend toutes ses valeurs depuis des registres.

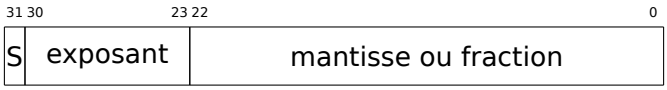
1.22.3 Décalages

Les décalages de bit sont implémentés en C/C++ avec les opérateurs ≪ et ≫. Le x86 ISA possède les instructions SHL (SHift Left / décalage à gauche) et SHR (SHift Right / décalage à droite) pour ceci. Les instructions de décalage sont souvent utilisées pour la division et la multiplication par des puissances de deux: 2^n (e.g., 1, 2, 4, 8, etc.): 1.18.1 on page 213, 1.18.2 on page 217.

Les opérations de décalage sont aussi si importantes car elles sont souvent utilisées pour isoler des bits spécifiques ou pour construire une valeur à partir de plusieurs bits épars.

1.22.4 Mettre et effacer des bits spécifiques: exemple avec le FPU

Voici comment les bits sont organisés dans le type *float* au format IEEE 754:



(Ssigne)

Le signe du nombre est dans le MSB[153]. Est-ce qu'il est possible de changer le signe d'un nombre en virgule flottante sans aucune instruction FPU?

```
#include <stdio.h>

float my_abs (float i)
{
        unsigned int tmp=(*(unsigned int*)&i) & 0x7FFFFFFF;
        return *(float*)&tmp;
};

float set_sign (float i)
{
        unsigned int tmp=(*(unsigned int*)&i) | 0x80000000;
        return *(float*)&tmp;
};

float negate (float i)
{
        unsigned int tmp=(*(unsigned int*)&i) ^ 0x80000000;
        return *(float*)&tmp;
};

int main()
{
        printf ("my_abs() :\n");
        printf ("%f\n", my_abs (123.456));
        printf ("%f\n", my_abs (-456.123));
        printf ("set_sign() :\n");
        printf ("%f\n", set_sign (123.456));
        printf ("%f\n", set_sign (-456.123));
        printf ("negate() :\n");
        printf ("%f\n", negate (123.456));
        printf ("%f\n", negate (-456.123));
};
```

Nous avons besoin de cette ruse en C/C++ pour copier vers/depuis des valeurs *float* sans conversion effective. Donc il y a trois fonctions: my_abs() supprime MSB; `set_sign()` met MSB et negate() l'inverse.

X0R peut être utilisé pour inverser un bit: 2.6 on page 465.

x86

Le code est assez simple.

[153] Bit le plus significatif

Listing 1.280: MSVC 2012 avec optimisation

```
_tmp$ = 8
_i$ = 8
_my_abs PROC
        and     DWORD PTR _i$[esp-4], 2147483647 ; 7fffffffH
        fld     DWORD PTR _tmp$[esp-4]
        ret     0
_my_abs ENDP

_tmp$ = 8
_i$ = 8
_set_sign PROC
        or      DWORD PTR _i$[esp-4], -2147483648 ; 80000000H
        fld     DWORD PTR _tmp$[esp-4]
        ret     0
_set_sign ENDP

_tmp$ = 8
_i$ = 8
_negate PROC
        xor     DWORD PTR _i$[esp-4], -2147483648 ; 80000000H
        fld     DWORD PTR _tmp$[esp-4]
        ret     0
_negate ENDP
```

Une valeur en entrée de type *float* est prise sur la pile, mais traitée comme une valeur entière.

AND et OR supprime et mette le bit désiré. XOR l'inverse.

Enfin, la valeur modifiée est chargée dans ST0,car les nombres en virgule flottante sont renvoyés dans ce registre.

Maintenant essayons l'optimisation de MSVC 2012 pour x64:

Listing 1.281: MSVC 2012 x64 avec optimisation

```
tmp$ = 8
i$ = 8
my_abs  PROC
        movss   DWORD PTR [rsp+8], xmm0
        mov     eax, DWORD PTR i$[rsp]
        btr     eax, 31
        mov     DWORD PTR tmp$[rsp], eax
        movss   xmm0, DWORD PTR tmp$[rsp]
        ret     0
my_abs  ENDP
_TEXT   ENDS

tmp$ = 8
i$ = 8
set_sign PROC
        movss   DWORD PTR [rsp+8], xmm0
        mov     eax, DWORD PTR i$[rsp]
        bts     eax, 31
        mov     DWORD PTR tmp$[rsp], eax
        movss   xmm0, DWORD PTR tmp$[rsp]
        ret     0
set_sign ENDP

tmp$ = 8
i$ = 8
negate  PROC
        movss   DWORD PTR [rsp+8], xmm0
        mov     eax, DWORD PTR i$[rsp]
        btc     eax, 31
        mov     DWORD PTR tmp$[rsp], eax
        movss   xmm0, DWORD PTR tmp$[rsp]
        ret     0
negate  ENDP
```

La valeur en entrée est passée dans XMM0, puis elle est copiée sur la pile locale et nous voyons des nouvelles instructions: BTR, BTS, BTC.

Ces instructions sont utilisées pour effacer (BTR), mettre (BTS) et inverser (ou faire le complément: BTC) de bits spécifiques. Le bit d'index 31 est le MSB, en comptant depuis 0.

Enfin, le résultat est copié dans XMM0, car les valeurs en virgule flottante sont renvoyées dans XMM0 en environnement Win64.

MIPS

GCC 4.4.5 pour MIPS fait essentiellement la même chose:

Listing 1.282: GCC 4.4.5 avec optimisation (IDA)

```
my_abs :
; déplacer depuis le coprocesseur 1:
                mfc1    $v1, $f12
                li      $v0, 0x7FFFFFFF
; $v0=0x7FFFFFFF
; faire AND :
                and     $v0, $v1
; déplacer vers le  coprocesseur 1:
                mtc1    $v0, $f0
; return
                jr      $ra
                or      $at, $zero ; slot de délai de branchement

set_sign :
; déplacer depuis le  coprocesseur 1:
                mfc1    $v0, $f12
                lui     $v1, 0x8000
; $v1=0x80000000
; faire OR :
                or      $v0, $v1, $v0
; déplacer vers le  coprocesseur 1:
                mtc1    $v0, $f0
; return
                jr      $ra
                or      $at, $zero ; slot de délai de branchement

negate :
; déplacer depuis le  coprocesseur 1:
                mfc1    $v0, $f12
                lui     $v1, 0x8000
; $v1=0x80000000
; do XOR :
                xor     $v0, $v1, $v0
; déplacer vers le  coprocesseur 1:
                mtc1    $v0, $f0
; sortir
                jr      $ra
                or      $at, $zero ; slot de délai de branchement
```

Une seule instruction LUI est utilisée pour charger 0x80000000 dans un registre, car LUI efface les 16 bits bas et ils sont à zéro dans la constante, donc un LUI sans ORI ultérieur est suffisant.

ARM

avec optimisation Keil 6/2013 (Mode ARM)

Listing 1.283: avec optimisation Keil 6/2013 (Mode ARM)

```
my_abs PROC
; effacer bit :
        BIC      r0,r0,#0x80000000
        BX       lr
```

```
        ENDP

set_sign PROC
; faire OR :
        ORR      r0,r0,#0x80000000
        BX       lr
        ENDP

negate PROC
; faire XOR :
        EOR      r0,r0,#0x80000000
        BX       lr
        ENDP
```

Jusqu'ici tout va bien.

ARM a l'instruction BIC, qui efface explicitement un (des) bit(s) spécifique(s). EOR est le nom de l'instruction ARM pour XOR (« Exclusive OR / OU exclusif »).

avec optimisation Keil 6/2013 (Mode Thumb)

Listing 1.284: avec optimisation Keil 6/2013 (Mode Thumb)

```
my_abs PROC
        LSLS     r0,r0,#1
; r0=i<<1
        LSRS     r0,r0,#1
; r0=(i<<1)>>1
        BX       lr
        ENDP

set_sign PROC
        MOVS     r1,#1
; r1=1
        LSLS     r1,r1,#31
; r1=1<<31=0x80000000
        ORRS     r0,r0,r1
; r0=r0 | 0x80000000
        BX       lr
        ENDP

negate PROC
        MOVS     r1,#1
; r1=1
        LSLS     r1,r1,#31
; r1=1<<31=0x80000000
        EORS     r0,r0,r1
; r0=r0 ^ 0x80000000
        BX       lr
        ENDP
```

En ARM, le mode Thumb offre des instructions 16-bit et peu de données peuvent y être encodées, donc ici une paire d'instructions MOVS/LSLS est utilisée pour former la constante 0x80000000. Ça fonctionne comme ceci: $1 << 31 = 0x80000000$.

Le code de my_abs est bizarre et fonctionne pratiquement comme cette expression: $(i << 1) >> 1$. Cette déclaration semble vide de sens. Mais néanmoins, lorsque $input << 1$ est exécuté, le MSB (bit de signe) est simplement supprimé. Puis lorsque la déclaration suivante $result >> 1$ est exécutée, tous les bits sont à nouveau à leur place, mais le MSB vaut zéro, car tous les « nouveaux » bits apparaissant lors d'une opération de décalage sont toujours zéro. C'est ainsi que la paire d'instructions LSLS/LSRS efface le MSB.

avec optimisation GCC 4.6.3 (Raspberry Pi, Mode ARM)

Listing 1.285: avec optimisation GCC 4.6.3 for Raspberry Pi (Mode ARM)

```
my_abs
```

```
; copier depuis S0 vers R2 :
                FMRS    R2, S0
; effacer le bit :
                BIC     R3, R2, #0x80000000
; copier depuis R3 vers S0 :
                FMSR    S0, R3
                BX      LR

set_sign
; copier depuis S0 vers R2 :
                FMRS    R2, S0
; faire OR :
                ORR     R3, R2, #0x80000000
; copier depuis R3 vers S0 :
                FMSR    S0, R3
                BX      LR

negate
; copier depuis S0 vers R2 :
                FMRS    R2, S0
; faire ADD :
                ADD     R3, R2, #0x80000000
; copier depuis R3 vers S0 :
                FMSR    S0, R3
                BX      LR
```

Lançons Linux pour Raspberry Pi dans QEMU et ça émule un FPU ARM, dons les S-registres sont utilisés pour les nombres en virgule flottante au lieu des R-registres.

L'instruction FMRS copie des données des GPR vers le FPU et retour.

`my_abs()` et `set_sign()` ressemblent a ce que l'on attend, mais negate()? Pourquoi est-ce qu'il y a ADD au lieu de XOR?

C'est dur à croire, mais l'instruction `ADD register, 0x80000000` fonctionne tout comme `XOR register, 0x80000000`. Tout d'abord, quel est notre but? Le but est de changer le MSB, donc oublions l'opération XOR. Des mathématiques niveau scolaire, nous nous rappelons qu'ajouter une valeur comme 1000 à une autre valeur n'affecte jamais les 3 derniers chiffres. Par exemple: $1234567 + 10000 = 1244567$ (les 4 derniers chiffres ne sont jamais affectés).

Mais ici nous opérons en base décimale et
0x80000000 est 0b10000000000000000000000000000000, i.e., seulement le bit le plus haut est mis.

Ajouter 0x80000000 à n'importe quelle valeur n'affecte jamais les 31 bits les plus bas, mais affecte seulement le MSB. Ajouter 1 à 0 donne 1.

Ajouter 1 à 1 donne 0b10 au format binaire, mais le bit d'indice 32 (en comptant à partir de zéro) est abandonné, car notre registre est large de 32 bit, donc le résultat est 0. C'est pourquoi XOR peut être remplacé par ADD ici.

Il est difficile de dire pourquoi GCC a décidé de faire ça, mais ça fonctionne correctement.

1.22.5 Compter les bits mis à 1

Voici un exemple simple d'une fonction qui compte le nombre de bits mis à 1 dans la valeur en entrée.

Cette opération est aussi appelée « population count »[154].

```
#include <stdio.h>

#define IS_SET(flag, bit)       ((flag) & (bit))

int f(unsigned int a)
{
    int i;
    int rt=0;

    for (i=0; i<32; i++)
```

[154] les CPUs x86 modernes (qui supportent SSE4) ont même une instruction POPCNT pour cela

```
        if (IS_SET (a, 1<<i))
            rt++;

    return rt;
};

int main()
{
    f(0x12345678); // test
};
```

Dans cette boucle, la variable d'itération i prend les valeurs de 0 à 31, donc la déclaration $1 \ll i$ prend les valeurs de 1 à 0x80000000. Pour décrire cette opération en langage naturel, nous dirions *décaler 1 par n bits à gauche*. En d'autres mots, la déclaration $1 \ll i$ produit consécutivement toutes les positions possible pour un bit dans un nombre de 32-bit. Le bit libéré à droite est toujours à 0.

Voici une table de tous les $1 \ll i$ possible for $i = 0 \dots 31$:

C/C++ expression	Puissance de deux	Forme décimale	Forme hexadécimale
$1 \ll 0$	1	1	1
$1 \ll 1$	2^1	2	2
$1 \ll 2$	2^2	4	4
$1 \ll 3$	2^3	8	8
$1 \ll 4$	2^4	16	0x10
$1 \ll 5$	2^5	32	0x20
$1 \ll 6$	2^6	64	0x40
$1 \ll 7$	2^7	128	0x80
$1 \ll 8$	2^8	256	0x100
$1 \ll 9$	2^9	512	0x200
$1 \ll 10$	2^{10}	1024	0x400
$1 \ll 11$	2^{11}	2048	0x800
$1 \ll 12$	2^{12}	4096	0x1000
$1 \ll 13$	2^{13}	8192	0x2000
$1 \ll 14$	2^{14}	16384	0x4000
$1 \ll 15$	2^{15}	32768	0x8000
$1 \ll 16$	2^{16}	65536	0x10000
$1 \ll 17$	2^{17}	131072	0x20000
$1 \ll 18$	2^{18}	262144	0x40000
$1 \ll 19$	2^{19}	524288	0x80000
$1 \ll 20$	2^{20}	1048576	0x100000
$1 \ll 21$	2^{21}	2097152	0x200000
$1 \ll 22$	2^{22}	4194304	0x400000
$1 \ll 23$	2^{23}	8388608	0x800000
$1 \ll 24$	2^{24}	16777216	0x1000000
$1 \ll 25$	2^{25}	33554432	0x2000000
$1 \ll 26$	2^{26}	67108864	0x4000000
$1 \ll 27$	2^{27}	134217728	0x8000000
$1 \ll 28$	2^{28}	268435456	0x10000000
$1 \ll 29$	2^{29}	536870912	0x20000000
$1 \ll 30$	2^{30}	1073741824	0x40000000
$1 \ll 31$	2^{31}	2147483648	0x80000000

Ces constantes (masques de bit) apparaissent très souvent le code et un rétro-ingénieur pratiquant doit pouvoir les repérer rapidement.

Les nombres décimaux avant 65536 et les hexadécimaux sont faciles à mémoriser. Tandis que les nombres décimaux après 65536 ne valent probablement pas la peine de l'être.

Ces constantes sont utilisées très souvent pour mapper des flags sur des bits spécifiques. Par exemple, voici un extrait de `ssl_private.h` du code source d'Apache 2.4.6:

```
/**
 * Define the SSL options
 */
#define SSL_OPT_NONE           (0)
#define SSL_OPT_RELSET         (1<<0)
#define SSL_OPT_STDENVVARS     (1<<1)
#define SSL_OPT_EXPORTCERTDATA (1<<3)
#define SSL_OPT_FAKEBASICAUTH  (1<<4)
#define SSL_OPT_STRICTREQUIRE  (1<<5)
```

```
#define SSL_OPT_OPTRENEGOTIATE (1<<6)
#define SSL_OPT_LEGACYDNFORMAT (1<<7)
```

Revenons à notre exemple.

La macro IS_SET teste la présence d'un bit dans a.

La macro IS_SET est en fait l'opération logique AND (*AND*) et elle renvoie 0 si le bit testé est absent (à 0), ou le masque de bit, si le bit est présent (à 1). L'opérateur *if()* en C/C++ exécute son code si l'expression n'est pas zéro, cela peut même être 123456, c'est pourquoi il fonctionne toujours correctement.

x86

MSVC

Compilons-le (MSVC 2010):

Listing 1.286: MSVC 2010

```
_rt$ = -8               ; taille = 4
_i$ = -4                ; taille = 4
_a$ = 8                 ; taille = 4
_f  PROC
    push   ebp
    mov    ebp, esp
    sub    esp, 8
    mov    DWORD PTR _rt$[ebp], 0
    mov    DWORD PTR _i$[ebp], 0
    jmp    SHORT $LN4@f
$LN3@f :
    mov    eax, DWORD PTR _i$[ebp]   ; incrémenter i
    add    eax, 1
    mov    DWORD PTR _i$[ebp], eax
$LN4@f :
    cmp    DWORD PTR _i$[ebp], 32    ; 00000020H
    jge    SHORT $LN2@f              ; boucle terminée?
    mov    edx, 1
    mov    ecx, DWORD PTR _i$[ebp]
    shl    edx, cl                   ; EDX=EDX<<CL
    and    edx, DWORD PTR _a$[ebp]
    je     SHORT $LN1@f              ; résultat de l'instruction AND égal à 0?
                                     ; alors passer les instructions suivantes
    mov    eax, DWORD PTR _rt$[ebp]  ; non, différent de zéro
    add    eax, 1                    ; incrémenter rt
    mov    DWORD PTR _rt$[ebp], eax
$LN1@f :
    jmp    SHORT $LN3@f
$LN2@f :
    mov    eax, DWORD PTR _rt$[ebp]
    mov    esp, ebp
    pop    ebp
    ret    0
_f    ENDP
```

OllyDbg

Chargeons cet exemple dans OllyDbg. Définissons la valeur d'entrée à 0x12345678.

Pour $i = 1$, nous voyons comment i est chargé dans ECX :

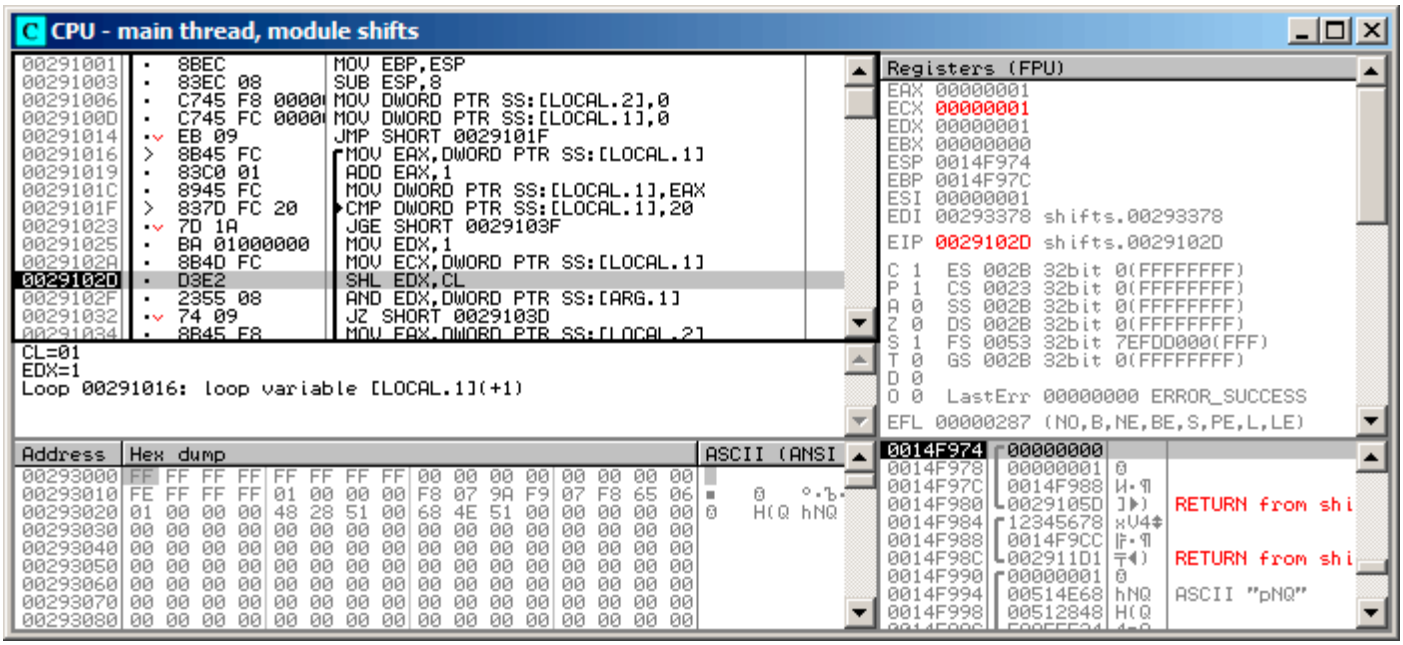


Fig. 1.98: OllyDbg : $i = 1$, i est chargé dans ECX

EDX contient 1. SHL va être exécuté maintenant.

SHL a été exécuté:

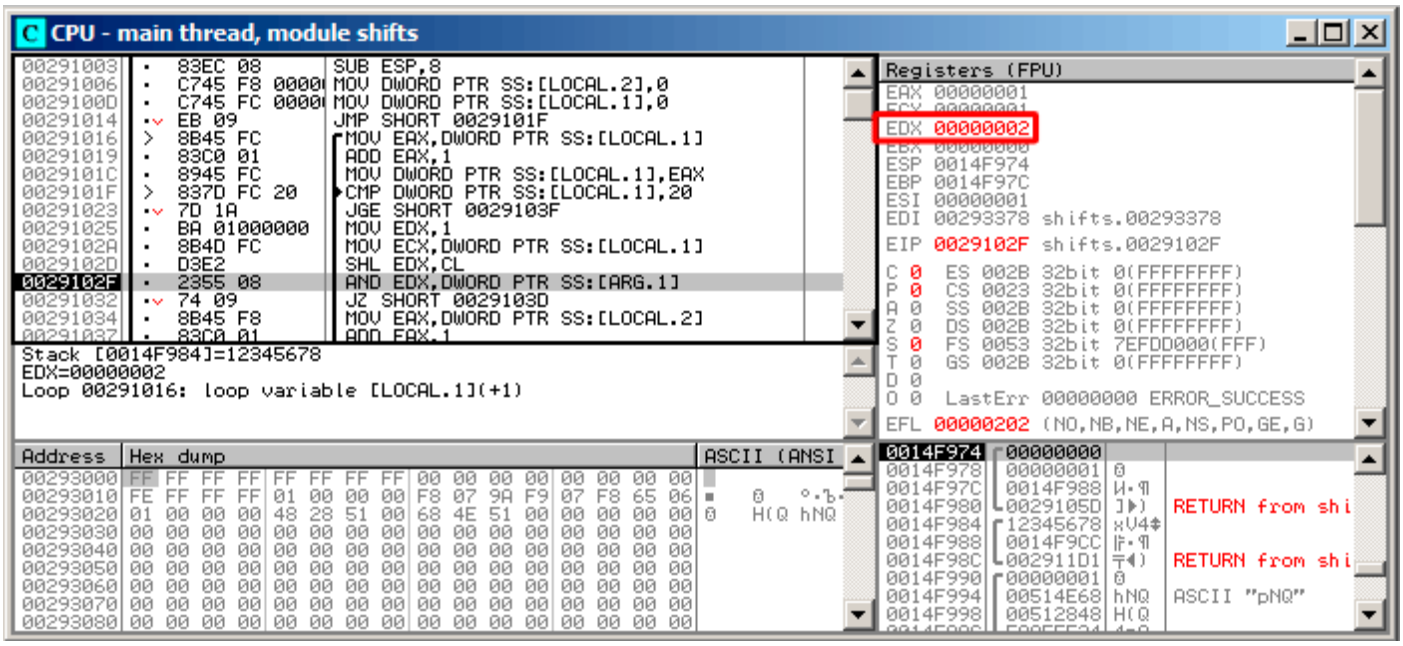


Fig. 1.99: OllyDbg : $i = 1$, EDX =$1 \ll 1 = 2$

EDX contient $1 \ll 1$ (ou 2). Ceci est un masque de bit.

AND met ZF à 1, ce qui implique que la valeur en entrée (0x12345678) ANDée avec 2 donne 0:

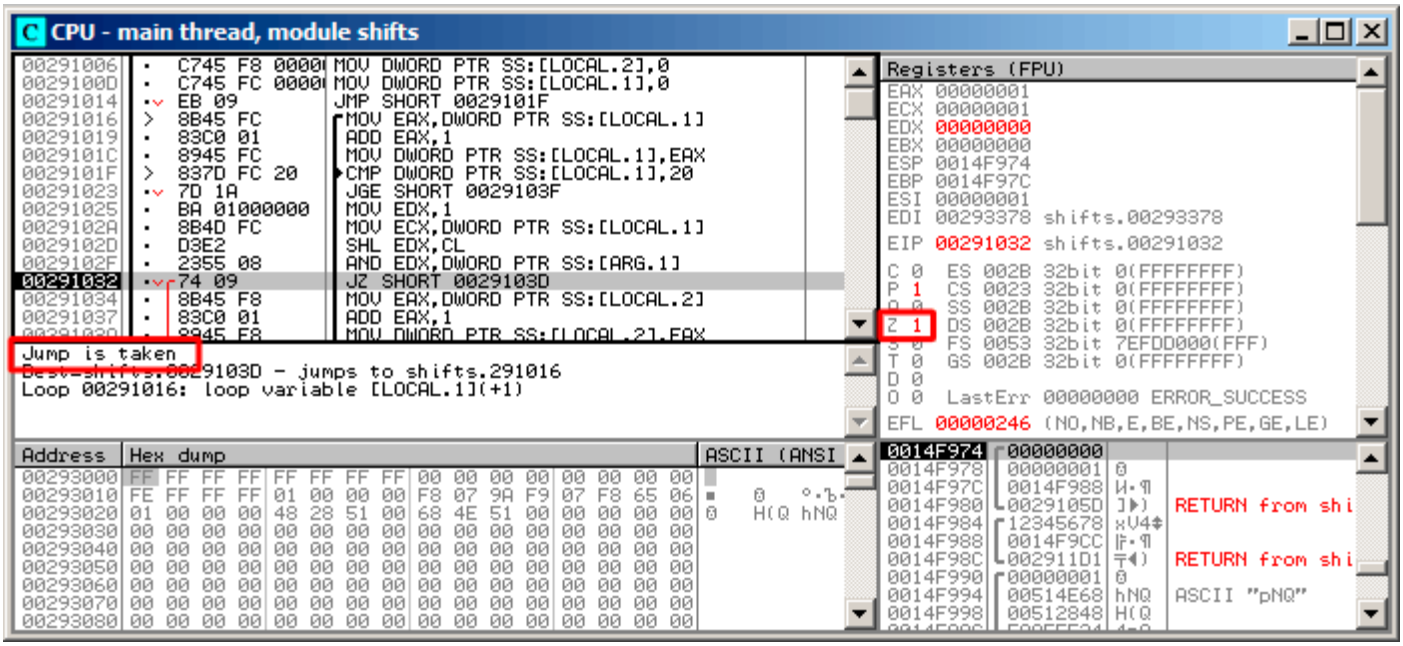


Fig. 1.100: OllyDbg : $i = 1$, y a-t-il ce bit dans la valeur en entrée? Non. (ZF =1)

Donc, il n'y a pas le bit correspondant dans la valeur en entrée.

Le morceau de code, qui incrémente le compteur ne va pas être exécuté: l'instruction JZ l'évite.

Avançons un peu plus et i vaut maintenant 4. SHL va être exécuté maintenant:

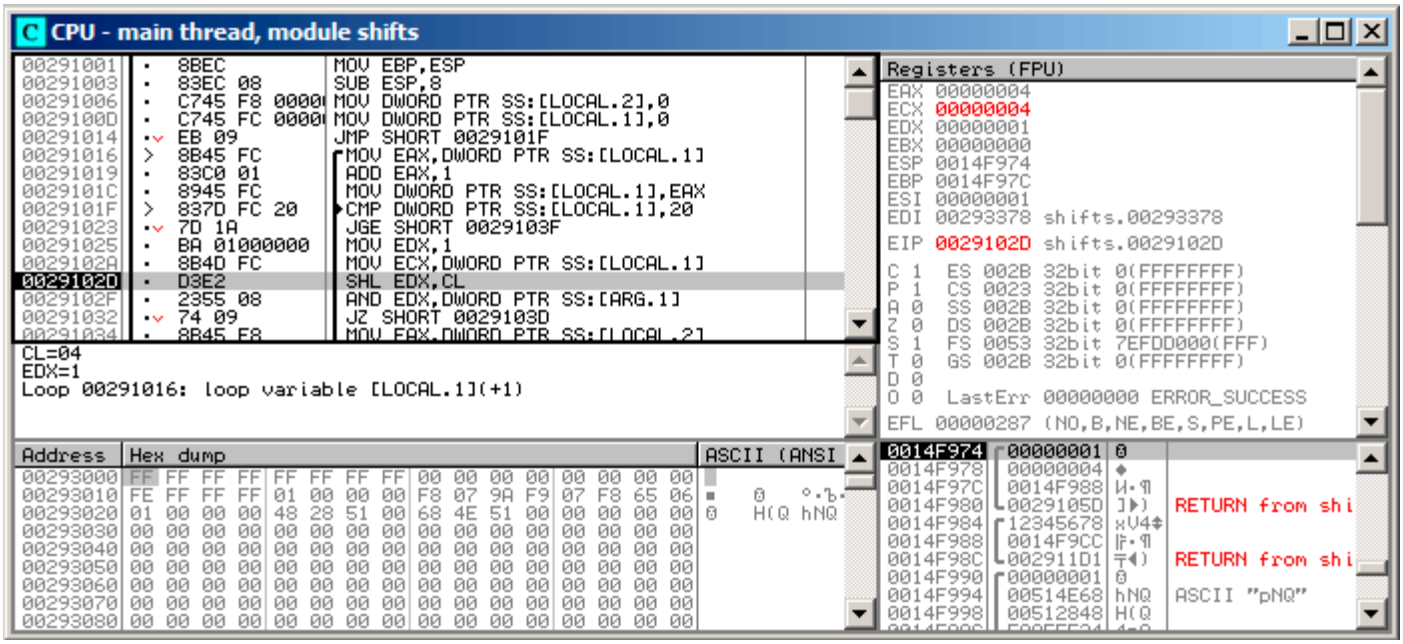


Fig. 1.101: OllyDbg : $i = 4$, i est chargée dans ECX

EDX =$1 \ll 4$ (ou 0x10 ou 16):

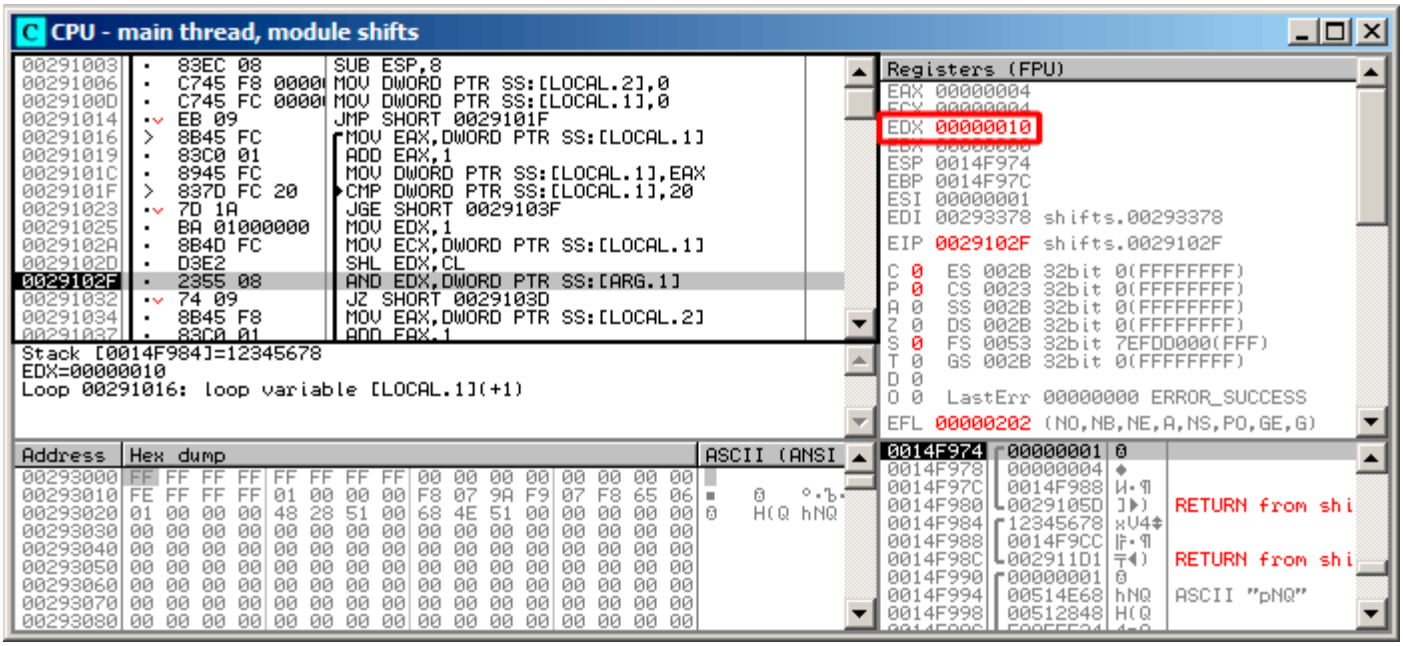


Fig. 1.102: OllyDbg : $i = 4$, EDX =$1 \ll 4 = 0x10$

Ceci est un autre masque de bit.

AND est exécuté:

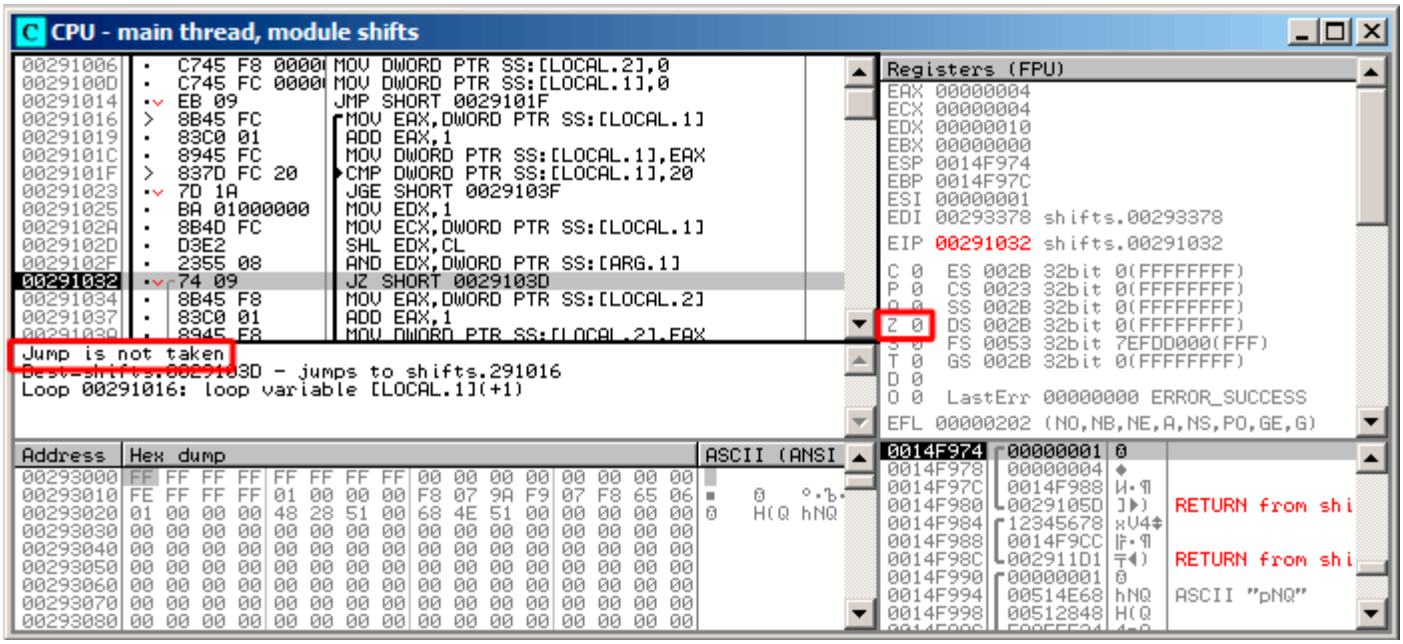


Fig. 1.103: OllyDbg : $i = 4$, y a-t-il ce bit dans la valeur en entrée? Oui. (ZF =0)

ZF est à 0 car ce bit est présent dans la valeur en entrée.
En effet, `0x12345678 & 0x10 = 0x10`.

Ce bit compte: le saut n'est pas effectué et le compteur de bit est incrémenté.

La fonction renvoie 13. C'est le nombre total de bits à 1 dans `0x12345678`.

GCC

Compilons-le avec GCC 4.4.1:

Listing 1.287: GCC 4.4.1

```
                public f
f               proc near

rt              = dword ptr -0Ch
i               = dword ptr -8
arg_0           = dword ptr  8

                push    ebp
                mov     ebp, esp
                push    ebx
                sub     esp, 10h
                mov     [ebp+rt], 0
                mov     [ebp+i], 0
                jmp     short loc_80483EF
loc_80483D0 :
                mov     eax, [ebp+i]
                mov     edx, 1
                mov     ebx, edx
                mov     ecx, eax
                shl     ebx, cl
                mov     eax, ebx
                and     eax, [ebp+arg_0]
                test    eax, eax
                jz      short loc_80483EB
                add     [ebp+rt], 1
loc_80483EB :
                add     [ebp+i], 1
loc_80483EF :
```

```
                cmp     [ebp+i], 1Fh
                jle     short loc_80483D0
                mov     eax, [ebp+rt]
                add     esp, 10h
                pop     ebx
                pop     ebp
                retn
f               endp
```

x64

Modifions légèrement l'exemple pour l'étendre à 64-bit:

```
#include <stdio.h>
#include <stdint.h>

#define IS_SET(flag, bit)       ((flag) & (bit))

int f(uint64_t a)
{
    uint64_t i;
    int rt=0;

    for (i=0; i<64; i++)
        if (IS_SET (a, 1ULL<<i))
            rt++;

    return rt;
};
```

GCC 4.8.2 sans optimisation

Jusqu'ici, c'est facile.

Listing 1.288: GCC 4.8.2 sans optimisation

```
f :
        push    rbp
        mov     rbp, rsp
        mov     QWORD PTR [rbp-24], rdi ; a
        mov     DWORD PTR [rbp-12], 0   ; rt=0
        mov     QWORD PTR [rbp-8], 0    ; i=0
        jmp     .L2
.L4 :
        mov     rax, QWORD PTR [rbp-8]
        mov     rdx, QWORD PTR [rbp-24]
; RAX = i, RDX = a
        mov     ecx, eax
; ECX = i
        shr     rdx, cl
; RDX = RDX>>CL = a>>i
        mov     rax, rdx
; RAX = RDX = a>>i
        and     eax, 1
; EAX = EAX&1 = (a>>i)&1
        test    rax, rax
; est-ce que le dernier bit est zéro?
; passer l'instruction ADD suivante, si c'est le cas.
        je      .L3
        add     DWORD PTR [rbp-12], 1   ; rt++
.L3 :
        add     QWORD PTR [rbp-8], 1    ; i++
.L2 :
        cmp     QWORD PTR [rbp-8], 63   ; i<63?
        jbe     .L4                     ; sauter au début du corps de la boucle, si oui
        mov     eax, DWORD PTR [rbp-12] ; renvoyer rt
```

```
        pop     rbp
        ret
```

GCC 4.8.2 avec optimisation

Listing 1.289: GCC 4.8.2 avec optimisation

```
1  f :
2          xor     eax, eax        ; la variable rt sera dans le registre EAX
3          xor     ecx, ecx        ; la variable i sera dans le registre ECX
4  .L3 :
5          mov     rsi, rdi        ; charger la valeur en entrée
6          lea     edx, [rax+1]    ; EDX=EAX+1
7  ; ici EDX est la nouvelle version de rt,
8  ; qui sera écrite dans la variable rt, si le dernier bit est à 1
9          shr     rsi, cl         ; RSI=RSI>>CL
10         and     esi, 1          ; ESI=ESI&1
11 ; est-ce que le dernier bit est 1? Si oui, écrire la nouvelle version de rt dans EAX
12         cmovne  eax, edx
13         add     rcx, 1          ; RCX++
14         cmp     rcx, 64
15         jne     .L3
16         rep ret                 ; AKA fatret
```

Ce code est plus concis, mais a une particularité.

Dans tous les exemples que nous avons vu jusqu'ici, nous incrémentions la valeur de « rt » après la comparaison d'un bit spécifique, mais le code ici incrémente « rt » avant (ligne 6), écrivant la nouvelle valeur dans le registre EDX. Donc, si le dernier bit est à 1, l'instruction CMOVNE[155] (qui est un synonyme pour CMOVNZ[156]) *commits* la nouvelle valeur de « rt » en déplaçant EDX (« valeur proposée de rt ») dans EAX (« rt courant » qui va être retourné à la fin).

C'est pourquoi l'incrémentation est effectuée à chaque étape de la boucle, i.e., 64 fois, sans relation avec la valeur en entrée.

L'avantage de ce code est qu'il contient seulement un saut conditionnel (à la fin de la boucle) au lieu de deux sauts (évitant l'incrément de la valeur de « rt » et à la fin de la boucle). Et cela doit s'exécuter plus vite sur les CPUs modernes avec des prédicteurs de branchement: 2.10.1 on page 470.

La dernière instruction est REP RET (opcode F3 C3) qui est aussi appelée FATRET par MSVC. C'est en quelque sorte une version optimisée de RET, qu'AMD recommande de mettre en fin de fonction, si RET se trouve juste après un saut conditionnel: [[*Software Optimization Guide for AMD Family 16h Processors*, (2013)]p.15] [157].

MSVC 2010 avec optimisation

Listing 1.290: MSVC 2010 avec optimisation

```
a$ = 8
f       PROC
; RCX = valeur en entrée
        xor     eax, eax
        mov     edx, 1
        lea     r8d, QWORD PTR [rax+64]
; R8D=64
        npad    5
$LL4@f :
        test    rdx, rcx
; il n'y a pas le même bit dans la valeur en entrée?
; alors passer la prochaine instruction INC.
        je      SHORT $LN3@f
        inc     eax     ; rt++
$LN3@f :
```

[155] Conditional MOVe if Not Equal
[156] Conditional MOVe if Not Zero
[157] Lire aussi à ce propos: http://go.yurichev.com/17328

```
        rol     rdx, 1   ; RDX=RDX<<1
        dec     r8       ; R8--
        jne     SHORT $LL4@f
        fatret  0
f       ENDP
```

Ici l'instruction ROL est utilisée au lieu de SHL, qui est en fait « rotate left / pivoter à gauche » au lieu de « shift left / décaler à gauche », mais dans cet exemple elle fonctionne tout comme SHL.

Vous pouvez en lire plus sur l'instruction de rotation ici: **??** on page ??.

R8 ici est compté de 64 à 0. C'est tout comme un i inversé.

Voici une table de quelques registres pendant l'exécution:

RDX	R8
0x0000000000000001	64
0x0000000000000002	63
0x0000000000000004	62
0x0000000000000008	61
...	...
0x4000000000000000	2
0x8000000000000000	1

À la fin, nous voyons l'instruction FATRET, qui a été expliquée ici: 1.22.5 on the previous page.

MSVC 2012 avec optimisation

Listing 1.291: MSVC 2012 avec optimisation

```
a$ = 8
f       PROC
; RCX = valeur en entrée
        xor     eax, eax
        mov     edx, 1
        lea     r8d, QWORD PTR [rax+32]
; EDX = 1, R8D = 32
        npad    5
$LL4@f :
; pass 1 ------------------------------
        test    rdx, rcx
        je      SHORT $LN3@f
        inc     eax      ; rt++
$LN3@f :
        rol     rdx, 1   ; RDX=RDX<<1
; -----------------------------------
; pass 2 ------------------------------
        test    rdx, rcx
        je      SHORT $LN11@f
        inc     eax      ; rt++
$LN11@f :
        rol     rdx, 1   ; RDX=RDX<<1
; -----------------------------------
        dec     r8       ; R8--
        jne     SHORT $LL4@f
        fatret  0
f       ENDP
```

MSVC 2012 avec optimisation fait presque le même job que MSVC 2010 avec optimisation, mais en quelque sorte, il génère deux corps de boucles identiques et le nombre de boucles est maintenant 32 au lieu de 64.

Pour être honnête, il n'est pas possible de dire pourquoi. Une ruse d'optimisation? Peut-être est-il meilleur pour le corps de la boucle d'être légèrement plus long?

De toute façon, ce genre de code est pertinent ici pour montrer que parfois la sortie du compilateur peut être vraiment bizarre et illogique, mais fonctionner parfaitement.

ARM + avec optimisation Xcode 4.6.3 (LLVM) (Mode ARM)

Listing 1.292: avec optimisation Xcode 4.6.3 (LLVM) (Mode ARM)

```
                MOV             R1, R0
                MOV             R0, #0
                MOV             R2, #1
                MOV             R3, R0
loc_2E54
                TST             R1, R2,LSL R3 ; mettre les flags suivant R1 & (R2<<R3)
                ADD             R3, R3, #1    ; R3++
                ADDNE           R0, R0, #1    ; si le flag ZF est mis par 0 TST, alors R0++
                CMP             R3, #32
                BNE             loc_2E54
                BX              LR
```

TST est la même chose que TEST en x86.

Comme noté précédemment (3.9.3 on page 504), il n'y a pas d'instruction de décalage séparée en mode ARM Toutefois, il y a ces modificateurs LSL (*Logical Shift Left / décalage logique à gauche*), LSR (*Logical Shift Right / décalage logique à droite*), ASR (*Arithmetic Shift Right décalage arithmétique à droite*), ROR (*Rotate Right / rotation à droite*) et RRX (*Rotate Right with Extend / rotation à droite avec extension*), qui peuvent être ajoutés à des instructions comme MOV, TST, CMP, ADD, SUB, RSB[158].

Ces modificateurs définissent comment décaler le second opérande et de combien de bits.

Ainsi l'instruction « TST R1, R2,LSL R3 » fonctionne ici comme $R1 \wedge (R2 \ll R3)$.

ARM + avec optimisation Xcode 4.6.3 (LLVM) (Mode Thumb-2)

Presque la même, mais ici il y a deux instructions utilisées, LSL.W/TST, au lieu d'une seule TST, car en mode Thumb il n'est pas possible de définir le modificateur LSL directement dans TST.

```
                MOV             R1, R0
                MOVS            R0, #0
                MOV.W           R9, #1
                MOVS            R3, #0
loc_2F7A
                LSL.W           R2, R9, R3
                TST             R2, R1
                ADD.W           R3, R3, #1
                IT NE
                ADDNE           R0, #1
                CMP             R3, #32
                BNE             loc_2F7A
                BX              LR
```

ARM64 + GCC 4.9 avec optimisation

Prenons un exemple en 64.bit qui a déjà été utilisé: 1.22.5 on page 332.

Listing 1.293: GCC (Linaro) 4.8 avec optimisation

```
f :
        mov     w2, 0           ; rt=0
        mov     x5, 1
        mov     w1, w2
.L2 :
        lsl     x4, x5, x1      ; w4 = w5<<w1 = 1<<i
        add     w3, w2, 1       ; new_rt=rt+1
        tst     x4, x0          ; (1<<i) & a
        add     w1, w1, 1       ; i++
; le résultat de TST était non-zéro?
; alors w2=w3 ou rt=new_rt.
; autrement : w2=w2 ou rt=rt (opération idle)
```

[158] Ces instructions sont également appelées « instructions de traitement de données »

```
        csel    w2, w3, w2, ne
        cmp     w1, 64          ; i<64?
        bne     .L2             ; oui
        mov     w0, w2          ; renvoyer rt
        ret
```

Le résultat est très semblable à ce que GCC génère pour x64: 1.289 on page 333.

L'instruction CSEL signifie « Conditional SELect / sélection conditionnelle ». Elle choisi une des deux variables en fonction des flags mis par TST et copie la valeur dans W2, qui contient la variable « rt ».

ARM64 + GCC 4.9 sans optimisation

De nouveau, nous travaillons sur un exemple 64-bit qui a déjà été utilisé: 1.22.5 on page 332. Le code est plus verbeux, comme d'habitude.

Listing 1.294: sans optimisation GCC (Linaro) 4.8

```
f :
        sub     sp, sp, #32
        str     x0, [sp,8]      ; stocker la valeur de "a" dans la zone de
                                ; sauvegarde des registres
        str     wzr, [sp,24]    ; rt=0
        str     wzr, [sp,28]    ; i=0
        b       .L2
.L4 :
        ldr     w0, [sp,28]
        mov     x1, 1
        lsl     x0, x1, x0      ; X0 = X1<<X0 = 1<<i
        mov     x1, x0
; X1 = 1<<i
        ldr     x0, [sp,8]
; X0 = a
        and     x0, x1, x0
; X0 = X1&X0 = (1<<i) & a
; X0 contient zéro? alors sauter en .L3, évitant d'incrémenter "rt"
        cmp     x0, xzr
        beq     .L3
; rt++
        ldr     w0, [sp,24]
        add     w0, w0, 1
        str     w0, [sp,24]
.L3 :
; i++
        ldr     w0, [sp,28]
        add     w0, w0, 1
        str     w0, [sp,28]
.L2 :
; i<=63? alors sauter en .L4
        ldr     w0, [sp,28]
        cmp     w0, 63
        ble     .L4
; renvoyer rt
        ldr     w0, [sp,24]
        add     sp, sp, 32
        ret
```

MIPS

GCC sans optimisation

Listing 1.295: GCC 4.4.5 sans optimisation (IDA)

```
f :
; IDA ne connaît pas le nom des variables, nous les donnons manuellement :
rt              = -0x10
```

```
i               = -0xC
var_4           = -4
a               =  0

                addiu   $sp, -0x18
                sw      $fp, 0x18+var_4($sp)
                move    $fp, $sp
                sw      $a0, 0x18+a($fp)
; initialiser les variables rt et i à zéro :
                sw      $zero, 0x18+rt($fp)
                sw      $zero, 0x18+i($fp)
; saut aux instructions de test de la boucle
                b       loc_68
                or      $at, $zero ; slot de délai de branchement, NOP

loc_20 :
                li      $v1, 1
                lw      $v0, 0x18+i($fp)
                or      $at, $zero ; slot de délai de chargement, NOP
                sllv    $v0, $v1, $v0
; $v0 = 1<<i
                move    $v1, $v0
                lw      $v0, 0x18+a($fp)
                or      $at, $zero ; slot de délai de chargement, NOP
                and     $v0, $v1, $v0
; $v0 = a&(1<<i)
; est-ce que a&(1<<i) est égal à zéro? sauter en loc_58 si oui :
                beqz    $v0, loc_58
                or      $at, $zero
; il n'y pas eu de saut, cela signifie que a&(1<<i)!=0, il faut donc incrémenter "rt":
                lw      $v0, 0x18+rt($fp)
                or      $at, $zero ; slot de délai de chargement, NOP
                addiu   $v0, 1
                sw      $v0, 0x18+rt($fp)

loc_58 :
; incrémenter i :
                lw      $v0, 0x18+i($fp)
                or      $at, $zero ; slot de délai de chargement, NOP
                addiu   $v0, 1
                sw      $v0, 0x18+i($fp)

loc_68 :
; charger i et le comparer avec 0x20 (32).
; sauter en loc_20 si il vaut moins de 0x20 (32) :
                lw      $v0, 0x18+i($fp)
                or      $at, $zero ; slot de délai de chargement, NOP
                slti    $v0, 0x20  # ' '
                bnez    $v0, loc_20
                or      $at, $zero ; slot de délai de branchement, NOP
;  épilogue de la fonction. renvoyer rt :
                lw      $v0, 0x18+rt($fp)
                move    $sp, $fp   ; slot de délai de chargement
                lw      $fp, 0x18+var_4($sp)
                addiu   $sp, 0x18  ; slot de délai de chargement
                jr      $ra
                or      $at, $zero ; slot de délai de branchement, NOP
```

C'est très verbeux: toutes les variables locales sont situées dans la pile locale et rechargées à chaque fois que l'on en a besoin.

L'instruction SLLV est « Shift Word Left Logical Variable », elle diffère de SLL seulement de ce que la valeur du décalage est encodée dans l'instruction SLL (et par conséquent fixée) mais SLLV lit cette valeur depuis un registre.

GCC avec optimisation

C'est plus concis. Il y a deux instructions de décalage au lieu d'une. Pourquoi?

Il est possible de remplacer la première instruction SLLV avec une instruction de branchement inconditionnel qui saute directement au second SLLV. Mais cela ferait une autre instruction de branchement dans la fonction, et il est toujours favorable de s'en passer: 2.10.1 on page 470.

Listing 1.296: GCC 4.4.5 avec optimisation (IDA)

```
f :
; $a0=a
; la variable rt sera dans $v0 :
                move    $v0, $zero
; la variable i sera dans $v1 :
                move    $v1, $zero
                li      $t0, 1
                li      $a3, 32
                sllv    $a1, $t0, $v1
; $a1 = $t0<<$v1 = 1<<i

loc_14 :
                and     $a1, $a0
; $a1 = a&(1<<i)
; incrémenter i :
                addiu   $v1, 1
; sauter en loc_28 si a&(1<<i)==0 et incrémenter rt :
                beqz    $a1, loc_28
                addiu   $a2, $v0, 1
; si le saut BEQZ n'a pas été suivi, sauver la nouvelle valeur de rt dans $v0 :
                move    $v0, $a2

loc_28 :
; si i!=32, sauter en loc_14 et préparer la prochaine valeur décalée :
                bne     $v1, $a3, loc_14
                sllv    $a1, $t0, $v1
; sortir
                jr      $ra
                or      $at, $zero ; slot de délai de branchement, NOP
```

1.22.6 Conclusion

Semblables aux opérateurs de décalage de C/C++ ≪ et ≫, les instructions de décalage en x86 sont SHR/SHL (pour les valeurs non-signées) et SAR/SHL (pour les valeurs signées).

Les instructions de décalages en ARM sont LSR/LSL (pour les valeurs non-signées) et ASR/LSL (pour les valeurs signées).

Il est aussi possible d'ajouter un suffixe de décalage à certaines instructions (qui sont appelées « data processing instructions/instructions de traitement de données »).

Tester un bit spécifique (connu à l'étape de compilation)

Tester si le bit 0b1000000 (0x40) est présent dans la valeur du registre:

Listing 1.297: C/C++

```
if (input&0x40)
    ...
```

Listing 1.298: x86

```
TEST REG, 40h
JNZ is_set
; le bit n'est pas mis (est à 0)
```

Listing 1.299: x86

```
TEST REG, 40h
JZ is_cleared
; le bit est mis (est à 1)
```

Listing 1.300: ARM (Mode ARM)

```
TST REG, #0x40
BNE is_set
; le bit n'est pas mis (est à 0)
```

Parfois, AND est utilisé au lieu de TEST, mais les flags qui sont mis sont les même.

Tester un bit spécifique (spécifié lors de l'exécution)

Ceci est effectué en général par ce bout de code C/C++ (décaler la valeur de n bits vers la droite, puis couper le plus petit bit):

Listing 1.301: C/C++

```
if ((value>>n)&1)
    ....
```

Ceci est en général implémenté en code x86 avec:

Listing 1.302: x86

```
; REG=input_value
; CL=n
SHR REG, CL
AND REG, 1
```

Ou (décaler 1 bit n fois à gauche, isoler ce bit dans la valeur entrée et tester si ce n'est pas zéro):

Listing 1.303: C/C++

```
if (value & (1<<n))
    ....
```

Ceci est en général implémenté en code x86 avec:

Listing 1.304: x86

```
; CL=n
MOV REG, 1
SHL REG, CL
AND input_value, REG
```

Mettre à 1 un bit spécifique (connu à l'étape de compilation)

Listing 1.305: C/C++

```
value=value|0x40;
```

Listing 1.306: x86

```
OR REG, 40h
```

Listing 1.307: ARM (Mode ARM) and ARM64

```
ORR R0, R0, #0x40
```

Mettre à 1 un bit spécifique (spécifié lors de l'exécution)

Listing 1.308: C/C++

```
value=value|(1<<n);
```

Ceci est en général implémenté en code x86 avec:

Listing 1.309: x86

```
; CL=n
MOV REG, 1
SHL REG, CL
OR input_value, REG
```

Mettre à 0 un bit spécifique (connu à l'étape de compilation)

Il suffit d'effectuer l'opération AND sur la valeur inversée:

Listing 1.310: C/C++

```
value=value&(~0x40);
```

Listing 1.311: x86

```
AND REG, 0FFFFFFBFh
```

Listing 1.312: x64

```
AND REG, 0FFFFFFFFFFFFFFBFh
```

Ceci laisse tous les bits qui sont à 1 inchangés excepté un.

ARM en mode ARM a l'instruction BIC, qui fonctionne comme la paire d'instructions: NOT +AND :

Listing 1.313: ARM (Mode ARM)

```
BIC R0, R0, #0x40
```

Mettre à 0 un bit spécifique (spécifié lors de l'exécution)

Listing 1.314: C/C++

```
value=value&(~(1<<n));
```

Listing 1.315: x86

```
; CL=n
MOV REG, 1
SHL REG, CL
NOT REG
AND input_value, REG
```

1.22.7 Exercices

- http://challenges.re/67
- http://challenges.re/68
- http://challenges.re/69
- http://challenges.re/70

1.23 Générateur congruentiel linéaire comme générateur de nombres pseudo-aléatoires

Peut-être que le générateur congruentiel linéaire est le moyen le plus simple possible de générer des nombres aléatoires.

Ce n'est plus très utilisé aujourd'hui[159], mais il est si simple (juste une multiplication, une addition et une opération AND) que nous pouvons l'utiliser comme un exemple.

[159] Le twister de Mersenne est meilleur.

```
#include <stdint.h>

// constantes du livre Numerical Recipes
#define RNG_a 1664525
#define RNG_c 1013904223

static uint32_t rand_state;

void my_srand (uint32_t init)
{
        rand_state=init;
}

int my_rand ()
{
        rand_state=rand_state*RNG_a;
        rand_state=rand_state+RNG_c;
        return rand_state & 0x7fff;
}
```

Il y a deux fonctions: la première est utilisée pour initialiser l'état interne, et la seconde est appelée pour générer un nombre pseudo-aléatoire.

Nous voyons que deux constantes sont utilisées dans l'algorithme. Elles proviennent de [William H. Press and Saul A. Teukolsky and William T. Vetterling and Brian P. Flannery, *Numerical Recipes*, (2007)].

Définissons-les en utilisant la déclaration C/C++ `#define`. C'est une macro.

La différence entre une macro C/C++ et une constante est que toutes les macros sont remplacées par leur valeur par le pré-processeur C/C++, et qu'elles n'utilisent pas de mémoire, contrairement aux variables.

Par contre, une constante est une variable en lecture seule.

Il est possible de prendre un pointeur (ou une adresse) d'une variable constante, mais c'est impossible de faire ça avec une macro.

La dernière opération AND est nécessaire car d'après le standard C `my_rand()` doit renvoyer une valeur dans l'intervalle 0..32767.

Si vous voulez obtenir des valeurs pseudo-aléatoires 32-bit, il suffit d'omettre la dernière opération AND.

1.23.1 x86

Listing 1.316: MSVC 2013 avec optimisation

```
_BSS    SEGMENT
_rand_state DD  01H DUP (?)
_BSS    ENDS

_init$ = 8
_srand  PROC
        mov     eax, DWORD PTR _init$[esp-4]
        mov     DWORD PTR _rand_state, eax
        ret     0
_srand  ENDP

_TEXT   SEGMENT
_rand   PROC
        imul    eax, DWORD PTR _rand_state, 1664525
        add     eax, 1013904223 ; 3c6ef35fH
        mov     DWORD PTR _rand_state, eax
        and     eax, 32767      ; 00007fffH
        ret     0
_rand   ENDP

_TEXT   ENDS
```

Nous les voyons ici: les deux constantes sont intégrées dans le code. Il n'y a pas de mémoire allouée pour elles.

La fonction my_srand() copie juste sa valeur en entrée dans la variable rand_state interne.

my_rand() la prend, calcule le rand_state suivant, le coupe et le laisse dans le registre EAX.

La version non optimisée est plus verbeuse:

Listing 1.317: MSVC 2013 sans optimisation

```
_BSS    SEGMENT
_rand_state DD  01H DUP (?)
_BSS    ENDS

_init$ = 8
_srand  PROC
        push    ebp
        mov     ebp, esp
        mov     eax, DWORD PTR _init$[ebp]
        mov     DWORD PTR _rand_state, eax
        pop     ebp
        ret     0
_srand  ENDP

_TEXT   SEGMENT
_rand   PROC
        push    ebp
        mov     ebp, esp
        imul    eax, DWORD PTR _rand_state, 1664525
        mov     DWORD PTR _rand_state, eax
        mov     ecx, DWORD PTR _rand_state
        add     ecx, 1013904223 ; 3c6ef35fH
        mov     DWORD PTR _rand_state, ecx
        mov     eax, DWORD PTR _rand_state
        and     eax, 32767      ; 00007fffH
        pop     ebp
        ret     0
_rand   ENDP

_TEXT   ENDS
```

1.23.2 x64

La version x64 est essentiellement la même et utilise des registres 32-bit au lieu de 64-bit (car nous travaillons avec des valeurs de type *int* ici).

Mais my_srand() prend son argument en entrée dans le registre ECX plutôt que sur la pile:

Listing 1.318: MSVC 2013 x64 avec optimisation

```
_BSS    SEGMENT
rand_state DD   01H DUP (?)
_BSS    ENDS

init$ = 8
my_srand PROC
; ECX = argument en entrée
        mov     DWORD PTR rand_state, ecx
        ret     0
my_srand ENDP

_TEXT   SEGMENT
my_rand PROC
        imul    eax, DWORD PTR rand_state, 1664525 ; 0019660dH
        add     eax, 1013904223 ; 3c6ef35fH
        mov     DWORD PTR rand_state, eax
        and     eax, 32767      ; 00007fffH
        ret     0
my_rand ENDP

_TEXT   ENDS
```

Le compilateur GCC génère en grande partie le même code.

1.23.3 ARM 32-bit

Listing 1.319: avec optimisation Keil 6/2013 (Mode ARM)

```
my_srand PROC
        LDR      r1,|L0.52|  ; charger un pointeur sur rand_state
        STR      r0,[r1,#0]  ; sauver rand_state
        BX       lr
        ENDP

my_rand PROC
        LDR      r0,|L0.52|  ; charger un pointeur sur rand_state
        LDR      r2,|L0.56|  ; charger RNG_a
        LDR      r1,[r0,#0]  ; charger rand_state
        MUL      r1,r2,r1
        LDR      r2,|L0.60|  ; charger RNG_c
        ADD      r1,r1,r2
        STR      r1,[r0,#0]  ; sauver rand_state
; AND avec 0x7FFF :
        LSL      r0,r1,#17
        LSR      r0,r0,#17
        BX       lr
        ENDP

|L0.52|
        DCD      ||.data||
|L0.56|
        DCD      0x0019660d
|L0.60|
        DCD      0x3c6ef35f

        AREA ||.data||, DATA, ALIGN=2

rand_state
        DCD      0x00000000
```

Il n'est pas possible d'intégrer une constante 32-bit dans des instructions ARM, donc Keil doit les stocker à l'extérieur et en outre les charger. Une chose intéressante est qu'il n'est pas possible non plus d'intégrer la constante 0x7FFF. Donc ce que fait Keil est de décaler `rand_state` vers la gauche de 17 bits et ensuite la décale de 17 bits vers la droite. Ceci est analogue à la déclaration $(rand\_state \ll 17) \gg 17$ en C/C++. Il semble que ça soit une opération inutile, mais ce qu'elle fait est de mettre à zéro les 17 bits hauts, laissant les 15 bits bas inchangés, et c'est notre but après tout.

Keil avec optimisation pour le mode Thumb génère essentiellement le même code.

1.23.4 MIPS

Listing 1.320: avec optimisation GCC 4.4.5 (IDA)

```
my_srand :
; stocker $a0 dans rand_state :
                lui     $v0, (rand_state >> 16)
                jr      $ra
                sw      $a0, rand_state
my_rand :
; charger rand_state dans $v0 :
                lui     $v1, (rand_state >> 16)
                lw      $v0, rand_state
                or      $at, $zero  ; slot de délai de branchement
; multiplier rand_state dans $v0 par 1664525 (RNG_a) :
                sll     $a1, $v0, 2
                sll     $a0, $v0, 4
                addu    $a0, $a1, $a0
                sll     $a1, $a0, 6
```

```
                subu    $a0, $a1, $a0
                addu    $a0, $v0
                sll     $a1, $a0, 5
                addu    $a0, $a1
                sll     $a0, 3
                addu    $v0, $a0, $v0
                sll     $a0, $v0, 2
                addu    $v0, $a0
; ajouter 1013904223 (RNG_c)
; l'instruction LI est la fusion par IDA de LUI et ORI
                li      $a0, 0x3C6EF35F
                addu    $v0, $a0
; stocker dans rand_state :
                sw      $v0, (rand_state & 0xFFFF)($v1)
                jr      $ra
                andi    $v0, 0x7FFF ; slot de délai de branchement
```

Ouah, ici nous ne voyons qu'une seule constante (0x3C6EF35F ou 1013904223). Où est l'autre (1664525)?

Il semble que la multiplication soit effectuée en utilisant seulement des décalages et des additions! Vérifions cette hypothèse:

```
#define RNG_a 1664525

int f (int a)
{
        return a*RNG_a;
}
```

Listing 1.321: GCC 4.4.5 avec optimisation (IDA)

```
f :
                sll     $v1, $a0, 2
                sll     $v0, $a0, 4
                addu    $v0, $v1, $v0
                sll     $v1, $v0, 6
                subu    $v0, $v1, $v0
                addu    $v0, $a0
                sll     $v1, $v0, 5
                addu    $v0, $v1
                sll     $v0, 3
                addu    $a0, $v0, $a0
                sll     $v0, $a0, 2
                jr      $ra
                addu    $v0, $a0, $v0 ; branch delay slot
```

En effet!

Relocations MIPS

Nous allons nous concentrer sur comment les opérations comme charger et stocker dans la mémoire fonctionnent.

Les listings ici sont produits par IDA, qui cache certains détails.

Nous allons lancer objdump deux fois: pour obtenir le listing désassemblé et aussi la liste des relocations:

Listing 1.322: GCC 4.4.5 avec optimisation (objdump)

```
# objdump -D rand_O3.o

...

00000000 <my_srand> :
   0:   3c020000        lui     v0,0x0
   4:   03e00008        jr      ra
   8:   ac440000        sw      a0,0(v0)

0000000c <my_rand> :
```

```
   c :   3c030000        lui     v1,0x0
  10:   8c620000        lw      v0,0(v1)
  14:   00200825        move    at,at
  18:   00022880        sll     a1,v0,0x2
  1c :   00022100        sll     a0,v0,0x4
  20:   00a42021        addu    a0,a1,a0
  24:   00042980        sll     a1,a0,0x6
  28:   00a42023        subu    a0,a1,a0
  2c :   00822021        addu    a0,a0,v0
  30:   00042940        sll     a1,a0,0x5
  34:   00852021        addu    a0,a0,a1
  38:   000420c0        sll     a0,a0,0x3
  3c :   00821021        addu    v0,a0,v0
  40:   00022080        sll     a0,v0,0x2
  44:   00441021        addu    v0,v0,a0
  48:   3c043c6e        lui     a0,0x3c6e
  4c :   3484f35f        ori     a0,a0,0xf35f
  50:   00441021        addu    v0,v0,a0
  54:   ac620000        sw      v0,0(v1)
  58:   03e00008        jr      ra
  5c :   30427fff        andi    v0,v0,0x7fff

...

# objdump -r rand_O3.o

...

RELOCATION RECORDS FOR [.text]:
OFFSET   TYPE              VALUE
00000000 R_MIPS_HI16       .bss
00000008 R_MIPS_LO16       .bss
0000000c R_MIPS_HI16       .bss
00000010 R_MIPS_LO16       .bss
00000054 R_MIPS_LO16       .bss

...
```

Considérons les deux relocations pour la fonction `my_srand()`.

La première, pour l'adresse 0 a un type de R_MIPS_HI16 et la seconde pour l'adresse 8 a un type de R_MIPS_LO16.

Cela implique que l'adresse du début du segment .bss soit écrite dans les instructions à l'adresse 0 (partie haute de l'adresse) et 8 (partie basse de l'adresse).

La variable `rand_state` est au tout début du segment .bss.

Donc nous voyons des zéros dans les opérandes des instructions LUI et SW, car il n'y a encore rien ici— le compilateur ne sait pas quoi y écrire.

L'éditeur de liens va arranger cela, et la partie haute de l'adresse sera écrite dans l'opérande de LUI et la partie basse de l'adresse—dans l'opérande de SW.

SW va ajouter la partie basse de l'adresse avec le contenu du registre $V0 (la partie haute y est).

C'est la même histoire avec la fonction my_rand(): la relocation R_MIPS_HI16 indique à l'éditeur de liens d'écrire la partie haute.

Donc la partie haute de l'adresse de la variable rand_state se trouve dans le registre $V1.

L'instruction LW à l'adresse 0x10 ajoute les parties haute et basse et charge la valeur de la variable rand_state dans $V0.

L'instruction SW à l'adresse 0x54 fait à nouveau la somme et stocke la nouvelle valeur dans la variable globale rand_state.

IDA traite les relocations pendant le chargement, cachant ainsi ces détails, mais nous devons les garder à l'esprit.

1.23.5 Version thread-safe de l'exemple

La version thread-safe de l'exemple sera montrée plus tard: 4.2.1 on page 553.

1.24 Structures

Moyennant quelques ajustements, on peut considérer qu'une structure C/C++ n'est rien d'autre qu'un ensemble de variables, pas toutes nécessairement du même type, et toujours stockées en mémoire côte à côte [160].

1.24.1 MSVC: exemple SYSTEMTIME

Considérons la structure win32 SYSTEMTIME[161] qui décrit un instant dans le temps. Voici comment elle est définie:

Listing 1.323: WinBase.h

```
typedef struct _SYSTEMTIME {
  WORD wYear;
  WORD wMonth;
  WORD wDayOfWeek;
  WORD wDay;
  WORD wHour;
  WORD wMinute;
  WORD wSecond;
  WORD wMilliseconds;
} SYSTEMTIME, *PSYSTEMTIME;
```

Écrivons une fonction C pour récupérer l'instant qu'il est:

```
#include <windows.h>
#include <stdio.h>

void main()
{
    SYSTEMTIME t;
    GetSystemTime (&t);

    printf ("%04d-%02d-%02d %02d :%02d :%02d\n",
        t.wYear, t.wMonth, t.wDay,
        t.wHour, t.wMinute, t.wSecond);

    return;
};
```

Le résultat de la compilation avec MSVC 2010 donne:

Listing 1.324: MSVC 2010 /GS-

```
_t$ = -16 ; size = 16
_main      PROC
    push   ebp
    mov    ebp, esp
    sub    esp, 16
    lea    eax, DWORD PTR _t$[ebp]
    push   eax
    call   DWORD PTR __imp__GetSystemTime@4
    movzx  ecx, WORD PTR _t$[ebp+12]  ; wSecond
    push   ecx
    movzx  edx, WORD PTR _t$[ebp+10]  ; wMinute
    push   edx
    movzx  eax, WORD PTR _t$[ebp+8]  ; wHour
    push   eax
```

[160] AKA « conteneur hétérogène »
[161] MSDN: SYSTEMTIME structure

```
    movzx  ecx, WORD PTR _t$[ebp+6] ; wDay
    push   ecx
    movzx  edx, WORD PTR _t$[ebp+2] ; wMonth
    push   edx
    movzx  eax, WORD PTR _t$[ebp] ; wYear
    push   eax
    push   OFFSET $SG78811 ; '%04d-%02d-%02d %02d :%02d :%02d', 0aH, 00H
    call   _printf
    add    esp, 28
    xor    eax, eax
    mov    esp, ebp
    pop    ebp
    ret    0
_main      ENDP
```

16 octets sont réservés sur la pile pour cette structure, ce qui correspond exactement à `sizeof(WORD)*8`. La structure comprend effectivement 8 variables d'un WORD chacun.

Faites attention au fait que le premier membre de la structure est le champ `wYear`. On peut donc considérer que la fonction `GetSystemTime()`[162] reçoit comme argument un pointeur sur la structure SYSTEMTIME, ou bien qu'elle reçoit un pointeur sur le champ `wYear`. Et en fait c'est exactement la même chose! `GetSystemTime()` écrit l'année courante dans à l'adresse du WORD qu'il a reçu, avance de 2 octets, écrit le mois courant et ainsi de suite.

[162] MSDN: SYSTEMTIME structure

OllyDbg

Compilons cet exemple avec MSVC 2010 et les options `/GS- /MD`, puis exécutons le avec OllyDbg.

Ouvrons la fenêtre des données et celle de la pile à l'adresse du premier argument fourni à la fonction `GetSystemTime()`, puis attendons que cette fonction se termine. Nous constatons :

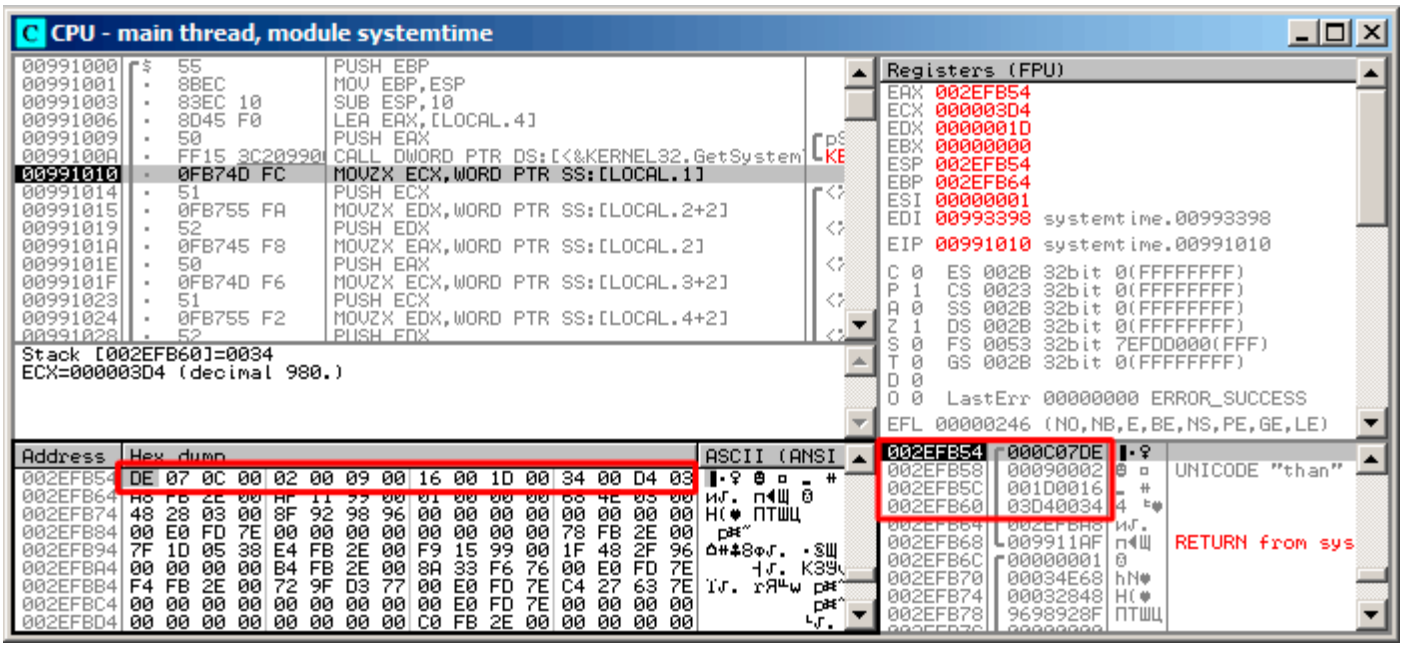


Fig. 1.104: OllyDbg : Juste après l'appel à `GetSystemTime()`

Sur mon ordinateur, le résultat de l'appel à la fonction est 9 décembre 2014, 22:29:52:

Listing 1.325: `printf()` output

```
2014-12-09 22:29:52
```

Nous observons donc ces 16 octets dans la fenêtre de données:

```
DE 07 0C 00 02 00 09 00 16 00 1D 00 34 00 D4 03
```

Chaque paire d'octets représente l'un des champs de la structure. Puisque nous sommes en mode petit-boutien l'octet de poids faible est situé en premier, suivi de l'octet de poids fort.

Les valeurs effectivement présentes en mémoire sont donc les suivantes:

nombre hexadécimal	nombre décimal	nom du champ
0x07DE	2014	wYear
0x000C	12	wMonth
0x0002	2	wDayOfWeek
0x0009	9	wDay
0x0016	22	wHour
0x001D	29	wMinute
0x0034	52	wSecond
0x03D4	980	wMilliseconds

Les mêmes valeurs apparaissent dans la fenêtre de la pile, mais elle y sont regroupées sous forme de valeurs 32 bits.

La fonction `printf()` utilise les valeurs qui lui sont nécessaires et les affiche à la console.

Bien que certaines valeurs telles que (`wDayOfWeek` et `wMilliseconds`) ne soient pas affichées par `printf()`, elles sont bien présentes en mémoire, prêtes à être utilisées.

Remplacer la structure par un tableau

Le fait que les champs d'une structure ne sont que des variables situées côte-à-côte peut être aisément démontré de la manière suivante. Tout en conservant à l'esprit la description de la structure SYSTEMTIME, il est possible de réécrire cet exemple simple de la manière suivante:

```
#include <windows.h>
#include <stdio.h>

void main()
{
    WORD array[8];
    GetSystemTime (array);

    printf ("%04d-%02d-%02d %02d :%02d :%02d\n",
        array[0] /* wYear */, array[1] /* wMonth */, array[3] /* wDay */,
        array[4] /* wHour */, array[5] /* wMinute */, array[6] /* wSecond */);

    return;
};
```

Le compilateur ronchonne certes un peu:

```
systemtime2.c(7)  : warning C4133 : 'function'  : incompatible types - from 'WORD [8]' to '↙
    ↳ LPSYSTEMTIME'
```

Mais, il consent quand même à produire le code suivant:

Listing 1.326: sans optimisation MSVC 2010

```
$SG78573 DB     '%04d-%02d-%02d %02d :%02d :%02d', 0aH, 00H

_array$ = -16   ; size = 16
_main   PROC
        push    ebp
        mov     ebp, esp
        sub     esp, 16
        lea     eax, DWORD PTR _array$[ebp]
        push    eax
        call    DWORD PTR __imp__GetSystemTime@4
        movzx   ecx, WORD PTR _array$[ebp+12] ; wSecond
        push    ecx
        movzx   edx, WORD PTR _array$[ebp+10] ; wMinute
        push    edx
        movzx   eax, WORD PTR _array$[ebp+8] ; wHoure
        push    eax
        movzx   ecx, WORD PTR _array$[ebp+6] ; wDay
        push    ecx
        movzx   edx, WORD PTR _array$[ebp+2] ; wMonth
        push    edx
        movzx   eax, WORD PTR _array$[ebp] ; wYear
        push    eax
        push    OFFSET $SG78573
        call    _printf
        add     esp, 28
        xor     eax, eax
        mov     esp, ebp
        pop     ebp
        ret     0
_main   ENDP
```

Qui fonctionne à l'identique du précédent!

Il est extrêmement intéressant de constater que le code assembleur produit est impossible à distinguer de celui produit par la compilation précédente.

Et ainsi celui qui observe ce code assembleur est incapable de décider avec certitude si une structure ou un tableau était déclaré dans le code source en C.

Cela étant, aucun esprit sain ne s'amuserait à déclarer un tableau ici. Car il faut aussi compter avec la possibilité que la structure soit modifiée par les développeurs, que les champs soient triés dans un autre ordre ...

Nous n'étudierons pas cet exemple avec OllyDbg, car les résultats seraient identiques à ceux que nous avons observé en utilisant la structure.

1.24.2 Allouons de l'espace pour une structure avec malloc()

Il est parfois plus simple de placer une structure sur le heap que sur la pile:

```
#include <windows.h>
#include <stdio.h>

void main()
{
    SYSTEMTIME *t;

    t=(SYSTEMTIME *)malloc (sizeof (SYSTEMTIME));

    GetSystemTime (t);

    printf ("%04d-%02d-%02d %02d :%02d :%02d\n",
        t->wYear, t->wMonth, t->wDay,
        t->wHour, t->wMinute, t->wSecond);

    free (t);

    return;
};
```

Compilons cet exemple en utilisant l'option (/0x) qui facilitera nos observations.

Listing 1.327: MSVC avec optimisation

```
_main      PROC
    push   esi
    push   16
    call   _malloc
    add    esp, 4
    mov    esi, eax
    push   esi
    call   DWORD PTR __imp__GetSystemTime@4
    movzx  eax, WORD PTR [esi+12] ; wSecond
    movzx  ecx, WORD PTR [esi+10] ; wMinute
    movzx  edx, WORD PTR [esi+8] ; wHour
    push   eax
    movzx  eax, WORD PTR [esi+6] ; wDay
    push   ecx
    movzx  ecx, WORD PTR [esi+2] ; wMonth
    push   edx
    movzx  edx, WORD PTR [esi] ; wYear
    push   eax
    push   ecx
    push   edx
    push   OFFSET $SG78833
    call   _printf
    push   esi
    call   _free
    add    esp, 32
    xor    eax, eax
    pop    esi
    ret    0
_main      ENDP
```

Puisque `sizeof(SYSTEMTIME) = 16` c'est aussi le nombre d'octets qui doit être alloué par `malloc()`. Celui-ci renvoie dans le registre EAX un pointeur vers un bloc mémoire fraîchement alloué. Puis le pointeur est copié dans le registre ESI. La fonction win32 `GetSystemTime()` prend soin que la valeur de ESI soit la

même à l'issue de la fonction que lors de son appel. C'est pourquoi nous pouvons continuer à l'utiliser après sans avoir eu besoin de le sauvegarder.

Tiens, une nouvelle instruction —MOVZX (*Move with Zero eXtend*). La plupart du temps, elle peut être utilisée comme MOVSX. La différence est qu'elle positionne systématiquement les bits supplémentaires à 0. Elle est utilisée ici car `printf()` attend une valeur sur 32 bits et que nous ne disposons que d'un WORD dans la structure —c'est à dire une valeur non signée sur 16 bits. Il nous faut donc forcer à zéro les bits 16 à 31 lorsque le WORD est copié dans un *int*, sinon nous risquons de récupérer des bits résiduels de la précédente opération sur le registre.

Dans cet exemple, il reste possible de représenter la structure sous forme d'un tableau de 8 WORDs:

```
#include <windows.h>
#include <stdio.h>

void main()
{
    WORD *t;

    t=(WORD *)malloc (16);

    GetSystemTime (t);

    printf ("%04d-%02d-%02d %02d :%02d :%02d\n",
        t[0] /* wYear */, t[1] /* wMonth */, t[3] /* wDay */,
        t[4] /* wHour */, t[5] /* wMinute */, t[6] /* wSecond */);

    free (t);

    return;
};
```

Nous avons alors:

Listing 1.328: MSVC avec optimisation

```
$SG78594 DB     '%04d-%02d-%02d %02d :%02d :%02d', 0aH, 00H

_main   PROC
        push    esi
        push    16
        call    _malloc
        add     esp, 4
        mov     esi, eax
        push    esi
        call    DWORD PTR __imp__GetSystemTime@4
        movzx   eax, WORD PTR [esi+12]
        movzx   ecx, WORD PTR [esi+10]
        movzx   edx, WORD PTR [esi+8]
        push    eax
        movzx   eax, WORD PTR [esi+6]
        push    ecx
        movzx   ecx, WORD PTR [esi+2]
        push    edx
        movzx   edx, WORD PTR [esi]
        push    eax
        push    ecx
        push    edx
        push    OFFSET $SG78594
        call    _printf
        push    esi
        call    _free
        add     esp, 32
        xor     eax, eax
        pop     esi
        ret     0
_main   ENDP
```

Encore une fois nous obtenons un code qu'il n'est pas possible de discerner du précédent.

Et encore une fois, vous n'avez pas intérêt à faire cela, sauf si vous savez exactement ce que vous faites.

1.24.3 UNIX: struct tm

Linux

Prenons pour exemple la structure `tm` dans l'en-tête `time.h` de Linux:

```
#include <stdio.h>
#include <time.h>

void main()
{
    struct tm t;
    time_t unix_time;

    unix_time=time(NULL);

    localtime_r (&unix_time, &t);

    printf ("Year : %d\n", t.tm_year+1900);
    printf ("Month : %d\n", t.tm_mon);
    printf ("Day : %d\n", t.tm_mday);
    printf ("Hour : %d\n", t.tm_hour);
    printf ("Minutes : %d\n", t.tm_min);
    printf ("Seconds : %d\n", t.tm_sec);
};
```

Compilons l'exemple avec GCC 4.4.1:

Listing 1.329: GCC 4.4.1

```
main proc near
     push    ebp
     mov     ebp, esp
     and     esp, 0FFFFFFF0h
     sub     esp, 40h
     mov     dword ptr [esp], 0 ; premier argument de la fonction time()
     call    time
     mov     [esp+3Ch], eax
     lea     eax, [esp+3Ch]  ; récupération de la valeur retournée par time()
     lea     edx, [esp+10h]  ; la structure tm est à l'adresse ESP+10h
     mov     [esp+4], edx    ; passons le pointeur vers la structure begin
     mov     [esp], eax      ; ... et le pointeur retourné par time()
     call    localtime_r
     mov     eax, [esp+24h]  ; tm_year
     lea     edx, [eax+76Ch] ; edx=eax+1900
     mov     eax, offset format ; "Year : %d\n"
     mov     [esp+4], edx
     mov     [esp], eax
     call    printf
     mov     edx, [esp+20h]      ; tm_mon
     mov     eax, offset aMonthD ; "Month : %d\n"
     mov     [esp+4], edx
     mov     [esp], eax
     call    printf
     mov     edx, [esp+1Ch]     ; tm_mday
     mov     eax, offset aDayD  ; "Day : %d\n"
     mov     [esp+4], edx
     mov     [esp], eax
     call    printf
     mov     edx, [esp+18h]     ; tm_hour
     mov     eax, offset aHourD ; "Hour : %d\n"
     mov     [esp+4], edx
     mov     [esp], eax
     call    printf
     mov     edx, [esp+14h]        ; tm_min
     mov     eax, offset aMinutesD ; "Minutes : %d\n"
     mov     [esp+4], edx
     mov     [esp], eax
     call    printf
     mov     edx, [esp+10h]
```

```
    mov     eax, offset aSecondsD ; "Seconds : %d\n"
    mov     [esp+4], edx          ; tm_sec
    mov     [esp], eax
    call    printf
    leave
    retn
main endp
```

IDA n'a pas utilisé le nom des variables locales pour identifier les éléments de la pile. Mais comme nous sommes déjà des rétro ingénieurs expérimentés :-) nous pouvons nous en passer dans cet exemple simple.

Notez l'instruction `lea edx, [eax+76Ch]` —qui incrémente la valeur de EAX de 0x76C (1900) sans modifier aucun des drapeaux. Référez-vous également à la section au sujet de LEA (**??** on page ??).

GDB

Tentons de charger l'exemple dans GDB [163] :

Listing 1.330: GDB

```
dennis@ubuntuvm :~/polygon$ date
Mon Jun  2 18:10:37 EEST 2014
dennis@ubuntuvm :~/polygon$ gcc GCC_tm.c -o GCC_tm
dennis@ubuntuvm :~/polygon$ gdb GCC_tm
GNU gdb (GDB) 7.6.1-ubuntu
...
Reading symbols from /home/dennis/polygon/GCC_tm...(no debugging symbols found)...done.
(gdb) b printf
Breakpoint 1 at 0x8048330
(gdb) run
Starting program : /home/dennis/polygon/GCC_tm

Breakpoint 1, __printf (format=0x80485c0 "Year : %d\n") at printf.c :29
29      printf.c : No such file or directory.
(gdb) x/20x $esp
0xbffff0dc :     0x080484c3      0x080485c0      0x000007de      0x00000000
0xbffff0ec :     0x08048301      0x538c93ed      0x00000025      0x0000000a
0xbffff0fc :     0x00000012      0x00000002      0x00000005      0x00000072
0xbffff10c :     0x00000001      0x00000098      0x00000001      0x00002a30
0xbffff11c :     0x0804b090      0x08048530      0x00000000      0x00000000
(gdb)
```

Nous retrouvons facilement notre structure dans la pile. Commençons par observer sa définition dans *time.h* :

Listing 1.331: time.h

```
struct tm
{
  int   tm_sec;
  int   tm_min;
  int   tm_hour;
  int   tm_mday;
  int   tm_mon;
  int   tm_year;
  int   tm_wday;
  int   tm_yday;
  int   tm_isdst;
};
```

Faites attention au fait qu'ici les champs sont des *int* sur 32 bits et non des WORD comme dans SYSTEMTIME.

Voici donc les champs de notre structure tels qu'ils sont présents dans la pile:

[163] Le résultat *date* est légèrement modifié pour les besoins de la démonstration, car il est bien entendu impossible d'exécuter GDB aussi rapidement.

```
0xbffff0dc :     0x080484c3      0x080485c0      0x000007de      0x00000000
0xbffff0ec :     0x08048301      0x538c93ed      0x00000025 sec  0x0000000a min
0xbffff0fc :     0x00000012 hour 0x00000002 mday 0x00000005 mon  0x00000072 year
0xbffff10c :     0x00000001 wday 0x00000098 yday 0x00000001 isdst0x00002a30
0xbffff11c :     0x0804b090      0x08048530      0x00000000      0x00000000
```

Représentés sous forme tabulaire, cela donne:

Hexadécimal	Décimal	nom
0x00000025	37	tm_sec
0x0000000a	10	tm_min
0x00000012	18	tm_hour
0x00000002	2	tm_mday
0x00000005	5	tm_mon
0x00000072	114	tm_year
0x00000001	1	tm_wday
0x00000098	152	tm_yday
0x00000001	1	tm_isdst

C'est très similaire à SYSTEMTIME (1.24.1 on page 346), Là encore certains champs sont présents qui ne sont pas utilisés tels que tm_wday, tm_yday, tm_isdst.

ARM

avec optimisation Keil 6/2013 (Mode Thumb)

Même exemple:

Listing 1.332: avec optimisation Keil 6/2013 (Mode Thumb)

```
var_38 = -0x38
var_34 = -0x34
var_30 = -0x30
var_2C = -0x2C
var_28 = -0x28
var_24 = -0x24
timer  = -0xC

        PUSH    {LR}
        MOVS    R0, #0          ; timer
        SUB     SP, SP, #0x34
        BL      time
        STR     R0, [SP,#0x38+timer]
        MOV     R1, SP          ; tp
        ADD     R0, SP, #0x38+timer ; timer
        BL      localtime_r
        LDR     R1, =0x76C
        LDR     R0, [SP,#0x38+var_24]
        ADDS    R1, R0, R1
        ADR     R0, aYearD      ; "Year : %d\n"
        BL      __2printf
        LDR     R1, [SP,#0x38+var_28]
        ADR     R0, aMonthD     ; "Month : %d\n"
        BL      __2printf
        LDR     R1, [SP,#0x38+var_2C]
        ADR     R0, aDayD       ; "Day : %d\n"
        BL      __2printf
        LDR     R1, [SP,#0x38+var_30]
        ADR     R0, aHourD      ; "Hour : %d\n"
        BL      __2printf
        LDR     R1, [SP,#0x38+var_34]
        ADR     R0, aMinutesD   ; "Minutes : %d\n"
        BL      __2printf
        LDR     R1, [SP,#0x38+var_38]
        ADR     R0, aSecondsD   ; "Seconds : %d\n"
        BL      __2printf
        ADD     SP, SP, #0x34
```

```
        POP     {PC}
```

avec optimisation Xcode 4.6.3 (LLVM) (Mode Thumb-2)

IDA reconnaît la structure `tm` (car le logiciel a connaissance des arguments attendus par les fonctions de la librairie telles que `localtime_r()`),

Il peut donc afficher les éléments de la structure ainsi que leurs noms.

Listing 1.333: avec optimisation Xcode 4.6.3 (LLVM) (Mode Thumb-2)

```
var_38 = -0x38
var_34 = -0x34

        PUSH {R7,LR}
        MOV  R7, SP
        SUB  SP, SP, #0x30
        MOVS R0, #0  ; time_t *
        BLX  _time
        ADD  R1, SP, #0x38+var_34 ; struct tm *
        STR  R0, [SP,#0x38+var_38]
        MOV  R0, SP  ; time_t *
        BLX  _localtime_r
        LDR  R1, [SP,#0x38+var_34.tm_year]
        MOV  R0, 0xF44 ; "Year : %d\n"
        ADD  R0, PC  ; char *
        ADDW R1, R1, #0x76C
        BLX  _printf
        LDR  R1, [SP,#0x38+var_34.tm_mon]
        MOV  R0, 0xF3A ; "Month : %d\n"
        ADD  R0, PC  ; char *
        BLX  _printf
        LDR  R1, [SP,#0x38+var_34.tm_mday]
        MOV  R0, 0xF35 ; "Day : %d\n"
        ADD  R0, PC  ; char *
        BLX  _printf
        LDR  R1, [SP,#0x38+var_34.tm_hour]
        MOV  R0, 0xF2E ; "Hour : %d\n"
        ADD  R0, PC  ; char *
        BLX  _printf
        LDR  R1, [SP,#0x38+var_34.tm_min]
        MOV  R0, 0xF28 ; "Minutes : %d\n"
        ADD  R0, PC  ; char *
        BLX  _printf
        LDR  R1, [SP,#0x38+var_34]
        MOV  R0, 0xF25 ; "Seconds : %d\n"
        ADD  R0, PC  ; char *
        BLX  _printf
        ADD  SP, SP, #0x30
        POP  {R7,PC}

...

00000000 tm        struc ; (sizeof=0x2C, standard type)
00000000 tm_sec    DCD ?
00000004 tm_min    DCD ?
00000008 tm_hour   DCD ?
0000000C tm_mday   DCD ?
00000010 tm_mon    DCD ?
00000014 tm_year   DCD ?
00000018 tm_wday   DCD ?
0000001C tm_yday   DCD ?
00000020 tm_isdst  DCD ?
00000024 tm_gmtoff DCD ?
00000028 tm_zone   DCD ? ; offset
0000002C tm        ends
```

MIPS

Listing 1.334: avec optimisation GCC 4.4.5 (IDA)

```
main :

; Le nommage des champs dans les structures a été effectué manuellement car IDA ne les connaît pas :

var_40          = -0x40
var_38          = -0x38
seconds         = -0x34
minutes         = -0x30
hour            = -0x2C
day             = -0x28
month           = -0x24
year            = -0x20
var_4           = -4

                lui     $gp, (__gnu_local_gp >> 16)
                addiu   $sp, -0x50
                la      $gp, (__gnu_local_gp & 0xFFFF)
                sw      $ra, 0x50+var_4($sp)
                sw      $gp, 0x50+var_40($sp)
                lw      $t9, (time & 0xFFFF)($gp)
                or      $at, $zero ; Gaspillage par NOP du délai de branchement
                jalr    $t9
                move    $a0, $zero ; Gaspillage par NOP du délai de branchement
                lw      $gp, 0x50+var_40($sp)
                addiu   $a0, $sp, 0x50+var_38
                lw      $t9, (localtime_r & 0xFFFF)($gp)
                addiu   $a1, $sp, 0x50+seconds
                jalr    $t9
                sw      $v0, 0x50+var_38($sp) ; Utilisation du délai de branchement
                lw      $gp, 0x50+var_40($sp)
                lw      $a1, 0x50+year($sp)
                lw      $t9, (printf & 0xFFFF)($gp)
                la      $a0, $LC0        # "Year : %d\n"
                jalr    $t9
                addiu   $a1, 1900 ; branch delay slot
                lw      $gp, 0x50+var_40($sp)
                lw      $a1, 0x50+month($sp)
                lw      $t9, (printf & 0xFFFF)($gp)
                lui     $a0, ($LC1 >> 16)  # "Month : %d\n"
                jalr    $t9
                la      $a0, ($LC1 & 0xFFFF)  # "Month : %d\n" ; Utilisation du délai de branchement
                lw      $gp, 0x50+var_40($sp)
                lw      $a1, 0x50+day($sp)
                lw      $t9, (printf & 0xFFFF)($gp)
                lui     $a0, ($LC2 >> 16)  # "Day : %d\n"
                jalr    $t9
                la      $a0, ($LC2 & 0xFFFF)  # "Day : %d\n" ; Utilisation du délai de branchement
                lw      $gp, 0x50+var_40($sp)
                lw      $a1, 0x50+hour($sp)
                lw      $t9, (printf & 0xFFFF)($gp)
                lui     $a0, ($LC3 >> 16)  # "Hour : %d\n"
                jalr    $t9
                la      $a0, ($LC3 & 0xFFFF)  # "Hour : %d\n" ; Utilisation du délai de branchement
                lw      $gp, 0x50+var_40($sp)
                lw      $a1, 0x50+minutes($sp)
                lw      $t9, (printf & 0xFFFF)($gp)
                lui     $a0, ($LC4 >> 16)  # "Minutes : %d\n"
                jalr    $t9
                la      $a0, ($LC4 & 0xFFFF)  # "Minutes : %d\n" ; Utilisation du délai de branchement
                lw      $gp, 0x50+var_40($sp)
                lw      $a1, 0x50+seconds($sp)
```

```
62                lw      $t9, (printf & 0xFFFF)($gp)
63                lui     $a0, ($LC5 >> 16)  # "Seconds : %d\n"
64                jalr    $t9
65                la      $a0, ($LC5 & 0xFFFF)  # "Seconds : %d\n" ; Utilisation du délai de ↙
    ↳ branchement
66                lw      $ra, 0x50+var_4($sp)
67                or      $at, $zero ; Gaspillage par NOP du délai de branchement
68                jr      $ra
69                addiu   $sp, 0x50
70
71 $LC0 :           .ascii "Year : %d\n"<0>
72 $LC1 :           .ascii "Month : %d\n"<0>
73 $LC2 :           .ascii "Day : %d\n"<0>
74 $LC3 :           .ascii "Hour : %d\n"<0>
75 $LC4 :           .ascii "Minutes : %d\n"<0>
76 $LC5 :           .ascii "Seconds : %d\n"<0>
```

Dans cet exemple, le retard à l'exécution des instructions de branchement peuvent nous égarer.

L'instruction `addiu $a1, 1900` en ligne 35 qui ajoute la valeur 1900 à l'année en est un exemple. N'oubliez pas qu'elle est exécutée avant que le l'instruction JALR ne fasse son effet.

Structure comme un ensemble de valeurs

Afin d'illustrer le fait qu'une structure n'est qu'une collection de variables située côte-à-côte, retravaillons notre exemple sur la base de la définition de la structure *tm* : liste.1.331.

```
#include <stdio.h>
#include <time.h>

void main()
{
    int tm_sec, tm_min, tm_hour, tm_mday, tm_mon, tm_year, tm_wday, tm_yday, tm_isdst;
    time_t unix_time;

    unix_time=time(NULL);

    localtime_r (&unix_time, &tm_sec);

    printf ("Year : %d\n", tm_year+1900);
    printf ("Month : %d\n", tm_mon);
    printf ("Day : %d\n", tm_mday);
    printf ("Hour : %d\n", tm_hour);
    printf ("Minutes : %d\n", tm_min);
    printf ("Seconds : %d\n", tm_sec);
};
```

N.B. Le pointeur vers le champ `tm_sec` est passé comme argument de la fonction `localtime_r`, en tant que premier élément de la « structure ».

Le compilateur nous alerte:

Listing 1.335: GCC 4.7.3

```
GCC_tm2.c : In function 'main' :
GCC_tm2.c :11:5: warning : passing argument 2 of 'localtime_r' from incompatible pointer type [↙
    ↳ enabled by default]
In file included from GCC_tm2.c :2:0:
/usr/include/time.h :59:12: note : expected 'struct tm *' but argument is of type 'int *'
```

Mais il génère cependant un fragment exécutable correspondant au code assembleur suivant:

Listing 1.336: GCC 4.7.3

```
main      proc near

var_30    = dword ptr -30h
var_2C    = dword ptr -2Ch
unix_time = dword ptr -1Ch
```

```
tm_sec    = dword ptr -18h
tm_min    = dword ptr -14h
tm_hour   = dword ptr -10h
tm_mday   = dword ptr -0Ch
tm_mon    = dword ptr -8
tm_year   = dword ptr -4

          push    ebp
          mov     ebp, esp
          and     esp, 0FFFFFFF0h
          sub     esp, 30h
          call    __main
          mov     [esp+30h+var_30], 0 ; arg 0
          call    time
          mov     [esp+30h+unix_time], eax
          lea     eax, [esp+30h+tm_sec]
          mov     [esp+30h+var_2C], eax
          lea     eax, [esp+30h+unix_time]
          mov     [esp+30h+var_30], eax
          call    localtime_r
          mov     eax, [esp+30h+tm_year]
          add     eax, 1900
          mov     [esp+30h+var_2C], eax
          mov     [esp+30h+var_30], offset aYearD ; "Year : %d\n"
          call    printf
          mov     eax, [esp+30h+tm_mon]
          mov     [esp+30h+var_2C], eax
          mov     [esp+30h+var_30], offset aMonthD ; "Month : %d\n"
          call    printf
          mov     eax, [esp+30h+tm_mday]
          mov     [esp+30h+var_2C], eax
          mov     [esp+30h+var_30], offset aDayD ; "Day : %d\n"
          call    printf
          mov     eax, [esp+30h+tm_hour]
          mov     [esp+30h+var_2C], eax
          mov     [esp+30h+var_30], offset aHourD ; "Hour : %d\n"
          call    printf
          mov     eax, [esp+30h+tm_min]
          mov     [esp+30h+var_2C], eax
          mov     [esp+30h+var_30], offset aMinutesD ; "Minutes : %d\n"
          call    printf
          mov     eax, [esp+30h+tm_sec]
          mov     [esp+30h+var_2C], eax
          mov     [esp+30h+var_30], offset aSecondsD ; "Seconds : %d\n"
          call    printf
          leave
          retn
main      endp
```

Ce code est similaire à ce que nous avons déjà vue et il n'est pas possible de dire si le code source original contenait une structure ou un groupe de variables.

Et cela fonctionne. Mais encore une fois ce n'est pas une bonne pratique.

En règle générale les compilateurs en l'absence d'optimisation allouent les variables sur la pile dans le même ordre que celui dans lequel elles ont été déclarées dans le code source. Pour autant, ce n'est pas une garantie.

Par ailleurs certains compilateurs peuvent vous avertir que les variables `tm_year`, `tm_mon`, `tm_mday`, `tm_hour`, `tm_min` n'ont pas été initialisées avant leur utilisation, mais resteront muets au sujet de `tm_sec`

Le compilateur lui non plus ne sait pas qu'ils sont appelés à être initialisés par la fonction `localtime_r()`.

Nous avons chois cet exemple car tous les champs de la structure sont de type *int*.

Tout ceci ne fonctionnerait pas sir les champs de la structure étaient des `WORD` de 16 bits, tel que dans le cas de la structure `SYSTEMTIME` structure—`GetSystemTime()` les initialiserait de manière erronée (puisque les variables locales sont alignées sur des frontières de 32bits). Vous en saurez plus à ce sujet dans la prochaine section: « Organisation des champs dans la structure » (1.24.4 on page 361).

Une structure n'est donc qu'un groupe de variables disposées côte-à-côte en mémoire. Nous pouvons dire

que la structure est une instruction adressée au compilateur et l'obligeant à conserver le groupement des variables. Cela étant dans les toutes premières versions du langage C (avant 1972), la notion de structure n'existait pas encore [Dennis M. Ritchie, *The development of the C language*, (1993)][164].

Pas d'exemple de débogage ici. Le comportement est toujours le même.

Une structure sous forme de table de 32 bits

```
#include <stdio.h>
#include <time.h>

void main()
{
    struct tm t;
    time_t unix_time;
    int i;

    unix_time=time(NULL);

    localtime_r (&unix_time, &t);

    for (i=0; i<9; i++)
    {
        int tmp=((int*)&t)[i];
        printf ("0x%08X (%d)\n", tmp, tmp);
    };
};
```

Nous n'avons qu'à utiliser l'opérateur *cast* pour transformer notre pointeur vers une structure en un tableau de *int*'s. Et cela fonctionne ! Nous avons exécuté l'exemple à 23h51m45s le 26 juillet 2014.

```
0x0000002D (45)
0x00000033 (51)
0x00000017 (23)
0x0000001A (26)
0x00000006 (6)
0x00000072 (114)
0x00000006 (6)
0x000000CE (206)
0x00000001 (1)
```

Les variables sont dans le même ordre que celui dans lequel elles apparaissent dans la définition de la structure: 1.331 on page 353.

Nous avons effectué la compilation avec:

Listing 1.337: avec optimisation GCC 4.8.1

```
main            proc near
                push    ebp
                mov     ebp, esp
                push    esi
                push    ebx
                and     esp, 0FFFFFFF0h
                sub     esp, 40h
                mov     dword ptr [esp], 0 ; timer
                lea     ebx, [esp+14h]
                call    _time
                lea     esi, [esp+38h]
                mov     [esp+4], ebx    ; tp
                mov     [esp+10h], eax
                lea     eax, [esp+10h]
                mov     [esp], eax      ; timer
                call    _localtime_r
                nop
                lea     esi, [esi+0]    ; NOP
loc_80483D8 :
```

[164] Aussi disponible en http://go.yurichev.com/17264

```
; EBX pointe sur la structure, ESI pointe sur la fin de celle-ci.
                mov     eax, [ebx]      ; get 32-bit word from array
                add     ebx, 4          ; prochain champ de la structure
                mov     dword ptr [esp+4], offset a0x08xD ; "0x%08X (%d)\n"
                mov     dword ptr [esp], 1
                mov     [esp+0Ch], eax  ; passage des arguments à printf()
                mov     [esp+8], eax
                call    ___printf_chk
                cmp     ebx, esi        ; Avons-nous atteint la fin de la structure ?
                jnz     short loc_80483D8       ; non - alors passons à la prochaine valeur
                lea     esp, [ebp-8]
                pop     ebx
                pop     esi
                pop     ebp
                retn
main            endp
```

En fait, l'espace dans la pile est tout d'abord traité comme une structure, puis ensuite comme un tableau.

Le pointeur sur le tableau permet même de modifier les champs de la structure.

Et encore une fois cette manière de procéder est extrêmement douteuse et pas du tout recommandée pour l'écriture d'un code qui atterrira en production.

Exercice

Tentez de modifier (en l'augmentant de 1) le numéro du mois, en traitant la structure comme s'il s'agissait d'un tableau.

Une structure sous forme d'un tableau d'octets

Nous pouvons aller plus loin. Utilisons l'opérateur *cast* pour transformer le pointeur en un tableau d'octets, puis affichons son contenu:

```
#include <stdio.h>
#include <time.h>

void main()
{
    struct tm t;
    time_t unix_time;
    int i, j;

    unix_time=time(NULL);

    localtime_r (&unix_time, &t);

    for (i=0; i<9; i++)
    {
        for (j=0; j<4; j++)
            printf ("0x%02X ", ((unsigned char*)&t)[i*4+j]);
        printf ("\n");
    };
};
```

```
0x2D 0x00 0x00 0x00
0x33 0x00 0x00 0x00
0x17 0x00 0x00 0x00
0x1A 0x00 0x00 0x00
0x06 0x00 0x00 0x00
0x72 0x00 0x00 0x00
0x06 0x00 0x00 0x00
0xCE 0x00 0x00 0x00
0x01 0x00 0x00 0x00
```

Cet exemple a été exécuté à 23h51m45s le 26 juillet 2014 [165]. Les valeurs sont identiques à celles du précédent affichage (1.24.3 on page 359), et bien entendu l'octet de poids faible figure en premier puisque nous sommes sur une architecture de type little-endian (2.8 on page 468).

Listing 1.338: avec optimisation GCC 4.8.1

```
main            proc near
                push    ebp
                mov     ebp, esp
                push    edi
                push    esi
                push    ebx
                and     esp, 0FFFFFFF0h
                sub     esp, 40h
                mov     dword ptr [esp], 0 ; timer
                lea     esi, [esp+14h]
                call    _time
                lea     edi, [esp+38h]  ; struct end
                mov     [esp+4], esi    ; tp
                mov     [esp+10h], eax
                lea     eax, [esp+10h]
                mov     [esp], eax      ; timer
                call    _localtime_r
                lea     esi, [esi+0]    ; NOP
; ESI pointe sur la structure située sur la pile. EDI pointe sur la fin de la structure.
loc_8048408 :
                xor     ebx, ebx        ; j=0

loc_804840A :
                movzx   eax, byte ptr [esi+ebx] ; load byte
                add     ebx, 1          ; j=j+1
                mov     dword ptr [esp+4], offset a0x02x ; "0x%02X "
                mov     dword ptr [esp], 1
                mov     [esp+8], eax    ; Fourniture à printf() des octets qui ont été chargés
                call    ___printf_chk
                cmp     ebx, 4
                jnz     short loc_804840A
; Imprime un retour chariot (CR)
                mov     dword ptr [esp], 0Ah ; c
                add     esi, 4
                call    _putchar
                cmp     esi, edi        ; Avons nous atteint la fin de la structure ?
                jnz     short loc_8048408 ; j=0
                lea     esp, [ebp-0Ch]
                pop     ebx
                pop     esi
                pop     edi
                pop     ebp
                retn
main            endp
```

1.24.4 Organisation des champs dans la structure

L'arrangement des champs au sein d'une structure est un élément très important[166].

Prenons un exemple simple:

```
#include <stdio.h>

struct s
{
    char a;
    int b;
    char c;
    int d;
};
```

[165] Les dates et heures sont les mêmes dans tous les exemples. Elles ont été éditées pour la clarté de la démonstration.
[166] See also: Wikipedia: Alignement en mémoire

```
void f(struct s s)
{
    printf ("a=%d; b=%d; c=%d; d=%d\n", s.a, s.b, s.c, s.d);
};

int main()
{
    struct s tmp;
    tmp.a=1;
    tmp.b=2;
    tmp.c=3;
    tmp.d=4;
    f(tmp);
};
```

Nous avons deux champs de type *char* (occupant chacun un octet) et deux autres —de type *int* (comportant 4 octets chacun).

x86

Le résultat de la compilation est:

Listing 1.339: MSVC 2012 /GS- /Ob0

```
1  _tmp$ = -16
2  _main    PROC
3      push   ebp
4      mov    ebp, esp
5      sub    esp, 16
6      mov    BYTE PTR _tmp$[ebp], 1     ; initialisation du champ a
7      mov    DWORD PTR _tmp$[ebp+4], 2  ; initialisation du champ b
8      mov    BYTE PTR _tmp$[ebp+8], 3   ; initialisation du champ c
9      mov    DWORD PTR _tmp$[ebp+12], 4 ; initialisation du champ d
10     sub    esp, 16                    ; Allocation d'espace pour la structure temporaire
11     mov    eax, esp
12     mov    ecx, DWORD PTR _tmp$[ebp]  ; Copie de notre structure dans la structure temporaire
13     mov    DWORD PTR [eax], ecx
14     mov    edx, DWORD PTR _tmp$[ebp+4]
15     mov    DWORD PTR [eax+4], edx
16     mov    ecx, DWORD PTR _tmp$[ebp+8]
17     mov    DWORD PTR [eax+8], ecx
18     mov    edx, DWORD PTR _tmp$[ebp+12]
19     mov    DWORD PTR [eax+12], edx
20     call    _f
21     add    esp, 16
22     xor    eax, eax
23     mov    esp, ebp
24     pop    ebp
25     ret    0
26 _main    ENDP
27
28 _s$ = 8 ; size = 16
29 ?f@@YAXUs@@@Z PROC ; f
30     push   ebp
31     mov    ebp, esp
32     mov    eax, DWORD PTR _s$[ebp+12]
33     push   eax
34     movsx  ecx, BYTE PTR _s$[ebp+8]
35     push   ecx
36     mov    edx, DWORD PTR _s$[ebp+4]
37     push   edx
38     movsx  eax, BYTE PTR _s$[ebp]
39     push   eax
40     push   OFFSET $SG3842
41     call   _printf
42     add    esp, 20
43     pop    ebp
44     ret    0
```

```
45 ?f@@YAXUs@@@Z ENDP ; f
46 _TEXT    ENDS
```

Nous passons la structure comme un tout, mais en réalité nous pouvons constater que la structure est copiée dans un espace temporaire. De l'espace est réservé pour cela ligne 10 et les 4 champs sont copiées par les lignes de 12 ... 19), puis le pointeur sur l'espace temporaire est passé à la fonction.

La structure est recopiée au cas où la fonction `f()` viendrait à en modifier le contenu. Si cela arrive, la copie de la structure qui existe dans `main()` restera inchangée.

Nous pourrions également utiliser des pointeurs C/C++. Le résulta demeurerait le même, sans qu'il soit nécessaire de procéder à la copie.

Nous observons que l'adresse de chaque champ est alignée sur un multiple de 4 octets. C'est pourquoi chaque *char* occupe 4 octets (de même qu'un *int*). Pourquoi en est-il ainsi? La réponse se situe au niveau de la CPU. Il est plus facile et performant pour elle d'accéder la mémoire et de gérer le cache de données en utilisant des adresses alignées.

En revanche ce n'est pas très économique en terme d'espace.

Tentons maintenant une compilation avec l'option (/Zp1) (*/Zp[n] indique qu'il faut compresser les structures en utilisant des frontières tous les n octets*).

Listing 1.340: MSVC 2012 /GS- /Zp1

```
 1 _main    PROC
 2     push   ebp
 3     mov    ebp, esp
 4     sub    esp, 12
 5     mov    BYTE PTR _tmp$[ebp], 1     ; Initialisation du champ a
 6     mov    DWORD PTR _tmp$[ebp+1], 2  ; Initialisation du champ b
 7     mov    BYTE PTR _tmp$[ebp+5], 3   ; Initialisation du champ c
 8     mov    DWORD PTR _tmp$[ebp+6], 4  ; Initialisation du champ d
 9     sub    esp, 12                    ; Allocation d'espace pour la structure temporaire
10     mov    eax, esp
11     mov    ecx, DWORD PTR _tmp$[ebp]  ; Copie de 10 octets
12     mov    DWORD PTR [eax], ecx
13     mov    edx, DWORD PTR _tmp$[ebp+4]
14     mov    DWORD PTR [eax+4], edx
15     mov    cx, WORD PTR _tmp$[ebp+8]
16     mov    WORD PTR [eax+8], cx
17     call    _f
18     add    esp, 12
19     xor    eax, eax
20     mov    esp, ebp
21     pop    ebp
22     ret    0
23 _main    ENDP
24
25 _TEXT    SEGMENT
26 _s$ = 8 ; size = 10
27 ?f@@YAXUs@@@Z PROC    ; f
28     push   ebp
29     mov    ebp, esp
30     mov    eax, DWORD PTR _s$[ebp+6]
31     push   eax
32     movsx  ecx, BYTE PTR _s$[ebp+5]
33     push   ecx
34     mov    edx, DWORD PTR _s$[ebp+1]
35     push   edx
36     movsx  eax, BYTE PTR _s$[ebp]
37     push   eax
38     push   OFFSET $SG3842
39     call   _printf
40     add    esp, 20
41     pop    ebp
42     ret    0
43 ?f@@YAXUs@@@Z ENDP    ; f
```

La structure n'occupe plus que 10 octets et chaque valeur de type *char* n'occupe plus qu'un octet. Quelles sont les conséquences ? Nous économisons de la place au prix d'un accès à ces champs moins rapide que

ne pourrait le faire la CPU.

La structure est également copiée dans `main()`. Cette opération ne s'effectue pas champ par champ mais par blocs en utilisant trois instructions `MOV`. Et pourquoi pas 4 ?

Tout simplement parce que le compilateur a décidé qu'il était préférable d'effectuer la copie en utilisant 3 paires d'instructions `MOV` plutôt que de copier deux mots de 32 bits puis 2 fois un octet ce qui aurait nécessité 4 paires d'instructions `MOV`.

Ce type d'implémentation de la copie qui repose sur les instructions `MOV` plutôt que sur l'appel à la fonction `memcpy()` est très répandu. La raison en est que pour de petits blocs, cette approche est plus rapide qu'un appel à `memcpy()` : 3.11.1 on page 516.

Comme vous pouvez le deviner, si la structure est utilisée dans de nombreux fichiers sources et objets, ils doivent tous être compilés avec la même convention de compactage de la structure.

Au delà de l'option MSVC `/Zp` qui permet de définir l'alignement des champs des structures, il existe également l'option du compilateur `#pragma pack` qui peut être utilisée directement dans le code source. Elle est supportée aussi bien par MSVC[167] que pars GCC[168].

Revenons à la structure `SYSTEMTIME` qui contient des champs de 16 bits. Comment notre compilateur sait-il les aligner sur des frontières de 1 octet ?

Le fichier `WinNT.h` contient ces instructions:

Listing 1.341: WinNT.h

```
#include "pshpack1.h"
```

et celles-ci:

Listing 1.342: WinNT.h

```
#include "pshpack4.h"                    // L'alignement sur 4 octets est la valeur par défaut
```

Le fichier PshPack1.h ressemble à ceci:

Listing 1.343: PshPack1.h

```
#if ! (defined(lint) || defined(RC_INVOKED))
#if ( _MSC_VER >= 800 && !defined(_M_I86)) || defined(_PUSHPOP_SUPPORTED)
#pragma warning(disable :4103)
#if !(defined( MIDL_PASS )) || defined( __midl )
#pragma pack(push,1)
#else
#pragma pack(1)
#endif
#else
#pragma pack(1)
#endif
#endif /* ! (defined(lint) || defined(RC_INVOKED)) */
```

Ces instructions indiquent au compilateur comment compresser les structures définies après `#pragma pack`.

[167] MSDN: Working with Packing Structures
[168] Structure-Packing Pragmas

OllyDbg et les champs alignés par défaut

Examinons dans OllyDbg notre exemple lorsque les champs sont alignés par défaut sur des frontières de 4 octets:

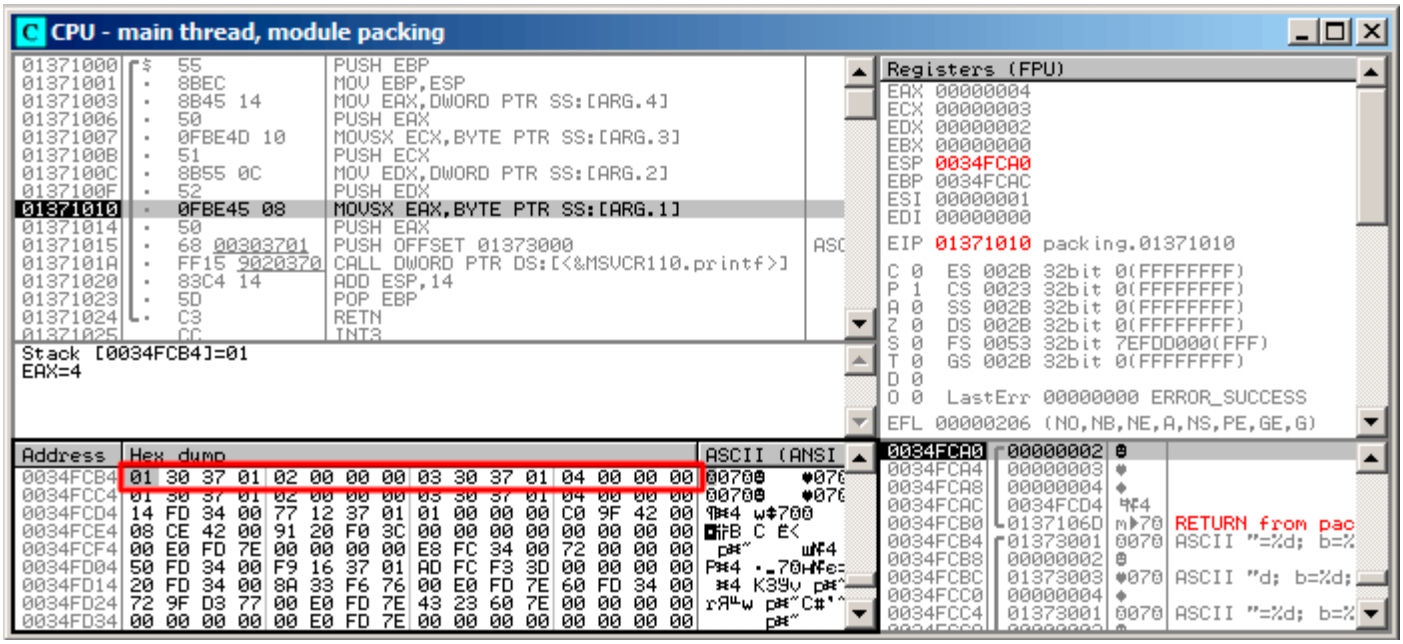


Fig. 1.105: OllyDbg : Before `printf()` execution

Nous voyons nos quatre champs dans la fenêtre de données.

Mais d'où viennent ces octets aléatoires (0x30, 0x37, 0x01) situé à côté des premier (a) et troisième (c) champs ?

Si nous revenons à notre listing 1.339 on page 362, nous constatons que ces deux champs sont de type *char*. Seul un octet est écrit pour chacun d'eux: 1 et 3 respectivement (lignes 6 et 8).

Les trois autres octets des deux mots de 32 bits ne sont pas modifiés en mémoire! Des débris aléatoires des précédentes opérations demeurent donc là.

Ces débris n'influencent nullement le résultat de la fonction `printf()` parce que les valeurs qui lui sont passées sont préparés avec l'instruction `MOVSX` qui opère sur des octets et non pas sur des mots: liste.1.339 (lignes 34 et 38).

L'instruction `MOVSX` (extension de signe) est utilisée ici car le type *char* est par défaut une valeur signée pour MSVC et GCC. Si l'un des types `unsigned char` ou `uint8_t` était utilisé ici, ce serait l'instruction `MOVZX` que le compilateur aurait choisi.

OllyDbg et les champs alignés sur des frontières de 1 octet

Les choses sont beaucoup plus simples ici. Les 4 champs occupent 10 octets et les valeurs sont stockées côte-à-côte.

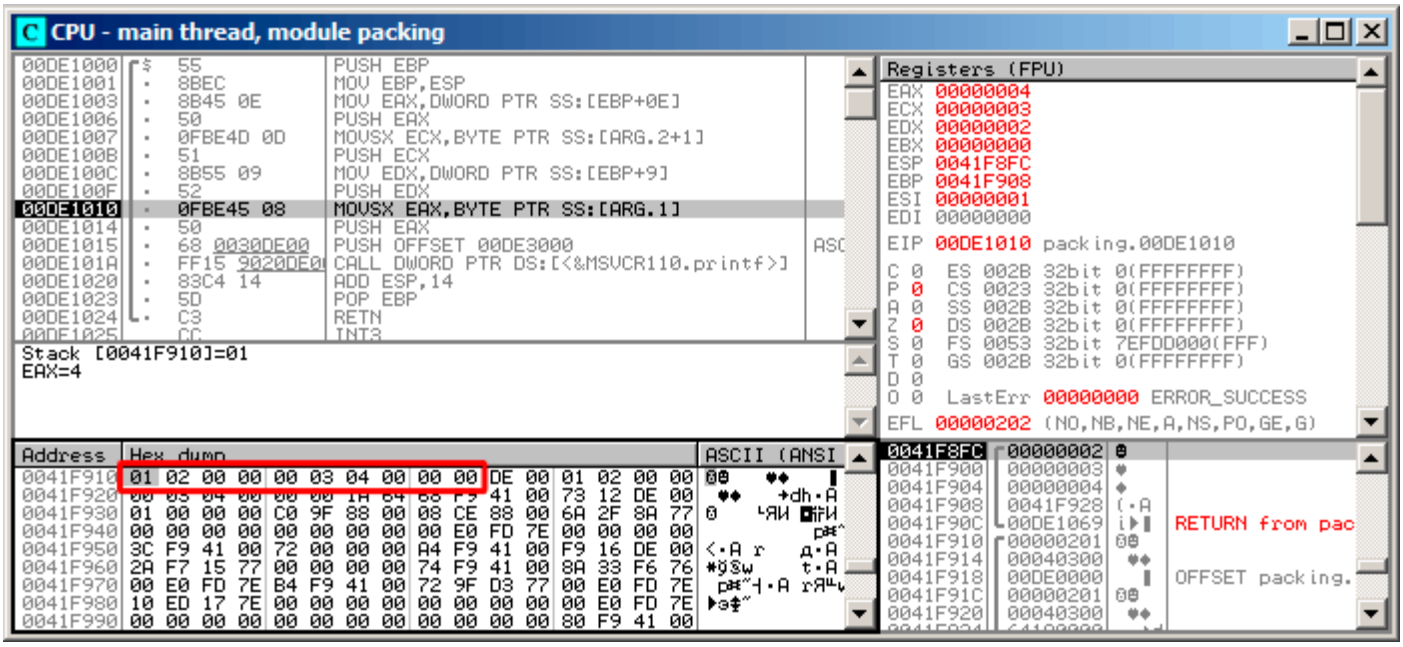


Fig. 1.106: OllyDbg : Avant appel de la fonction `printf()`

ARM

avec optimisation Keil 6/2013 (Mode Thumb)

Listing 1.344: avec optimisation Keil 6/2013 (Mode Thumb)

```
.text :0000003E             exit ; CODE XREF : f+16
.text :0000003E 05 B0               ADD     SP, SP, #0x14
.text :00000040 00 BD               POP     {PC}


.text :00000280             f
.text :00000280
.text :00000280             var_18 = -0x18
.text :00000280             a      = -0x14
.text :00000280             b      = -0x10
.text :00000280             c      = -0xC
.text :00000280             d      = -8
.text :00000280
.text :00000280 0F B5               PUSH    {R0-R3,LR}
.text :00000282 81 B0               SUB     SP, SP, #4
.text :00000284 04 98               LDR     R0, [SP,#16]    ; d
.text :00000286 02 9A               LDR     R2, [SP,#8]     ; b
.text :00000288 00 90               STR     R0, [SP]
.text :0000028A 68 46               MOV     R0, SP
.text :0000028C 03 7B               LDRB    R3, [R0,#12]    ; c
.text :0000028E 01 79               LDRB    R1, [R0,#4]     ; a
.text :00000290 59 A0               ADR     R0, aADBDCDDD   ; "a=%d; b=%d; c=%d; d=%d\n"
.text :00000292 05 F0 AD FF         BL      __2printf
.text :00000296 D2 E6               B       exit
```

Rappelons-nous que c'est une structure qui est passée ici et non pas un pointeur vers une structure. Comme les 4 premiers arguments d'une fonction sont passés dans les registres sur les processeurs ARM, les champs de la structure sont passés dans les registres `R0-R3`.

LDRB charge un octet présent en mémoire et l'étend sur 32bits en prenant en compte son signe. Cette opération est similaire à celle effectuée par MOVSX dans les architectures x86. Elle est utilisée ici pour charger les champs a et c de la structure.

Un autre détail que nous remarquons aisément est que la fonction ne s'achève pas sur un épilogue qui lui est propre. A la place, il y a un saut vers l'épilogue d'une autre fonction! Qui plus est celui d'une fonction très différente sans aucun lien avec la nôtre. Cependant elle possède exactement le même épilogue, probablement parce qu'elle accepte utilise elle aussi 5 variables locales ($5 * 4 = 0x14$).

De plus elle est située à une adresse proche.

En réalité, peut importe l'épilogue qui est utilisé du moment que le fonctionnement est celui attendu.

Il semble donc que le compilateur Keil décide de réutiliser à des fins d'économie un fragment d'une autre fonction. Notre épilogue aurait nécessité 4 octets. L'instruction de saut n'en utilise que 2.

ARM + avec optimisation Xcode 4.6.3 (LLVM) (Mode Thumb-2)

Listing 1.345: avec optimisation Xcode 4.6.3 (LLVM) (Mode Thumb-2)

```
var_C = -0xC

      PUSH   {R7,LR}
      MOV    R7, SP
      SUB    SP, SP, #4
      MOV    R9, R1 ; b
      MOV    R1, R0 ; a
      MOVW   R0, #0xF10 ; "a=%d; b=%d; c=%d; d=%d\n"
      SXTB   R1, R1  ; prepare a
      MOVT.W R0, #0
      STR    R3, [SP,#0xC+var_C] ; place d to stack for printf()
      ADD    R0, PC  ; format-string
      SXTB   R3, R2  ; prepare c
      MOV    R2, R9  ; b
      BLX    _printf
      ADD    SP, SP, #4
      POP    {R7,PC}
```

SXTB (*Signed Extend Byte*) est similaire à MOVSX pour les architectures x86. Pour le reste—c'est identique.

MIPS

Listing 1.346: avec optimisation GCC 4.4.5 (IDA)

```
1  f :
2
3  var_18          = -0x18
4  var_10          = -0x10
5  var_4           = -4
6  arg_0           =  0
7  arg_4           =  4
8  arg_8           =  8
9  arg_C           =  0xC
10
11 ; $a0=s.a
12 ; $a1=s.b
13 ; $a2=s.c
14 ; $a3=s.d
15                 lui     $gp, (__gnu_local_gp >> 16)
16                 addiu   $sp, -0x28
17                 la      $gp, (__gnu_local_gp & 0xFFFF)
18                 sw      $ra, 0x28+var_4($sp)
19                 sw      $gp, 0x28+var_10($sp)
20 ; Transformation d'un octet en entier 32 bits grand-boutien (big-endian) :
21                 sra     $t0, $a0, 24
22                 move    $v1, $a1
23 ; Transformation d'un entier 32 bits grand-boutien (big-endian) en octet :
24                 sra     $v0, $a2, 24
```

```
25                lw      $t9, (printf & 0xFFFF)($gp)
26                sw      $a0, 0x28+arg_0($sp)
27                lui     $a0, ($LC0 >> 16)  # "a=%d ; b=%d ; c=%d ; d=%d\n"
28                sw      $a3, 0x28+var_18($sp)
29                sw      $a1, 0x28+arg_4($sp)
30                sw      $a2, 0x28+arg_8($sp)
31                sw      $a3, 0x28+arg_C($sp)
32                la      $a0, ($LC0 & 0xFFFF)  # "a=%d ; b=%d ; c=%d ; d=%d\n"
33                move    $a1, $t0
34                move    $a2, $v1
35                jalr    $t9
36                move    $a3, $v0 ; Gaspillage volontaire du délai de branchement
37                lw      $ra, 0x28+var_4($sp)
38                or      $at, $zero ; Gaspillage par NOP du délai de chargement
39                jr      $ra
40                addiu   $sp, 0x28 ; Gaspillage volontaire du délai de branchement
41
42 $LC0 :           .ascii "a=%d ; b=%d ; c=%d ; d=%d\n"<0>
```

Les champs de la structure sont fournis dans les registres $A0..$A3 puis transformé dans les registres $A1..$A3 pour l'utilisation par `printf()`, tandis que le 4ème champ (provenant de $A3) est passé sur la pile en utilisant l'instruction `SW`.

Mais à quoi servent ces deux instructions SRA (« Shift Word Right Arithmetic ») lors de la préparation des champs *char* ?

MIPS est une architecture grand-boutien (big-endian) par défaut 2.8 on page 468, de même que la distribution Debian Linux que nous utilisons.

En conséquence, lorsqu'un octet est stocké dans un emplacement 32bits d'une structure, ils occupent les bits 31..24 bits.

Quand une variable *char* doit être étendue en une valeur sur 32 bits, elle doit tout d'abord être décalée vers la droite de 24 bits.

char étant un type signé, un décalage arithmétique est utilisé ici, à la place d'un décalage logique.

Un dernier mot

Passer une structure comme argument d'une fonction (plutôt que de passer un pointeur sur cette structure) revient à passer chaque champ de la structure individuellement.

Si les champs de la structure utilisent l'alignement par défaut, la fonction f() peut être réécrite ainsi:

```
void f(char a, int b, char c, int d)
{
    printf ("a=%d ; b=%d ; c=%d ; d=%d\n", a, b, c, d);
};
```

Le code généré par le compilateur sera le même.

1.24.5 Structures imbriquées

Maintenant qu'en est-il lorsqu'une structure est définie au sein d'une autre structure ?

```
#include <stdio.h>

struct inner_struct
{
    int a;
    int b;
};

struct outer_struct
{
    char a;
    int b;
    struct inner_struct c;
```

```
    char d;
    int e;
};

void f(struct outer_struct s)
{
    printf ("a=%d; b=%d; c.a=%d; c.b=%d; d=%d; e=%d\n",
        s.a, s.b, s.c.a, s.c.b, s.d, s.e);
};

int main()
{
    struct outer_struct s;
    s.a=1;
    s.b=2;
    s.c.a=100;
    s.c.b=101;
    s.d=3;
    s.e=4;
    f(s);
};
```

...dans ce cas, l'ensemble des champs de `inner_struct` doivent être situés entre les champs a,b et d,e de `outer_struct`.

Compilons (MSVC 2010):

Listing 1.347: avec optimisation MSVC 2010 /Ob0

```
$SG2802 DB    'a=%d; b=%d; c.a=%d; c.b=%d; d=%d; e=%d', 0aH, 00H

_TEXT    SEGMENT
_s$ = 8
_f    PROC
    mov    eax, DWORD PTR _s$[esp+16]
    movsx  ecx, BYTE PTR _s$[esp+12]
    mov    edx, DWORD PTR _s$[esp+8]
    push   eax
    mov    eax, DWORD PTR _s$[esp+8]
    push   ecx
    mov    ecx, DWORD PTR _s$[esp+8]
    push   edx
    movsx  edx, BYTE PTR _s$[esp+8]
    push   eax
    push   ecx
    push   edx
    push   OFFSET $SG2802 ; 'a=%d; b=%d; c.a=%d; c.b=%d; d=%d; e=%d'
    call   _printf
    add    esp, 28
    ret    0
_f    ENDP

_s$ = -24
_main    PROC
    sub    esp, 24
    push   ebx
    push   esi
    push   edi
    mov    ecx, 2
    sub    esp, 24
    mov    eax, esp
; depuis ce moment, EAX est synonyme de ESP :
    mov    BYTE PTR _s$[esp+60], 1
    mov    ebx, DWORD PTR _s$[esp+60]
    mov    DWORD PTR [eax], ebx
    mov    DWORD PTR [eax+4], ecx
    lea    edx, DWORD PTR [ecx+98]
    lea    esi, DWORD PTR [ecx+99]
    lea    edi, DWORD PTR [ecx+2]
    mov    DWORD PTR [eax+8], edx
    mov    BYTE PTR _s$[esp+76], 3
```

```
    mov    ecx, DWORD PTR _s$[esp+76]
    mov    DWORD PTR [eax+12], esi
    mov    DWORD PTR [eax+16], ecx
    mov    DWORD PTR [eax+20], edi
    call   _f
    add    esp, 24
    pop    edi
    pop    esi
    xor    eax, eax
    pop    ebx
    add    esp, 24
    ret    0
_main    ENDP
```

Un point troublant est qu'en observant le code assembleur généré, nous n'avons aucun indice qui laisse penser qu'il existe une structure imbriquée! Nous pouvons donc dire que les structures imbriquées sont fusionnées avec leur conteneur pour former une seule structure *linear* ou *one-dimensional*.

Bien entendu, si nous remplaçons la déclaration `struct inner_struct c;` par `struct inner_struct *c;` (en introduisant donc un pointeur) la situation sera totalement différente.

OllyDbg

Chargeons notre exemple dans OllyDbg et observons `outer_struct` en mémoire:

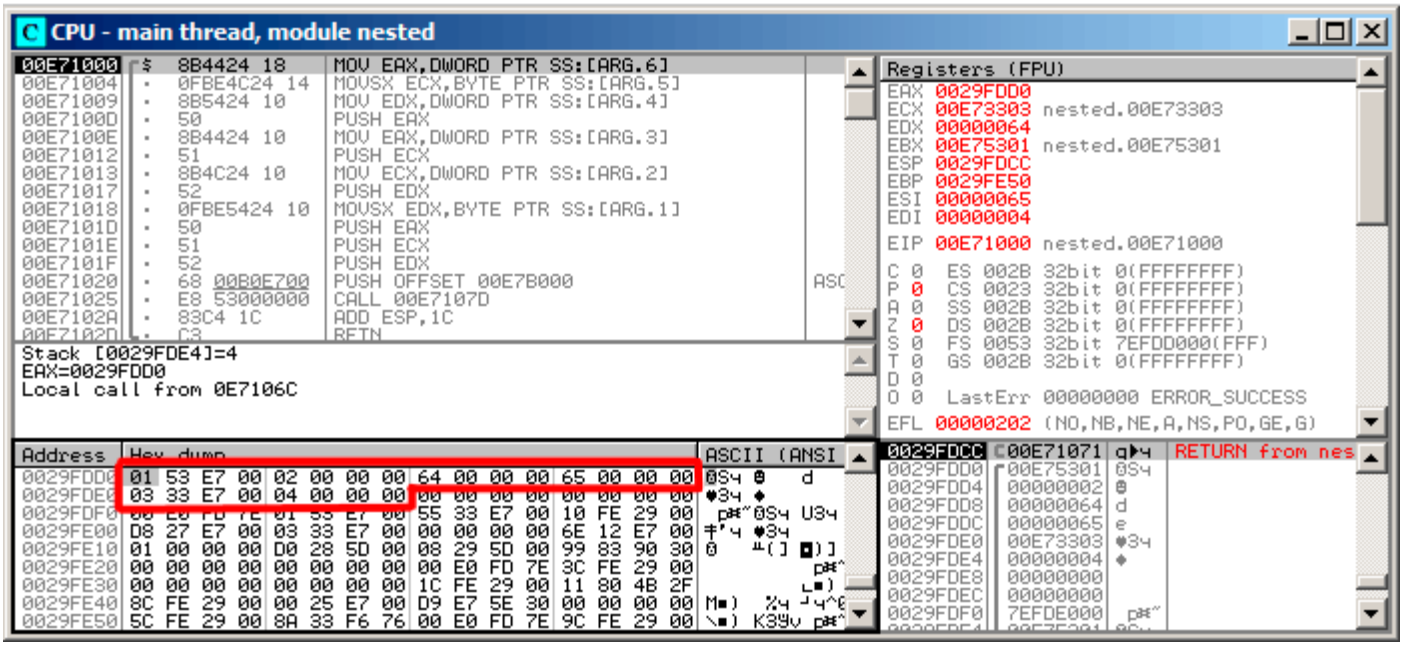


Fig. 1.107: OllyDbg : Avant appel de la fonction `printf()`

Les valeurs sont organisées en mémoire de la manière suivante:

- *(outer_struct.a)* (octet) 1 + 3 octets de détritus;
- *(outer_struct.b)* (mot de 32 bits) 2;
- *(inner_struct.a)* (mot de 32 bits) 0x64 (100);
- *(inner_struct.b)* (mot de 32 bits) 0x65 (101);
- *(outer_struct.d)* (octet) 3 + 3 octets de détritus;
- *(outer_struct.e)* (mot de 32 bits) 4.

1.24.6 Champs de bits dans une structure

Exemple CPUID

Le langage C/C++ permet de définir précisément le nombre de bits occupés par chaque champ d'une structure. Ceci est très utile lorsque l'on cherche à économise de la place. Par exemple, chaque bit permet de représenter une variable *bool*. Bien entendu, c'est au détriment de la vitesse d'exécution.

Prenons par exemple l'instruction `CPUID`[169]. Elle retourne des informations au sujet de la CPU qui exécute le programme et de ses capacités.

Si le registre EAX est positionné à la valeur 1 avant d'invoquer cette instruction, `CPUID` va retourné les informations suivantes dans le registre EAX :

3:0 (4 bits)	Stepping
7:4 (4 bits)	Modèle
11:8 (4 bits)	Famille
13:12 (2 bits)	Type de processeur
19:16 (4 bits)	Sous-modèle
27:20 (8 bits)	Sous-famille

MSVC 2010 fourni une macro `CPUID`, qui est absente de GCC 4.4.1. Tentons donc de rédiger nous même cette fonction pour une utilisation dans GCC grâce à l'assembleur[170]intégré à ce compilateur.

[169] Wikipédia
[170] Complément sur le fonctionnement interne de l'assembleur GCC

```
#include <stdio.h>

#ifdef __GNUC__
static inline void cpuid(int code, int *a, int *b, int *c, int *d) {
  asm volatile("cpuid":"=a"(*a),"=b"(*b),"=c"(*c),"=d"(*d) :"a"(code));
}
#endif

#ifdef _MSC_VER
#include <intrin.h>
#endif

struct CPUID_1_EAX
{
    unsigned int stepping :4;
    unsigned int model :4;
    unsigned int family_id :4;
    unsigned int processor_type :2;
    unsigned int reserved1 :2;
    unsigned int extended_model_id :4;
    unsigned int extended_family_id :8;
    unsigned int reserved2 :4;
};

int main()
{
    struct CPUID_1_EAX *tmp;
    int b[4];

#ifdef _MSC_VER
    __cpuid(b,1);
#endif

#ifdef __GNUC__
    cpuid (1, &b[0], &b[1], &b[2], &b[3]);
#endif

    tmp=(struct CPUID_1_EAX *)&b[0];

    printf ("stepping=%d\n", tmp->stepping);
    printf ("model=%d\n", tmp->model);
    printf ("family_id=%d\n", tmp->family_id);
    printf ("processor_type=%d\n", tmp->processor_type);
    printf ("extended_model_id=%d\n", tmp->extended_model_id);
    printf ("extended_family_id=%d\n", tmp->extended_family_id);

    return 0;
};
```

Après que l'instruction CPUID ait rempli les registres EAX/EBX/ECX/EDX, ceux-ci doivent être recopiés dans le tableau b[]. Nous affectons dont le pointeur de structure CPUID_1_EAX pour qu'il contienne l'adresse du tableau b[].

En d'autres termes, nous traitons une valeur *int* comme une structure, puis nous lisons des bits spécifiques de la structure.

MSVC

Compilons notre exemple avec MSVC 2008 en utilisant l'option /0x :

Listing 1.348: avec optimisation MSVC 2008

```
_b$ = -16  ; size = 16
_main    PROC
    sub    esp, 16
    push   ebx

    xor    ecx, ecx
```

```
    mov    eax, 1
    cpuid
    push   esi
    lea    esi, DWORD PTR _b$[esp+24]
    mov    DWORD PTR [esi], eax
    mov    DWORD PTR [esi+4], ebx
    mov    DWORD PTR [esi+8], ecx
    mov    DWORD PTR [esi+12], edx

    mov    esi, DWORD PTR _b$[esp+24]
    mov    eax, esi
    and    eax, 15
    push   eax
    push   OFFSET $SG15435 ; 'stepping=%d', 0aH, 00H
    call   _printf

    mov    ecx, esi
    shr    ecx, 4
    and    ecx, 15
    push   ecx
    push   OFFSET $SG15436 ; 'model=%d', 0aH, 00H
    call   _printf

    mov    edx, esi
    shr    edx, 8
    and    edx, 15
    push   edx
    push   OFFSET $SG15437 ; 'family_id=%d', 0aH, 00H
    call   _printf

    mov    eax, esi
    shr    eax, 12
    and    eax, 3
    push   eax
    push   OFFSET $SG15438 ; 'processor_type=%d', 0aH, 00H
    call   _printf

    mov    ecx, esi
    shr    ecx, 16
    and    ecx, 15
    push   ecx
    push   OFFSET $SG15439 ; 'extended_model_id=%d', 0aH, 00H
    call   _printf

    shr    esi, 20
    and    esi, 255
    push   esi
    push   OFFSET $SG15440 ; 'extended_family_id=%d', 0aH, 00H
    call   _printf
    add    esp, 48
    pop    esi

    xor    eax, eax
    pop    ebx

    add    esp, 16
    ret    0
_main    ENDP
```

L'instruction SHR va décaler la valeur du registre EAX d'un certain nombre de bits qui vont être abandonnées. Nous ignorons donc certains des bits de la partie droite.

L'instruction AND "efface" les bits inutiles sur la gauche, ou en d'autres termes, ne laisse dans le registre EAX que les bits qui nous intéressent.

MSVC + OllyDbg

Chargeons notre exemple dans OllyDbg et voyons quelles valeurs sont présentes dans EAX/EBX/ECX/EDX après exécution de l'instruction CPUID:

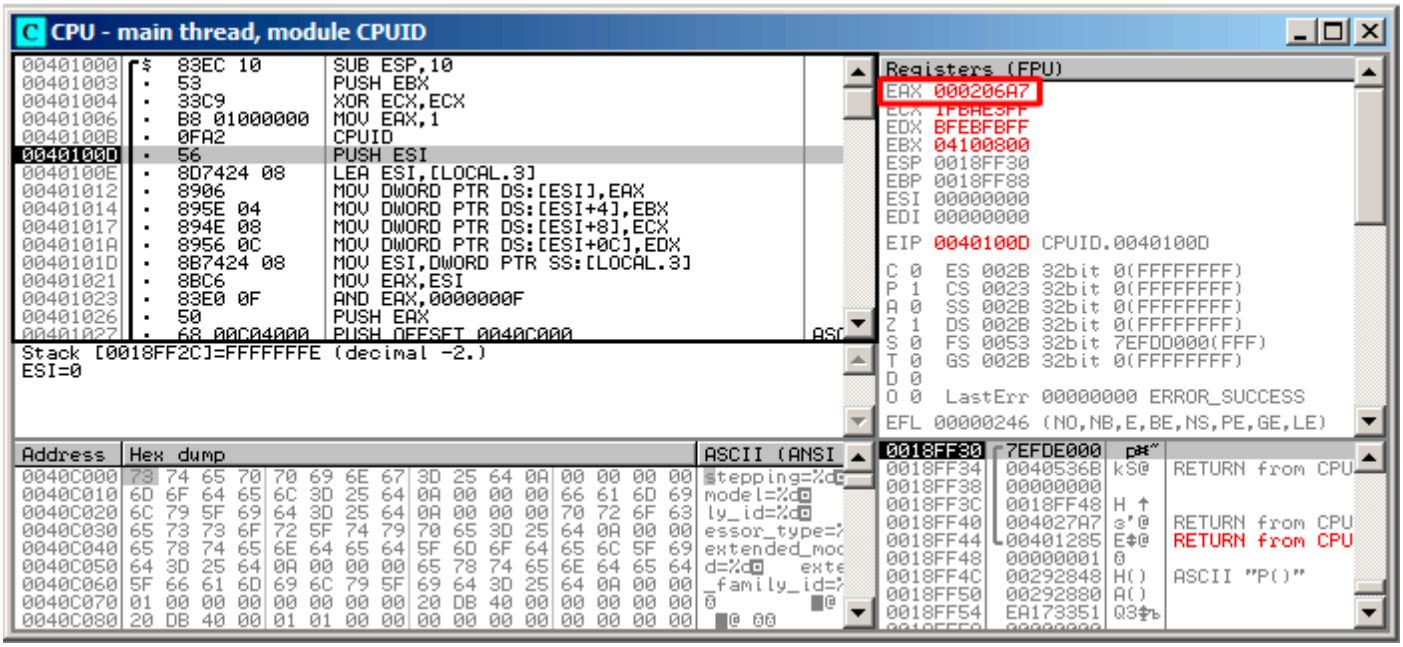


Fig. 1.108: OllyDbg : Après exécution de CPUID

La valeur de EAX est `0x000206A7` (ma CPU est un Intel Xeon E3-1220).
Cette valeur exprimée en binaire vaut $0b00000000000000100000011010100111$.

Voici la manière dont les bits sont répartis sur les différents champs:

champ	format binaire	format décimal
reserved2	0000	0
extended_family_id	00000000	0
extended_model_id	0010	2
reserved1	00	0
processor_id	00	0
family_id	0110	6
model	1010	10
stepping	0111	7

Listing 1.349: Console output

```
stepping=7
model=10
family_id=6
processor_type=0
extended_model_id=2
extended_family_id=0
```

GCC

Essayons maintenant une compilation avec GCC 4.4.1 en utilisant l'option `-03`.

Listing 1.350: avec optimisation GCC 4.4.1

```
main            proc near  ; DATA XREF : _start+17
    push    ebp
    mov     ebp, esp
    and     esp, 0FFFFFFF0h
    push    esi
```

```
    mov     esi, 1
    push    ebx
    mov     eax, esi
    sub     esp, 18h
    cpuid
    mov     esi, eax
    and     eax, 0Fh
    mov     [esp+8], eax
    mov     dword ptr [esp+4], offset aSteppingD ; "stepping=%d\n"
    mov     dword ptr [esp], 1
    call    ___printf_chk
    mov     eax, esi
    shr     eax, 4
    and     eax, 0Fh
    mov     [esp+8], eax
    mov     dword ptr [esp+4], offset aModelD ; "model=%d\n"
    mov     dword ptr [esp], 1
    call    ___printf_chk
    mov     eax, esi
    shr     eax, 8
    and     eax, 0Fh
    mov     [esp+8], eax
    mov     dword ptr [esp+4], offset aFamily_idD ; "family_id=%d\n"
    mov     dword ptr [esp], 1
    call    ___printf_chk
    mov     eax, esi
    shr     eax, 0Ch
    and     eax, 3
    mov     [esp+8], eax
    mov     dword ptr [esp+4], offset aProcessor_type ; "processor_type=%d\n"
    mov     dword ptr [esp], 1
    call    ___printf_chk
    mov     eax, esi
    shr     eax, 10h
    shr     esi, 14h
    and     eax, 0Fh
    and     esi, 0FFh
    mov     [esp+8], eax
    mov     dword ptr [esp+4], offset aExtended_model ; "extended_model_id=%d\n"
    mov     dword ptr [esp], 1
    call    ___printf_chk
    mov     [esp+8], esi
    mov     dword ptr [esp+4], offset unk_80486D0
    mov     dword ptr [esp], 1
    call    ___printf_chk
    add     esp, 18h
    xor     eax, eax
    pop     ebx
    pop     esi
    mov     esp, ebp
    pop     ebp
    retn
main            endp
```

Le résultat est quasiment identique. Le seul élément notable est que GCC combine en quelques sortes le calcul de `extended_model_id` et `extended_family_id` en un seul bloc au lieu de les calculer séparément avant chaque appel à `printf()`.

Travailler avec le type float comme une structure

Comme nous l'avons expliqué dans la section traitant de la FPU (1.19 on page 218), les types *float* et *double* sont constitués d'un *signe*, d'un *significande* (ou *fraction*) et d'un *exposant*. Mais serions nous capable de travailler avec chacun de ces champs indépendamment? Essayons avec un *float*.

(Ssigne)

```
#include <stdio.h>
#include <assert.h>
#include <stdlib.h>
#include <memory.h>

struct float_as_struct
{
    unsigned int fraction  : 23; // fraction
    unsigned int exponent  : 8;  // exposant + 0x3FF
    unsigned int sign  : 1;      // bit de signe
};

float f(float _in)
{
    float f=_in;
    struct float_as_struct t;

    assert (sizeof (struct float_as_struct) == sizeof (float));

    memcpy (&t, &f, sizeof (float));

    t.sign=1; // Positionnons le bit de signe
    t.exponent=t.exponent+2; // multiplions d par 2^n(n vaut 2 ici)

    memcpy (&f, &t, sizeof (float));

    return f;
};

int main()
{
    printf ("%f\n", f(1.234));
};
```

La structure `float_as_struct` occupe le même espace qu'un *float*, soit 4 octets ou 32 bits.

Nous positionnons maintenant le signe pour qu'il soit négatif puis en ajoutant à la valeur de l'exposant, ce qui fait que nous multiplions le nombre par 2^2, soit 4.

Compilons notre exemple avec MSVC 2008, sans optimisation:

Listing 1.351: MSVC 2008 sans optimisation

```
_t$ = -8   ; size = 4
_f$ = -4   ; size = 4
__in$ = 8  ; size = 4
?f@@YAMM@Z PROC ; f
    push   ebp
    mov    ebp, esp
    sub    esp, 8

    fld    DWORD PTR __in$[ebp]
    fstp   DWORD PTR _f$[ebp]

    push   4
    lea    eax, DWORD PTR _f$[ebp]
    push   eax
    lea    ecx, DWORD PTR _t$[ebp]
    push   ecx
    call   _memcpy
    add    esp, 12

    mov    edx, DWORD PTR _t$[ebp]
    or     edx, -2147483648 ; 80000000H - positionnement du site négatif
    mov    DWORD PTR _t$[ebp], edx

    mov    eax, DWORD PTR _t$[ebp]
    shr    eax, 23          ; 00000017H - suppression du signifiant
    and    eax, 255         ; 000000ffH - nous ne conservons ici que l'exposant
```

```
    add    eax, 2           ; ajouter 2
    and    eax, 255         ; 000000ffH
    shl    eax, 23          ; 00000017H - décalage du résultat pour supprimer les bits 30:23
    mov    ecx, DWORD PTR _t$[ebp]
    and    ecx, -2139095041 ; 807fffffH - suppression de l'exposant

; ajout de la valeur originale de l'exposant avec le nouvel exposant qui vient d'être calculé :
    or     ecx, eax
    mov    DWORD PTR _t$[ebp], ecx

    push   4
    lea    edx, DWORD PTR _t$[ebp]
    push   edx
    lea    eax, DWORD PTR _f$[ebp]
    push   eax
    call   _memcpy
    add    esp, 12

    fld    DWORD PTR _f$[ebp]

    mov    esp, ebp
    pop    ebp
    ret    0
?f@@YAMM@Z ENDP   ; f
```

Si nous avions compilé avec le flag `/Ox` il n'y aurait pas d'appel à la fonction `memcpy()`, et la variable `f` serait utilisée directement. Mais la compréhension est facilitée lorsque l'on s'intéresse à la version non optimisée.

A quoi cela ressemblerait si nous utilisions l'option `-O3` avec le compilateur GCC 4.4.1 ?

Listing 1.352: GCC 4.4.1 avec optimisation

```
; f(float)
       public _Z1ff
_Z1ff  proc near

var_4  = dword ptr -4
arg_0  = dword ptr  8

       push    ebp
       mov     ebp, esp
       sub     esp, 4
       mov     eax, [ebp+arg_0]
       or      eax, 80000000h  ; positionnement du signe négatif
       mov     edx, eax
       and     eax, 807FFFFFh  ; Nous ne conservons que le signe et le signifiant dans EAX
       shr     edx, 23         ; Préparation de l'exposant
       add     edx, 2          ; Ajout de 2
       movzx   edx, dl         ; RAZ de tous les octets dans EAX à l'exception des bits 7:0
       shl     edx, 23         ; Décalage du nouvel exposant pour qu'ils soit à sa place
       or      eax, edx        ; Consolidation du nouvel exposant et de la valeur originale de ↙
    ↳ l'exposant
       mov     [ebp+var_4], eax
       fld     [ebp+var_4]
       leave
       retn
_Z1ff  endp

       public main
main   proc near
       push    ebp
       mov     ebp, esp
       and     esp, 0FFFFFFF0h
       sub     esp, 10h
       fld     ds :dword_8048614 ; -4.936
       fstp    qword ptr [esp+8]
       mov     dword ptr [esp+4], offset asc_8048610 ; "%f\n"
       mov     dword ptr [esp], 1
       call    ___printf_chk
       xor     eax, eax
```

```
        leave
        retn
main    endp
```

La fonction `f()` est à peu près compréhensible. Par contre ce qui est intéressant c'est que GCC a été capable de calculer le résultat de `f(1.234)` durant la compilation malgré tous les triturages des champs de la structure et a directement préparé l'argument passé à `printf()` durant la compilation!

1.24.7 Exercices

- http://challenges.re/71
- http://challenges.re/72

1.25 Unions

Les *unions* en C/C++ sont utilisées principalement pour interpréter une variable (ou un bloc de mémoire) d'un type de données comme une variable d'un autre type de données.

1.25.1 Exemple de générateur de nombres pseudo-aléatoires

Si nous avons besoin de nombres aléatoires à virgule flottante entre 0 et 1, le plus simple est d'utiliser un PRNG[171] comme le Twister de Mersenne. Il produit une valeur aléatoire non signée sur 32-bit (en d'autres mots, il produit une valeur 32-bit aléatoire). Puis, nous pouvons transformer cette valeur en *float* et le diviser par `RAND_MAX` (`0xFFFFFFFF` dans notre cas)—nous obtenons une valeur dans l'intervalle 0..1.

Mais nous savons que la division est lente. Aussi, nous aimerions utiliser le moins d'opérations FPU possible. Peut-on se passer de la division?

Rappelons-nous en quoi consiste un nombre en virgule flottante: un bit de signe, un significande et un exposant. Nous n'avons qu'à stocker des bits aléatoires dans toute le significande pour obtenir un nombre réel aléatoire!

L'exposant ne peut pas être zéro (le nombre flottant est dénormalisé dans ce cas), donc nous stockons 0b01111111 dans l'exposant—ce qui signifie que l'exposant est 1. Ensuite nous remplissons le significande avec des bits aléatoires, mettons le signe à 0 (ce qui indique un nombre positif) et voilà. Les nombres générés sont entre 1 et 2, donc nous devons soustraire 1.

Un générateur congruentiel linéaire de nombres aléatoire très simple est utilisé dans mon exemple[172], il produit des nombres 32-bit. Le PRNG est initialisé avec le temps courant au format UNIX timestamp.

Ici, nous représentons un type *float* comme une *union*—c'est la construction C/C++ qui nous permet d'interpréter un bloc de mémoire sous différents types. Dans notre cas, nous pouvons créer une variable de type *union* et y accéder comme si c'est un *float* ou un *uint32_t*. On peut dire que c'est juste un hack. Un sale.

Le code du PRNG entier est le même que celui que nous avons déjà considéré: 1.23 on page 340. Donc la forme compilée du code est omise.

```
#include <stdio.h>
#include <stdint.h>
#include <time.h>

// PRNG entier définitions, données et routines :

// constantes provenant du livre Numerical Recipes
const uint32_t RNG_a=1664525;
const uint32_t RNG_c=1013904223;
uint32_t RNG_state; // variable globale

void my_srand(uint32_t i)
{
```

[171] Nombre généré pseudo-aléatoirement
[172] l'idée a été prise de: http://go.yurichev.com/17308

```
        RNG_state=i;
};

uint32_t my_rand()
{
        RNG_state=RNG_state*RNG_a+RNG_c;
        return RNG_state;
};

// PRNG FPU définitions et routines :

union uint32_t_float
{
        uint32_t i;
        float f;
};

float float_rand()
{
        union uint32_t_float tmp;
        tmp.i=my_rand() & 0x007fffff | 0x3F800000;
        return tmp.f-1;
};

// test

int main()
{
        my_srand(time(NULL)); // initialisation du PRNG

        for (int i=0; i<100; i++)
                printf ("%f\n", float_rand());

        return 0;
};
```

x86

Listing 1.353: MSVC 2010 avec optimisation

```
$SG4238 DB      '%f', 0aH, 00H

__real@3ff0000000000000 DQ 03ff0000000000000r   ; 1

tv130 = -4
_tmp$ = -4
?float_rand@@YAMXZ PROC
        push    ecx
        call    ?my_rand@@YAIXZ
; EAX=valeur pseudo-aléatoire
        and     eax, 8388607            ; 007fffffH
        or      eax, 1065353216         ; 3f800000H
; EAX=valeur pseudo-aléatoire & 0x007fffff | 0x3f800000
; la stocker dans la pile locale :
        mov     DWORD PTR _tmp$[esp+4], eax
; la recharger en tant que nombre à virgule flottante :
        fld     DWORD PTR _tmp$[esp+4]
; soustraire 1.0:
        fsub    QWORD PTR __real@3ff0000000000000
; stocker la valeur obtenue dans la pile locale et la recharger :
        fstp    DWORD PTR tv130[esp+4] ; \  ces instructions sont redondantes
        fld     DWORD PTR tv130[esp+4] ; /
        pop     ecx
        ret     0
?float_rand@@YAMXZ ENDP

_main   PROC
        push    esi
```

```
        xor     eax, eax
        call    _time
        push    eax
        call    ?my_srand@@YAXI@Z
        add     esp, 4
        mov     esi, 100
$LL3@main :
        call    ?float_rand@@YAMXZ
        sub     esp, 8
        fstp    QWORD PTR [esp]
        push    OFFSET $SG4238
        call    _printf
        add     esp, 12
        dec     esi
        jne     SHORT $LL3@main
        xor     eax, eax
        pop     esi
        ret     0
_main   ENDP
```

Les noms de fonctions sont étranges ici car cet exemple a été compilé en tant que C++ et ceci est la modification des noms en C++, nous en parlerons plus loin: **??** on page ??. Si nous compilons ceci avec MSVC 2012, il utilise des instructions SIMD pour le FPU, pour en savoir plus: 1.31.5 on page 441.

MIPS

Listing 1.354: GCC 4.4.5 avec optimisation

```
float_rand :

var_10          = -0x10
var_4           = -4

                lui     $gp, (__gnu_local_gp >> 16)
                addiu   $sp, -0x20
                la      $gp, (__gnu_local_gp & 0xFFFF)
                sw      $ra, 0x20+var_4($sp)
                sw      $gp, 0x20+var_10($sp)
; appeler my_rand() :
                jal     my_rand
                or      $at, $zero ; slot de délai de branchement, NOP
; $v0=32-bit valeur pseudo-aléatoire
                li      $v1, 0x7FFFFF
; $v1=0x7FFFFF
                and     $v1, $v0, $v1
; $v1=valeur pseudo-aléatoire & 0x7FFFFF
                lui     $a0, 0x3F80
; $a0=0x3F800000
                or      $v1, $a0
; $v1=valeur pseudo-aléatoire & 0x7FFFFF | 0x3F800000
; le cas de l'instruction suivante est encore difficile à comprendre :
                lui     $v0, ($LC0 >> 16)
; charger 1.0 dans $f0 :
                lwc1    $f0, $LC0
; déplacer la valeur de $v1 vers le coprocesseur 1 (dans le registre $f2)
; ça se comporte comme une copie bit à bit, pas de conversion faite :
                mtc1    $v1, $f2
                lw      $ra, 0x20+var_4($sp)
; soustraire 1.0. laisser le résultat dans $f0 :
                sub.s   $f0, $f2, $f0
                jr      $ra
                addiu   $sp, 0x20  ; slot de délai de branchement

main :

var_18          = -0x18
var_10          = -0x10
var_C           = -0xC
```

```
var_8           = -8
var_4           = -4

                lui     $gp, (__gnu_local_gp >> 16)
                addiu   $sp, -0x28
                la      $gp, (__gnu_local_gp & 0xFFFF)
                sw      $ra, 0x28+var_4($sp)
                sw      $s2, 0x28+var_8($sp)
                sw      $s1, 0x28+var_C($sp)
                sw      $s0, 0x28+var_10($sp)
                sw      $gp, 0x28+var_18($sp)
                lw      $t9, (time & 0xFFFF)($gp)
                or      $at, $zero ; slot de délai de chargement, NOP
                jalr    $t9
                move    $a0, $zero ; slot de délai de branchement
                lui     $s2, ($LC1 >> 16)  # "%f\n"
                move    $a0, $v0
                la      $s2, ($LC1 & 0xFFFF)  # "%f\n"
                move    $s0, $zero
                jal     my_srand
                li      $s1, 0x64  # 'd' ; slot de délai de branchement

loc_104 :
                jal     float_rand
                addiu   $s0, 1
                lw      $gp, 0x28+var_18($sp)
; convertir la valeur obtenue de float_rand() en type double (printf() en a besoin) :
                cvt.d.s $f2, $f0
                lw      $t9, (printf & 0xFFFF)($gp)
                mfc1    $a3, $f2
                mfc1    $a2, $f3
                jalr    $t9
                move    $a0, $s2
                bne     $s0, $s1, loc_104
                move    $v0, $zero
                lw      $ra, 0x28+var_4($sp)
                lw      $s2, 0x28+var_8($sp)
                lw      $s1, 0x28+var_C($sp)
                lw      $s0, 0x28+var_10($sp)
                jr      $ra
                addiu   $sp, 0x28 ; slot de délai de branchement

$LC1 :           .ascii "%f\n"<0>
$LC0 :           .float 1.0
```

Il y a aussi une instruction LUI inutile, ajoutée pour quelque étrange raison. Nous avons déjà considéré cet artefact plus tôt: 1.19.5 on page 231.

ARM (Mode ARM)

Listing 1.355: GCC 4.6.3 avec optimisation (IDA)

```
float_rand
                STMFD   SP!, {R3,LR}
                BL      my_rand
; R0=valeur pseudo-aléatoire
                FLDS    S0, =1.0
; S0=1.0
                BIC     R3, R0, #0xFF000000
                BIC     R3, R3, #0x800000
                ORR     R3, R3, #0x3F800000
; R3=valeur pseudo-aléatoire & 0x007fffff | 0x3f800000
; copier de R3 vers FPU (registre S15).
; ça se comporte comme une copie bit à bit, pas de conversion faite :
                FMSR    S15, R3
; soustraire 1.0 et laisser le résultat dans S0 :
                FSUBS   S0, S15, S0
                LDMFD   SP!, {R3,PC}
```

```
flt_5C          DCFS 1.0

main
                STMFD   SP!, {R4,LR}
                MOV     R0, #0
                BL      time
                BL      my_srand
                MOV     R4, #0x64 ; 'd'

loc_78
                BL      float_rand
; S0=valeur pseudo-aléatoire
                LDR     R0, =aF         ; "%f"
; convertir la valeur obtenue en type double (printf() en a besoin) :
                FCVTDS  D7, S0
; copie bit à bit de D7 dans la paire de registres R2/R3 (pour printf()) :
                FMRRD   R2, R3, D7
                BL      printf
                SUBS    R4, R4, #1
                BNE     loc_78
                MOV     R0, R4
                LDMFD   SP!, {R4,PC}

aF              DCB "%f",0xA,0
```

Nous allons faire un dump avec objdump et nous allons voir que les instructions FPU ont un nom différent que dans IDA. Apparemment, les développeurs de IDA et binutils ont utilisés des manuels différents? Peut-être qu'il serait bon de connaître les deux variantes de noms des instructions.

Listing 1.356: GCC 4.6.3 avec optimisation (objdump)

```
00000038 <float_rand> :
  38:   e92d4008        push    {r3, lr}
  3c :   ebffffe        bl      10 <my_rand>
  40:   ed9f0a05        vldr    s0, [pc, #20]    ; 5c <float_rand+0x24>
  44:   e3c034ff        bic     r3, r0, #-16777216       ; 0xff000000
  48:   e3c33502        bic     r3, r3, #8388608         ; 0x800000
  4c :   e38335fe        orr     r3, r3, #1065353216      ; 0x3f800000
  50:   ee073a90        vmov    s15, r3
  54:   ee370ac0        vsub.f32        s0, s15, s0
  58:   e8bd8008        pop     {r3, pc}
  5c :   3f800000        svccc   0x00800000

00000000 <main> :
   0:   e92d4010        push    {r4, lr}
   4:   e3a00000        mov     r0, #0
   8:   ebffffe        bl      0 <time>
   c :   ebffffe        bl      0 <main>
  10:   e3a04064        mov     r4, #100         ; 0x64
  14:   ebffffe        bl      38 <main+0x38>
  18:   e59f0018        ldr     r0, [pc, #24]    ; 38 <main+0x38>
  1c :   eeb77ac0        vcvt.f64.f32    d7, s0
  20:   ec532b17        vmov    r2, r3, d7
  24:   ebffffe        bl      0 <printf>
  28:   e2544001        subs    r4, r4, #1
  2c :   1afffff8        bne     14 <main+0x14>
  30:   e1a00004        mov     r0, r4
  34:   e8bd8010        pop     {r4, pc}
  38:   00000000        andeq   r0, r0, r0
```

Les instructions en 0x5c dans `float_rand()` et en 0x38 dans `main()` sont du bruit (pseudo-)aléatoire.

1.25.2 Calcul de l'epsilon de la machine

L'epsilon de la machine est la plus petite valeur avec laquelle le FPU peut travailler. Plus il y a de bits alloués pour représenter un nombre en virgule flottante, plus l'epsilon est petit, C'est $2^{-23} = 1.19e-07$ pour les *float* et $2^{-52} = 2.22e-16$ pour les *double*. Voir aussi: l'article de Wikipédia.

Il intéressant de voir comme il est facile de calculer l'epsilon de la machine:

```
#include <stdio.h>
#include <stdint.h>

union uint_float
{
        uint32_t i;
        float f;
};

float calculate_machine_epsilon(float start)
{
        union uint_float v;
        v.f=start;
        v.i++;
        return v.f-start;
}

void main()
{
        printf ("%g\n", calculate_machine_epsilon(1.0));
};
```

Ce que l'on fait ici est simplement de traiter la partie fractionnaire du nombre au format IEEE 754 comme un entier et de lui ajouter 1. Le nombre flottant en résultant est égal à $starting\_value+machine\_epsilon$, donc il suffit de soustraire $starting\_value$ (en utilisant l'arithmétique flottante) pour mesurer ce que la différence d'un bit représente dans un nombre flottant simple précision(*float*). L' *union* permet ici d'accéder au nombre IEEE 754 comme à un entier normal. Lui ajouter 1 ajoute en fait 1 au *significande* du nombre, toutefois, inutile de dire, un débordement est possible, qui ajouterait 1 à l'exposant.

x86

Listing 1.357: avec optimisation MSVC 2010

```
tv130 = 8
_v$ = 8
_start$ = 8
_calculate_machine_epsilon PROC
        fld     DWORD PTR _start$[esp-4]
        fst     DWORD PTR _v$[esp-4]     ; cette instruction est redondante
        inc     DWORD PTR _v$[esp-4]
        fsubr   DWORD PTR _v$[esp-4]
        fstp    DWORD PTR tv130[esp-4]   ; \ cette paire d'instructions est aussi redondante
        fld     DWORD PTR tv130[esp-4]   ; /
        ret     0
_calculate_machine_epsilon ENDP
```

La seconde instruction FST est redondante: il n'est pas nécessaire de stocker la valeur en entrée à la même place (le compilateur a décidé d'allouer la variable v à la même place dans la pile locale que l'argument en entrée). Puis elle est incrémentée avec `INC`, puisque c'est une variable entière normale. Ensuite elle est chargée dans le FPU comme un nombre IEEE 754 32-bit, `FSUBR` fait le reste du travail et la valeur résultante est stockée dans `ST0`. La dernière paire d'instructions `FSTP`/`FLD` est redondante, mais le compilateur n'a pas optimisé le code.

ARM64

Étendons notre exemple à 64-bit:

```
#include <stdio.h>
#include <stdint.h>

typedef union
{
        uint64_t i;
        double d;
```

```
} uint_double;

double calculate_machine_epsilon(double start)
{
        uint_double v;
        v.d=start;
        v.i++;
        return v.d-start;
}

void main()
{
        printf ("%g\n", calculate_machine_epsilon(1.0));
};
```

ARM64 n'a pas d'instruction qui peut ajouter un nombre a un D-registre FPU, donc la valeur en entrée (qui provient du registre x64 D0) est d'abord copiée dans le GPR, incrémentée, copiée dans le registre FPU D1, et puis la soustraction est faite.

Listing 1.358: GCC 4.9 ARM64 avec optimisation

```
calculate_machine_epsilon :
        fmov    x0, d0          ; charger la valeur d'entrée de type double dans X0
        add     x0, x0, 1       ; X0++
        fmov    d1, x0          ; la déplacer dans le registre du FPU
        fsub    d0, d1, d0      ; soustraire
        ret
```

Voir aussi cet exemple compilé pour x64 avec instructions SIMD: 1.31.4 on page 440.

MIPS

Il y a ici la nouvelle instruction MTC1 (« Move To Coprocessor 1 »), elle transfère simplement des données vers les registres du FPU.

Listing 1.359: GCC 4.4.5 avec optimisation (IDA)

```
calculate_machine_epsilon :
                mfc1    $v0, $f12
                or      $at, $zero  ; NOP
                addiu   $v1, $v0, 1
                mtc1    $v1, $f2
                jr      $ra
                sub.s   $f0, $f2, $f12  ; branch delay slot
```

Conclusion

Il est difficile de dire si quelqu'un pourrait avoir besoin de cette astuce dans du code réel, mais comme cela a été mentionné plusieurs fois dans ce livre, cet exemple est utile pour expliquer le format IEEE 754 et les *union*s en C/C++.

1.26 Remplacement de FSCALE

Agner Fog dans son travail[173] *Optimizing subroutines in assembly language / An optimization guide for x86 platforms* indique que l'instruction FPU FSCALE (qui calcule 2^n) peut être lente sur de nombreux CPUs, et propose un remplacement plus rapide.

Voici ma conversion de son code assembleur en C/C++ :

[173] http://www.agner.org/optimize/optimizing_assembly.pdf

```
#include <stdint.h>
#include <stdio.h>

union uint_float
{
        uint32_t i;
        float f;
};

float flt_2n(int N)
{
        union uint_float tmp;

        tmp.i=(N<<23)+0x3f800000;
        return tmp.f;
};

struct float_as_struct
{
        unsigned int fraction  : 23;
        unsigned int exponent  : 8;
        unsigned int sign  : 1;
};

float flt_2n_v2(int N)
{
        struct float_as_struct tmp;

        tmp.fraction=0;
        tmp.sign=0;
        tmp.exponent=N+0x7f;
        return *(float*)(&tmp);
};

union uint64_double
{
        uint64_t i;
        double d;
};

double dbl_2n(int N)
{
        union uint64_double tmp;

        tmp.i=((uint64_t)N<<52)+0x3ff0000000000000UL;
        return tmp.d;
};

struct double_as_struct
{
        uint64_t fraction  : 52;
        int exponent  : 11;
        int sign  : 1;
};

double dbl_2n_v2(int N)
{
        struct double_as_struct tmp;

        tmp.fraction=0;
        tmp.sign=0;
        tmp.exponent=N+0x3ff;
        return *(double*)(&tmp);
};

int main()
{
        // 2^11=2048
        printf ("%f\n", flt_2n(11));
```

```
        printf ("%f\n", flt_2n_v2(11));
        printf ("%lf\n", dbl_2n(11));
        printf ("%lf\n", dbl_2n_v2(11));
};
```

L'instruction FSCALE peut être plus rapide dans votre environnement, mais néanmoins, c'est un bon exemple d'*union* et du fait que l'exposant est stocké sous la forme 2^n, donc une valeur n en entrée est décalée à l'exposant dans le nombre encodé en IEEE 754. Ensuite, l'exposant est corrigé avec l'ajout de 0x3f800000 ou de 0x3ff0000000000000.

La même chose peut être faite sans décalage utilisant *struct*, mais en interne, l'opération de décalage aura toujours lieu.

1.26.1

Un autre algorithme connu où un *float* est interprété comme un entier est celui de calcul rapide de racine carrée.

Listing 1.360: Le code source provient de Wikipedia: http://go.yurichev.com/17364

```
 * and that int is 32 bits. */
float sqrt_approx(float z)
{
    int val_int = *(int*)&z; /* Same bits, but as an int */
    /*
     * To justify the following code, prove that
     *
     * ((((val_int / 2^m) - b) / 2) + b) * 2^m = ((val_int - 2^m) / 2) + ((b + 1) / 2) * 2^m)
     *
     * where
     *
     * b = exponent bias
     * m = number of mantissa bits
     *
     * .
     */

    val_int -= 1 << 23; /* Subtract 2^m. */
    val_int >>= 1; /* Divide by 2. */
    val_int += 1 << 29; /* Add ((b + 1) / 2) * 2^m. */

    return *(float*)&val_int; /* Interpret again as float */
}
```

À titre d'exercice, vous pouvez essayez de compiler cette fonction et de comprendre comme elle fonctionne.

C'est un algorithme connu de calcul rapide de $\frac{1}{\sqrt{x}}$. L'algorithme devînt connu, supposément, car il a été utilisé dans Quake III Arena.

La description de l'algorithme peut être trouvée sur Wikipédia: http://go.yurichev.com/17360.

1.27 Pointeurs sur des fonctions

Un pointeur sur une fonction, comme tout autre pointeur, est juste l'adresse de début de la fonction dans son segment de code.

Ils sont souvent utilisés pour appeler des fonctions callback[174] (de rappel).

Des exemples bien connus sont:

- `qsort()`[175], `atexit()`[176] de la bibliothèque C standard;

[174] Wikipédia
[175] Wikipédia
[176] http://go.yurichev.com/17073

- signaux des OS *NIX[177] ;
- démarrage de thread: `CreateThread()` (win32), `pthread_create()` (POSIX);
- beaucoup de fonctions win32, comme `EnumChildWindows()`[178].
- dans de nombreux endroits du noyau Linux, par exemple les fonctions des drivers du système de fichier sont appelées via callbacks: http://go.yurichev.com/17076
- Les fonctions des plugins GCC sont aussi appelées via callbacks: http://go.yurichev.com/17077
- Un autre exemple de pointeurs de fonction est une table dans le gestionnaire de fenêtres Linux « dwm » qui définit les raccourcis. Chaque raccourci a une fonction correspondante à appeler si une touche spécifiques est pressée: GitHub Comme on le voit, une telle table est plus pratique à gérer qu'une grande déclaration switch().

Donc, la fonction `qsort()` est une implémentation du tri rapide dans la bibliothèque standard C/C++. La fonction est capable de trier n'importe quoi, tout type de données, tant que vous avez une fonction pour comparer deux éléments et que `qsort()` est capable de l'appeler.

La fonction de comparaison peut être définie comme:

```
int (*compare)(const void *, const void *)
```

Utilisons l'exemple suivant:

```
/* ex3 Sorting ints with qsort */

#include <stdio.h>
#include <stdlib.h>

int comp(const void * _a, const void * _b)
{
  const int *a=(const int *)_a;
  const int *b=(const int *)_b;

  if (*a==*b)
    return 0;
  else
    if (*a < *b)
        return -1;
     else
      return 1;
}

int main(int argc, char* argv[])
{
   int numbers[10]={1892,45,200,-98,4087,5,-12345,1087,88,-100000};
   int i;

  /* Sort the array */
  qsort(numbers,10,sizeof(int),comp) ;
  for (i=0;i<9;i++)
    printf("Number = %d\n",numbers[ i ]) ;
  return 0;
}
```

1.27.1 MSVC

Compilons le dans MSVC 2010 (certaines parties ont été omises, dans un but de concision) avec l'option /Ox :

Listing 1.361: MSVC 2010 avec optimisation : /GS- /MD

```
__a$ = 8                                                ; size = 4
__b$ = 12                                               ; size = 4
_comp   PROC
        mov     eax, DWORD PTR __a$[esp-4]
```

[177] Wikipédia
[178] MSDN

```
        mov     ecx, DWORD PTR __b$[esp-4]
        mov     eax, DWORD PTR [eax]
        mov     ecx, DWORD PTR [ecx]
        cmp     eax, ecx
        jne     SHORT $LN4@comp
        xor     eax, eax
        ret     0
$LN4@comp :
        xor     edx, edx
        cmp     eax, ecx
        setge   dl
        lea     eax, DWORD PTR [edx+edx-1]
        ret     0
_comp   ENDP


_numbers$ = -40                                         ; size = 40
_argc$ = 8                                              ; size = 4
_argv$ = 12                                             ; size = 4
_main   PROC
        sub     esp, 40                                 ; 00000028H
        push    esi
        push    OFFSET _comp
        push    4
        lea     eax, DWORD PTR _numbers$[esp+52]
        push    10                                      ; 0000000aH
        push    eax
        mov     DWORD PTR _numbers$[esp+60], 1892       ; 00000764H
        mov     DWORD PTR _numbers$[esp+64], 45         ; 0000002dH
        mov     DWORD PTR _numbers$[esp+68], 200        ; 000000c8H
        mov     DWORD PTR _numbers$[esp+72], -98        ; ffffff9eH
        mov     DWORD PTR _numbers$[esp+76], 4087       ; 00000ff7H
        mov     DWORD PTR _numbers$[esp+80], 5
        mov     DWORD PTR _numbers$[esp+84], -12345     ; ffffcfc7H
        mov     DWORD PTR _numbers$[esp+88], 1087       ; 0000043fH
        mov     DWORD PTR _numbers$[esp+92], 88         ; 00000058H
        mov     DWORD PTR _numbers$[esp+96], -100000    ; fffe7960H
        call    _qsort
        add     esp, 16                                 ; 00000010H

...
```

Rien de surprenant jusqu'ici. Comme quatrième argument, l'adresse du label `_comp` est passée, qui est juste l'endroit où se trouve `comp()`, ou, en d'autres mots, l'adresse de la première instruction de cette fonction.

Comment est-ce que `qsort()` l'appelle?

Regardons cette fonction, située dans MSVCR80.DLL (un module DLL de MSVC avec des fonctions de la bibliothèque C standard):

Listing 1.362: MSVCR80.DLL

```
.text :7816CBF0 ; void __cdecl qsort(void *, unsigned int, unsigned int, int (__cdecl *)(const ↙
    ↳ void *, const void *))
.text :7816CBF0                 public _qsort
.text :7816CBF0 _qsort          proc near
.text :7816CBF0
.text :7816CBF0 lo              = dword ptr -104h
.text :7816CBF0 hi              = dword ptr -100h
.text :7816CBF0 var_FC          = dword ptr -0FCh
.text :7816CBF0 stkptr          = dword ptr -0F8h
.text :7816CBF0 lostk           = dword ptr -0F4h
.text :7816CBF0 histk           = dword ptr -7Ch
.text :7816CBF0 base            = dword ptr  4
.text :7816CBF0 num             = dword ptr  8
.text :7816CBF0 width           = dword ptr  0Ch
.text :7816CBF0 comp            = dword ptr  10h
.text :7816CBF0
.text :7816CBF0                 sub     esp, 100h
```

```
....

.text :7816CCE0 loc_7816CCE0 :                          ; CODE XREF : _qsort+B1
.text :7816CCE0                 shr     eax, 1
.text :7816CCE2                 imul    eax, ebp
.text :7816CCE5                 add     eax, ebx
.text :7816CCE7                 mov     edi, eax
.text :7816CCE9                 push    edi
.text :7816CCEA                 push    ebx
.text :7816CCEB                 call    [esp+118h+comp]
.text :7816CCF2                 add     esp, 8
.text :7816CCF5                 test    eax, eax
.text :7816CCF7                 jle     short loc_7816CD04
```

comp—est le quatrième argument de la fonction. Ici, le contrôle est passé à l'adresse dans l'argument comp. Avant cela, deux arguments sont préparés pour comp(). Son résultat est testé après son exécution.

C'est pourquoi il est dangereux d'utiliser des pointeurs sur des fonctions. Tout d'abord, si vous appelez qsort() avec un pointeur de fonction incorrect, qsort() peut passer le contrôle du flux à un point incorrect, le processus peut planter et ce bug sera difficile à trouver.

La seconde raison est que les types de la fonction de callback doivent être strictement conforme, appeler la mauvaise fonction avec de mauvais arguments du mauvais type peut conduire à de sérieux problèmes, toutefois, le plantage du processus n'est pas un problème ici —le problème est de comment déterminer la raison du plantage —car le compilateur peut être silencieux sur le problème potentiel lors de la compilation.

MSVC + OllyDbg

Chargeons notre exemple dans OllyDbg et mettons un point d'arrêt sur `comp()`. Nous voyons comment les valeurs sont comparées lors du premier appel de `comp()` :

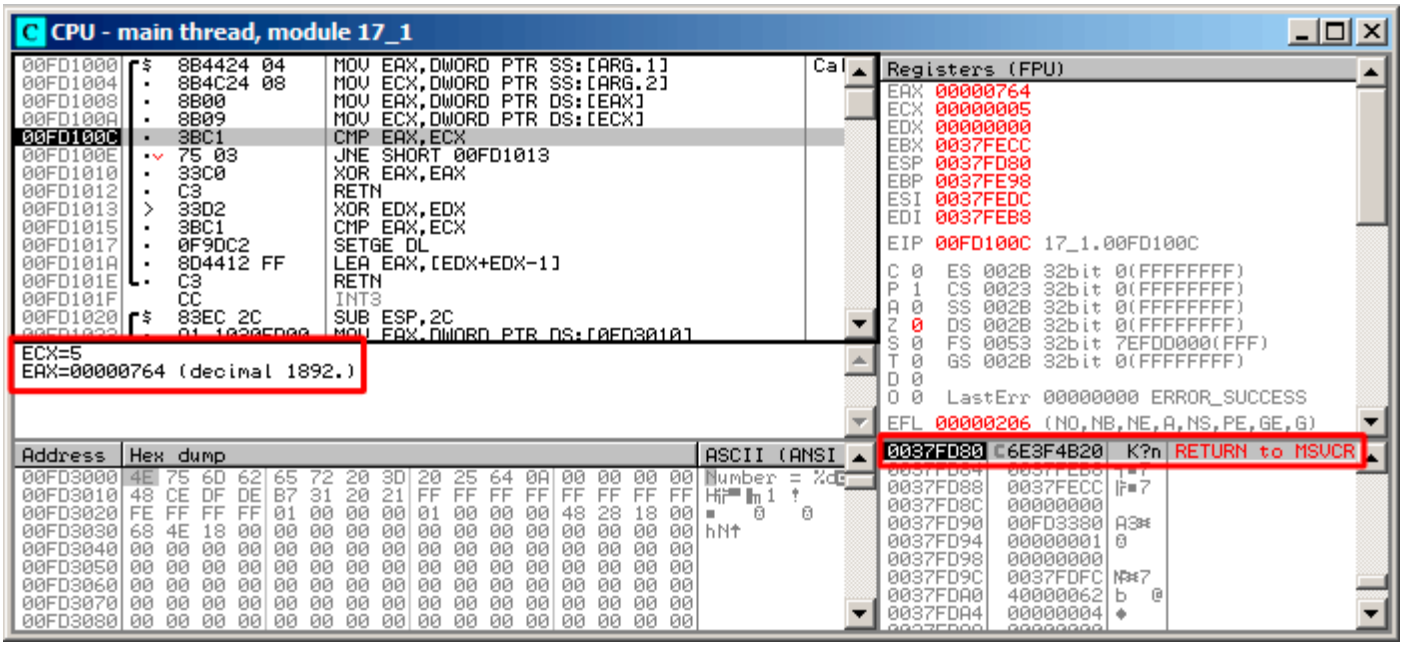


Fig. 1.109: OllyDbg : premier appel de `comp()`

OllyDbg montre les valeurs comparées dans la fenêtre sous celle du code, par commodité. Nous voyons que SP pointe sur RA, où se trouve la fonction `qsort()` (dans `MSVCR100.DLL`).

En traçant (F8) jusqu'à l'instruction RETN et appuyant sur F8 une fois de plus, nous retournons à la fonction `qsort()` :

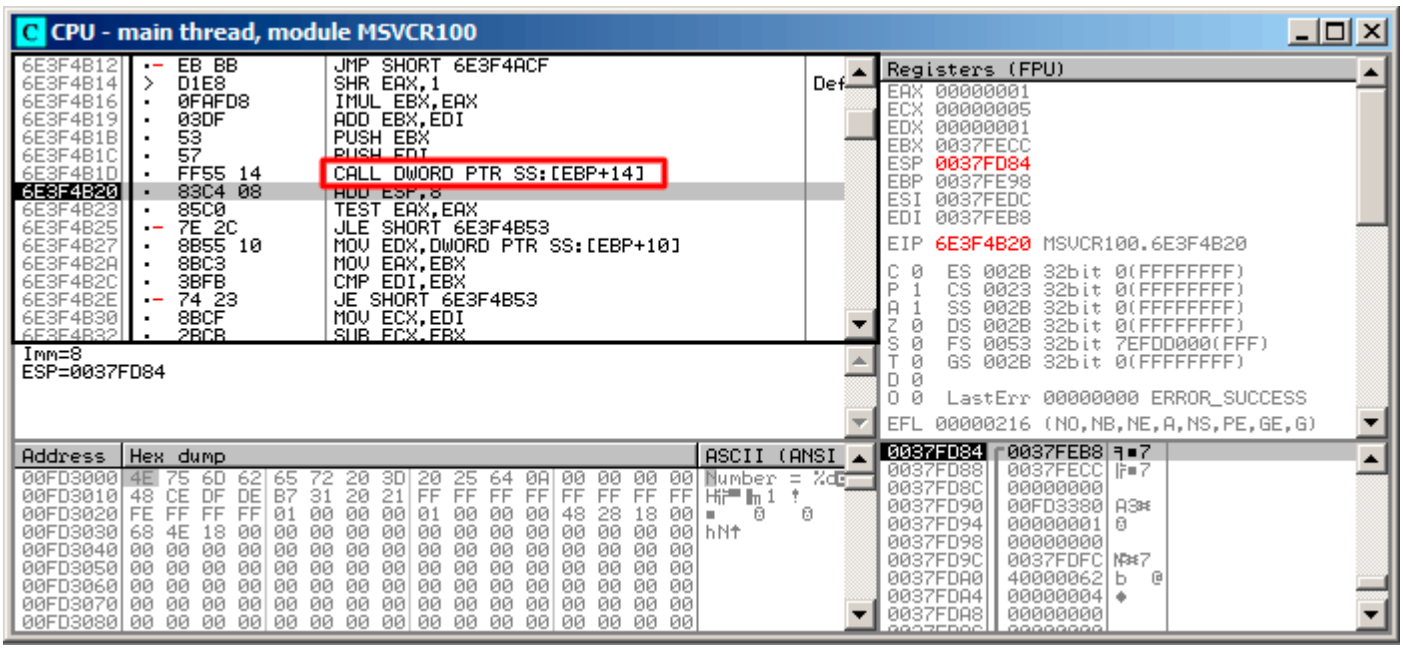


Fig. 1.110: OllyDbg : le code dans `qsort()` juste après l'appel de `comp()`

Ça a été un appel à la fonction de comparaison.

Voici aussi une copie d'écran au moment du second appel à `comp()`—maintenant les valeurs qui doivent être comparées sont différentes:

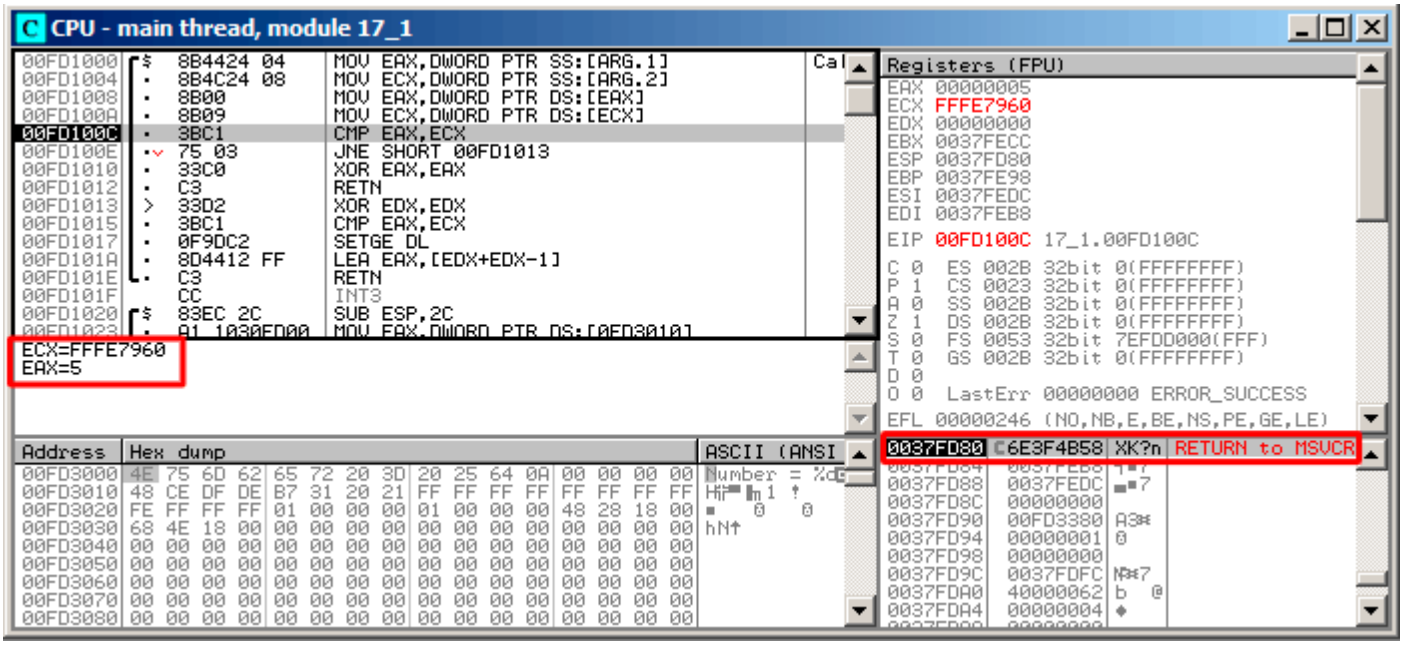


Fig. 1.111: OllyDbg : second appel de `comp()`

MSVC + tracer

Regardons quelles sont le paires comparées. Ces 10 nombres vont être triés: 1892, 45, 200, -98, 4087, 5, -12345, 1087, 88, -100000.

Nous avons l'adresse de la première instruction `CMP` dans `comp()`, c'est `0x0040100C` et nous y avons mis un point d'arrêt:

```
tracer.exe -l :17_1.exe bpx=17_1.exe!0x0040100C
```

Maintenant nous avons des informations sur les registres au point d'arrêt:

```
PID=4336|New process 17_1.exe
(0) 17_1.exe!0x40100c
EAX=0x00000764 EBX=0x0051f7c8 ECX=0x00000005 EDX=0x00000000
ESI=0x0051f7d8 EDI=0x0051f7b4 EBP=0x0051f794 ESP=0x0051f67c
EIP=0x0028100c
FLAGS=IF
(0) 17_1.exe!0x40100c
EAX=0x00000005 EBX=0x0051f7c8 ECX=0xfffe7960 EDX=0x00000000
ESI=0x0051f7d8 EDI=0x0051f7b4 EBP=0x0051f794 ESP=0x0051f67c
EIP=0x0028100c
FLAGS=PF ZF IF
(0) 17_1.exe!0x40100c
EAX=0x00000764 EBX=0x0051f7c8 ECX=0x00000005 EDX=0x00000000
ESI=0x0051f7d8 EDI=0x0051f7b4 EBP=0x0051f794 ESP=0x0051f67c
EIP=0x0028100c
FLAGS=CF PF ZF IF
...
```

Filtrons sur EAX et ECX et nous obtenons:

```
EAX=0x00000764 ECX=0x00000005
EAX=0x00000005 ECX=0xfffe7960
EAX=0x00000764 ECX=0x00000005
EAX=0x0000002d ECX=0x00000005
EAX=0x00000058 ECX=0x00000005
EAX=0x0000043f ECX=0x00000005
EAX=0xffffcfc7 ECX=0x00000005
```

```
EAX=0x000000c8 ECX=0x00000005
EAX=0xffffff9e ECX=0x00000005
EAX=0x00000ff7 ECX=0x00000005
EAX=0x00000ff7 ECX=0x00000005
EAX=0xffffff9e ECX=0x00000005
EAX=0xffffff9e ECX=0x00000005
EAX=0xffffcfc7 ECX=0xfffe7960
EAX=0x00000005 ECX=0xffffcfc7
EAX=0xffffff9e ECX=0x00000005
EAX=0xffffcfc7 ECX=0xfffe7960
EAX=0xffffff9e ECX=0xffffcfc7
EAX=0xffffcfc7 ECX=0xfffe7960
EAX=0x000000c8 ECX=0x00000ff7
EAX=0x0000002d ECX=0x00000ff7
EAX=0x0000043f ECX=0x00000ff7
EAX=0x00000058 ECX=0x00000ff7
EAX=0x00000764 ECX=0x00000ff7
EAX=0x000000c8 ECX=0x00000764
EAX=0x0000002d ECX=0x00000764
EAX=0x0000043f ECX=0x00000764
EAX=0x00000058 ECX=0x00000764
EAX=0x000000c8 ECX=0x00000058
EAX=0x0000002d ECX=0x000000c8
EAX=0x0000043f ECX=0x000000c8
EAX=0x000000c8 ECX=0x00000058
EAX=0x0000002d ECX=0x000000c8
EAX=0x0000002d ECX=0x00000058
```

Il y a 34 paires. C’est pourquoi, l’algorithme de tri rapide a besoin de 34 opérations de comparaison pour trier ces 10 nombres.

MSVC + tracer (couverture du code)

Nous pouvons aussi utiliser la capacité du tracer pour collecter tous les registres possible et les montrer dans IDA.

Exécutons pas à pas toutes les instructions dans `comp()` :

```
tracer.exe -l :17_1.exe bpf=17_1.exe!0x00401000,trace :cc
```

Nous obtenons un scipt .idc pour charger dans IDA et chargeons le:

```
.text:00401000
.text:00401000 ; int __cdecl PtFuncCompare(const void *, const void *)
.text:00401000 PtFuncCompare   proc near               ; DATA XREF: _main+5↓o
.text:00401000
.text:00401000 arg_0           = dword ptr  4
.text:00401000 arg_4           = dword ptr  8
.text:00401000
.text:00401000                 mov     eax, [esp+arg_0] ; [ESP+4]=0x45f7ec..0x45f810(step=4), L"?\x04?
.text:00401004                 mov     ecx, [esp+arg_4] ; [ESP+8]=0x45f7ec..0x45f7f4(step=4), 0x45f7fc
.text:00401008                 mov     eax, [eax]       ; [EAX]=5, 0x2d, 0x58, 0xc8, 0x43f, 0x764, 0xff
.text:0040100A                 mov     ecx, [ecx]       ; [ECX]=5, 0x58, 0xc8, 0x764, 0xff7, 0xfffe7960
.text:0040100C                 cmp     eax, ecx         ; EAX=5, 0x2d, 0x58, 0xc8, 0x43f, 0x764, 0xff7,
.text:0040100E                 jnz     short loc_401013 ; ZF=false
.text:00401010                 xor     eax, eax
.text:00401012                 retn
.text:00401013 ; ---------------------------------------------------------------------------
.text:00401013
.text:00401013 loc_401013:                              ; CODE XREF: PtFuncCompare+E↑j
.text:00401013                 xor     edx, edx
.text:00401015                 cmp     eax, ecx         ; EAX=5, 0x2d, 0x58, 0xc8, 0x43f, 0x764, 0xff7,
.text:00401017                 setnl   dl               ; SF=false,true OF=false
.text:0040101A                 lea     eax, [edx+edx-1]
.text:0040101E                 retn                     ; EAX=1, 0xffffffff
.text:0040101E PtFuncCompare   endp
.text:0040101F
```

Fig. 1.112: tracer et IDA. N.B.: certaines valeurs sont coupées à droite

IDA a donné un nom à la fonction (PtFuncCompare)—car IDA voit que le pointeur sur cette fonction est passé à `qsort()`.

Nous voyons que les pointeurs a et b pointent sur différents emplacements dans le tableau, mais le pas entre eux est 4, puisque des valeurs 32-bit sont stockées dans le tableau.

Nous voyons que les instructions en `0x401010` et `0x401012` ne sont jamais exécutées (donc elles sont laissées en blanc): en effet, `comp()` ne renvoie jamais 0, car il n'y a pas d'éléments égaux dans le tableau.

1.27.2 GCC

Il n'y a pas beaucoup de différence:

Listing 1.363: GCC

```
                lea     eax, [esp+40h+var_28]
                mov     [esp+40h+var_40], eax
                mov     [esp+40h+var_28], 764h
                mov     [esp+40h+var_24], 2Dh
                mov     [esp+40h+var_20], 0C8h
                mov     [esp+40h+var_1C], 0FFFFFF9Eh
                mov     [esp+40h+var_18], 0FF7h
                mov     [esp+40h+var_14], 5
                mov     [esp+40h+var_10], 0FFFFCFC7h
                mov     [esp+40h+var_C], 43Fh
                mov     [esp+40h+var_8], 58h
                mov     [esp+40h+var_4], 0FFFE7960h
                mov     [esp+40h+var_34], offset comp
                mov     [esp+40h+var_38], 4
                mov     [esp+40h+var_3C], 0Ah
```

```
                call    _qsort
```

Fonction comp() :

```
                public comp
comp            proc near

arg_0           = dword ptr  8
arg_4           = dword ptr  0Ch

                push    ebp
                mov     ebp, esp
                mov     eax, [ebp+arg_4]
                mov     ecx, [ebp+arg_0]
                mov     edx, [eax]
                xor     eax, eax
                cmp     [ecx], edx
                jnz     short loc_8048458
                pop     ebp
                retn
loc_8048458 :
                setnl   al
                movzx   eax, al
                lea     eax, [eax+eax-1]
                pop     ebp
                retn
comp            endp
```

L'implémentation de qsort() se trouve dans libc.so.6 et c'est en fait juste un wrapper[179] pour qsort_r().

À son tour, elle appelle quicksort(), où notre fonction défini est appelée via le pointeur passé:

Listing 1.364: (fihier libc.so.6, glibc version—2.10.1)

```
...
.text :0002DDF6                 mov     edx, [ebp+arg_10]
.text :0002DDF9                 mov     [esp+4], esi
.text :0002DDFD                 mov     [esp], edi
.text :0002DE00                 mov     [esp+8], edx
.text :0002DE04                 call    [ebp+arg_C]
...
```

GCC + GDB (avec code source)

Évidemment, nous avons le code source C de notre exemple (1.27 on page 387), donc nous pouvons mettre un point d'arrêt (b) sur le numéro de ligne (11—la ligne où la première comparaison se produit. Nous devons aussi compiler l'exemple avec les informations de débogage incluses (-g), donc la table avec les adresses et les numéros de ligne correspondants est présente.

Nous pouvons aussi afficher les valeurs en utilisant les noms de variable (p): les informations de débogage nous disent aussi quel registre et/ou élément de la pile locale contient quelle variable.

Nous pouvons aussi voir la pile (bt) et y trouver qu'il y a une fonction intermédiaire msort_with_tmp() utilisée dans la Glibc.

Listing 1.365: session GDB

```
dennis@ubuntuvm :~/polygon$ gcc 17_1.c -g
dennis@ubuntuvm :~/polygon$ gdb ./a.out
GNU gdb (GDB) 7.6.1-ubuntu
Copyright (C) 2013 Free Software Foundation, Inc.
...
Reading symbols from /home/dennis/polygon/a.out...done.
(gdb) b 17_1.c :11
Breakpoint 1 at 0x804845f : file 17_1.c, line 11.
(gdb) run
Starting program : /home/dennis/polygon/./a.out
```

[179] un concept similaire à une fonction thunk

```
Breakpoint 1, comp (_a=0xbffff0f8, _b=_b@entry=0xbffff0fc) at 17_1.c :11
11        if (*a==*b)
(gdb) p *a
$1 = 1892
(gdb) p *b
$2 = 45
(gdb) c
Continuing.

Breakpoint 1, comp (_a=0xbffff104, _b=_b@entry=0xbffff108) at 17_1.c :11
11        if (*a==*b)
(gdb) p *a
$3 = -98
(gdb) p *b
$4 = 4087
(gdb) bt
#0  comp (_a=0xbffff0f8, _b=_b@entry=0xbffff0fc) at 17_1.c :11
#1  0xb7e42872 in msort_with_tmp (p=p@entry=0xbffff07c, b=b@entry=0xbffff0f8, n=n@entry=2)
    at msort.c :65
#2  0xb7e4273e in msort_with_tmp (n=2, b=0xbffff0f8, p=0xbffff07c) at msort.c :45
#3  msort_with_tmp (p=p@entry=0xbffff07c, b=b@entry=0xbffff0f8, n=n@entry=5) at msort.c :53
#4  0xb7e4273e in msort_with_tmp (n=5, b=0xbffff0f8, p=0xbffff07c) at msort.c :45
#5  msort_with_tmp (p=p@entry=0xbffff07c, b=b@entry=0xbffff0f8, n=n@entry=10) at msort.c :53
#6  0xb7e42cef in msort_with_tmp (n=10, b=0xbffff0f8, p=0xbffff07c) at msort.c :45
#7  __GI_qsort_r (b=b@entry=0xbffff0f8, n=n@entry=10, s=s@entry=4, cmp=cmp@entry=0x804844d <↙
    ↳ comp>,
    arg=arg@entry=0x0) at msort.c :297
#8  0xb7e42dcf in __GI_qsort (b=0xbffff0f8, n=10, s=4, cmp=0x804844d <comp>) at msort.c :307
#9  0x0804850d in main (argc=1, argv=0xbffff1c4) at 17_1.c :26
(gdb)
```

GCC + GDB (pas de code source)

Mais souvent, il n'y a pas de code source du tout, donc nous pouvons désassembler la fonction `comp()` (`disas`), trouver la toute première instruction CMP et placer un point d'arrêt (b) à cette adresse.

À chaque point d'arrêt, nous allons afficher le contenu de tous les registres (`info registers`). Le contenu de la pile est aussi disponible (`bt`),

mais partiellement: il n'y a pas l'information du numéro de ligne pour `comp()`.

Listing 1.366: session GDB

```
dennis@ubuntuvm :~/polygon$ gcc 17_1.c
dennis@ubuntuvm :~/polygon$ gdb ./a.out
GNU gdb (GDB) 7.6.1-ubuntu
Copyright (C) 2013 Free Software Foundation, Inc.
...
Reading symbols from /home/dennis/polygon/a.out...(no debugging symbols found)...done.
(gdb) set disassembly-flavor intel
(gdb) disas comp
Dump of assembler code for function comp :
   0x0804844d <+0> :     push   ebp
   0x0804844e <+1> :     mov    ebp,esp
   0x08048450 <+3> :     sub    esp,0x10
   0x08048453 <+6> :     mov    eax,DWORD PTR [ebp+0x8]
   0x08048456 <+9> :     mov    DWORD PTR [ebp-0x8],eax
   0x08048459 <+12> :    mov    eax,DWORD PTR [ebp+0xc]
   0x0804845c <+15> :    mov    DWORD PTR [ebp-0x4],eax
   0x0804845f <+18> :    mov    eax,DWORD PTR [ebp-0x8]
   0x08048462 <+21> :    mov    edx,DWORD PTR [eax]
   0x08048464 <+23> :    mov    eax,DWORD PTR [ebp-0x4]
   0x08048467 <+26> :    mov    eax,DWORD PTR [eax]
   0x08048469 <+28> :    cmp    edx,eax
   0x0804846b <+30> :    jne    0x8048474 <comp+39>
   0x0804846d <+32> :    mov    eax,0x0
   0x08048472 <+37> :    jmp    0x804848e <comp+65>
   0x08048474 <+39> :    mov    eax,DWORD PTR [ebp-0x8]
```

```
   0x08048477 <+42> :    mov    edx,DWORD PTR [eax]
   0x08048479 <+44> :    mov    eax,DWORD PTR [ebp-0x4]
   0x0804847c <+47> :    mov    eax,DWORD PTR [eax]
   0x0804847e <+49> :    cmp    edx,eax
   0x08048480 <+51> :    jge    0x8048489 <comp+60>
   0x08048482 <+53> :    mov    eax,0xffffffff
   0x08048487 <+58> :    jmp    0x804848e <comp+65>
   0x08048489 <+60> :    mov    eax,0x1
   0x0804848e <+65> :    leave
   0x0804848f <+66> :    ret
End of assembler dump.
(gdb) b *0x08048469
Breakpoint 1 at 0x8048469
(gdb) run
Starting program : /home/dennis/polygon/./a.out

Breakpoint 1, 0x08048469 in comp ()
(gdb) info registers
eax            0x2d     45
ecx            0xbffff0f8       -1073745672
edx            0x764    1892
ebx            0xb7fc0000       -1208221696
esp            0xbfffeeb8       0xbfffeeb8
ebp            0xbfffeec8       0xbfffeec8
esi            0xbffff0fc       -1073745668
edi            0xbffff010       -1073745904
eip            0x8048469        0x8048469 <comp+28>
eflags         0x286    [ PF SF IF ]
cs             0x73     115
ss             0x7b     123
ds             0x7b     123
es             0x7b     123
fs             0x0      0
gs             0x33     51
(gdb) c
Continuing.

Breakpoint 1, 0x08048469 in comp ()
(gdb) info registers
eax            0xff7    4087
ecx            0xbffff104       -1073745660
edx            0xffffff9e       -98
ebx            0xb7fc0000       -1208221696
esp            0xbfffee58       0xbfffee58
ebp            0xbfffee68       0xbfffee68
esi            0xbffff108       -1073745656
edi            0xbffff010       -1073745904
eip            0x8048469        0x8048469 <comp+28>
eflags         0x282    [ SF IF ]
cs             0x73     115
ss             0x7b     123
ds             0x7b     123
es             0x7b     123
fs             0x0      0
gs             0x33     51
(gdb) c
Continuing.

Breakpoint 1, 0x08048469 in comp ()
(gdb) info registers
eax            0xffffff9e       -98
ecx            0xbffff100       -1073745664
edx            0xc8     200
ebx            0xb7fc0000       -1208221696
esp            0xbfffeeb8       0xbfffeeb8
ebp            0xbfffeec8       0xbfffeec8
esi            0xbffff104       -1073745660
edi            0xbffff010       -1073745904
eip            0x8048469        0x8048469 <comp+28>
eflags         0x286    [ PF SF IF ]
```

```
cs             0x73     115
ss             0x7b     123
ds             0x7b     123
es             0x7b     123
fs             0x0      0
gs             0x33     51
(gdb) bt
#0  0x08048469 in comp ()
#1  0xb7e42872 in msort_with_tmp (p=p@entry=0xbffff07c, b=b@entry=0xbffff0f8, n=n@entry=2)
    at msort.c :65
#2  0xb7e4273e in msort_with_tmp (n=2, b=0xbffff0f8, p=0xbffff07c) at msort.c :45
#3  msort_with_tmp (p=p@entry=0xbffff07c, b=b@entry=0xbffff0f8, n=n@entry=5) at msort.c :53
#4  0xb7e4273e in msort_with_tmp (n=5, b=0xbffff0f8, p=0xbffff07c) at msort.c :45
#5  msort_with_tmp (p=p@entry=0xbffff07c, b=b@entry=0xbffff0f8, n=n@entry=10) at msort.c :53
#6  0xb7e42cef in msort_with_tmp (n=10, b=0xbffff0f8, p=0xbffff07c) at msort.c :45
#7  __GI_qsort_r (b=b@entry=0xbffff0f8, n=n@entry=10, s=s@entry=4, cmp=cmp@entry=0x804844d <↙
    ↳ comp>,
    arg=arg@entry=0x0) at msort.c :297
#8  0xb7e42dcf in __GI_qsort (b=0xbffff0f8, n=10, s=4, cmp=0x804844d <comp>) at msort.c :307
#9  0x0804850d in main ()
```

1.27.3 Danger des pointeurs sur des fonctions

Comme nous pouvons le voir, la fonction `qsort()` attend un pointeur sur une fonction qui prend deux arguments de type *void** et renvoie un entier: Si vous avez plusieurs fonctions de comparaison dans votre code (une qui compare les chaînes, une autre—les entiers, etc.), il est très facile de les mélanger les unes avec les autres. Vous pouvez essayer de trier un tableau de chaîne en utilisant une fonction qui compare les entiers, et le compilateur ne vous avertira pas de ce bogue.

1.28 Valeurs 64-bit dans un environnement 32-bit

Dans un environnement 32-bit, les GPR sont 32-bit, donc les valeurs 64-bit sont stockées et passées comme une paire de registres 32-bit[180].

1.28.1 Renvoyer une valeur 64-bit

```
#include <stdint.h>

uint64_t f ()
{
        return 0x1234567890ABCDEF;
};
```

x86

Dans un environnement 32-bit, les valeurs 64-bit sont renvoyées des fonctions dans la paire de registres EDX :EAX.

Listing 1.367: MSVC 2010 avec optimisation

```
_f      PROC
        mov     eax, -1867788817 ; 90abcdefH
        mov     edx, 305419896   ; 12345678H
        ret     0
_f      ENDP
```

[180] A propos, les valeurs 32-bit sont passées en tant que paire dans les environnements 16-bit de la même manière: **??** on page ??

ARM

Une valeur 64-bit est renvoyée dans la paire de registres R0-R1 (R1 est pour la partie haute et R0 pour la partie basse):

Listing 1.368: avec optimisation Keil 6/2013 (Mode ARM)

```
||f|| PROC
        LDR      r0,|L0.12|
        LDR      r1,|L0.16|
        BX       lr
        ENDP

|L0.12|
        DCD      0x90abcdef
|L0.16|
        DCD      0x12345678
```

MIPS

Une valeur 64-bit est renvoyée dans la paire de registres V0-V1 ($2-$3) (V0 ($2) est pour la partie haute et V1 ($3) pour la partie basse):

Listing 1.369: GCC 4.4.5 avec optimisation (listing assembleur)

```
        li      $3,-1867841536   # 0xffffffff90ab0000
        li      $2,305397760     # 0x12340000
        ori     $3,$3,0xcdef
        j       $31
        ori     $2,$2,0x5678
```

Listing 1.370: GCC 4.4.5 avec optimisation (IDA)

```
       lui     $v1, 0x90AB
       lui     $v0, 0x1234
       li      $v1, 0x90ABCDEF
       jr      $ra
       li      $v0, 0x12345678
```

1.28.2 Passage d'arguments, addition, soustraction

```
#include <stdint.h>

uint64_t f_add (uint64_t a, uint64_t b)
{
        return a+b;
};

void f_add_test ()
{
#ifdef __GNUC__
        printf ("%lld\n", f_add(12345678901234, 23456789012345));
#else
        printf ("%I64d\n", f_add(12345678901234, 23456789012345));
#endif
};

uint64_t f_sub (uint64_t a, uint64_t b)
{
        return a-b;
};
```

x86

Listing 1.371: MSVC 2012 /Ob1 avec optimisation

```
_a$ = 8         ; size = 8
_b$ = 16        ; size = 8
_f_add  PROC
        mov     eax, DWORD PTR _a$[esp-4]
        add     eax, DWORD PTR _b$[esp-4]
        mov     edx, DWORD PTR _a$[esp]
        adc     edx, DWORD PTR _b$[esp]
        ret     0
_f_add  ENDP

_f_add_test PROC
        push    5461            ; 00001555H
        push    1972608889      ; 75939f79H
        push    2874            ; 00000b3aH
        push    1942892530      ; 73ce2ff_subH
        call    _f_add
        push    edx
        push    eax
        push    OFFSET $SG1436 ; '%I64d', 0aH, 00H
        call    _printf
        add     esp, 28
        ret     0
_f_add_test ENDP

_f_sub  PROC
        mov     eax, DWORD PTR _a$[esp-4]
        sub     eax, DWORD PTR _b$[esp-4]
        mov     edx, DWORD PTR _a$[esp]
        sbb     edx, DWORD PTR _b$[esp]
        ret     0
_f_sub  ENDP
```

Nous voyons dans la fonction `f_add_test()` que chaque valeur 64-bit est passée en utilisant deux valeurs 32-bit, partie haute d'abord, puis partie basse.

L'addition et la soustraction se déroulent aussi par paire.

Pour l'addition, la partie basse 32-bit est d'abord additionnée. Si il y a eu une retenue pendant l'addition, le flag CF est mis.

L'instruction suivante ADC additionne les parties hautes, et ajoute aussi 1 si $CF = 1$.

La soustraction est aussi effectuée par paire. Le premier SUB peut aussi mettre le flag CF, qui doit être testé lors de l'instruction SBB suivante: Si le flag de retenue est mis, alors 1 est soustrait du résultat.

Il est facile de voir comment le résultat de la fonction `f_add()` est passé à `printf()`.

Listing 1.372: GCC 4.8.1 -O1 -fno-inline

```
_f_add :
        mov     eax, DWORD PTR [esp+12]
        mov     edx, DWORD PTR [esp+16]
        add     eax, DWORD PTR [esp+4]
        adc     edx, DWORD PTR [esp+8]
        ret

_f_add_test :
        sub     esp, 28
        mov     DWORD PTR [esp+8], 1972608889   ; 75939f79H
        mov     DWORD PTR [esp+12], 5461        ; 00001555H
        mov     DWORD PTR [esp], 1942892530     ; 73ce2ff_subH
        mov     DWORD PTR [esp+4], 2874         ; 00000b3aH
        call    _f_add
        mov     DWORD PTR [esp+4], eax
        mov     DWORD PTR [esp+8], edx
        mov     DWORD PTR [esp], OFFSET FLAT :LC0 ; "%lld\12\0"
        call    _printf
        add     esp, 28
```

```
        ret

_f_sub :
        mov     eax, DWORD PTR [esp+4]
        mov     edx, DWORD PTR [esp+8]
        sub     eax, DWORD PTR [esp+12]
        sbb     edx, DWORD PTR [esp+16]
        ret
```

Le code de GCC est le même.

ARM

Listing 1.373: avec optimisation Keil 6/2013 (Mode ARM)

```
f_add PROC
        ADDS    r0,r0,r2
        ADC     r1,r1,r3
        BX      lr
        ENDP

f_sub PROC
        SUBS    r0,r0,r2
        SBC     r1,r1,r3
        BX      lr
        ENDP

f_add_test PROC
        PUSH    {r4,lr}
        LDR     r2,|L0.68| ; 0x75939f79
        LDR     r3,|L0.72| ; 0x00001555
        LDR     r0,|L0.76| ; 0x73ce2ff2
        LDR     r1,|L0.80| ; 0x00000b3a
        BL      f_add
        POP     {r4,lr}
        MOV     r2,r0
        MOV     r3,r1
        ADR     r0,|L0.84| ; "%I64d\n"
        B       __2printf
        ENDP

|L0.68|
        DCD     0x75939f79
|L0.72|
        DCD     0x00001555
|L0.76|
        DCD     0x73ce2ff2
|L0.80|
        DCD     0x00000b3a
|L0.84|
        DCB     "%I64d\n",0
```

La première valeur 64-bit est passée par la paire de registres R0 et R1, la seconde dans la paire de registres R2 et R3. ARM a aussi l'instruction ADC (qui compte le flag de retenue) et SBC (« soustraction avec retenue »). Chose importante: lorsque les parties basses sont ajoutées/soustraites, les instructions ADDS et SUBS avec le suffixe -S sont utilisées. Le suffixe -S signifie « mettre les flags », et les flags (en particulier le flag de retenue) est ce dont les instructions suivantes ADC/SBC ont besoin. Autrement, les instructions sans le suffixe -S feraient le travail (ADD et SUB).

MIPS

Listing 1.374: GCC 4.4.5 avec optimisation (IDA)

```
f_add :
; $a0 - partie haute de a
; $a1 - partie basse de a
```

```
; $a2 - partie haute de b
; $a3 - partie basse de b
                addu    $v1, $a3, $a1 ; ajouter les parties basses
                addu    $a0, $a2, $a0 ; ajouter les parties hautes
; est-ce qu'une retenue a été générée lors de l'addition des parties basses ?
; si oui, mettre $v0 à 1
                sltu    $v0, $v1, $a3
                jr      $ra
; ajouter 1 à la partie haute du résultat si la retenue doit être générée :
                addu    $v0, $a0 ; slot de délai de branchement
; $v0 - partie haute du résultat
; $v1 - partie basse du résultat

f_sub :
; $a0 - partie haute de a
; $a1 - partie basse de a
; $a2 - partie haute de b
; $a3 - partie basse de b
                subu    $v1, $a1, $a3 ; soustraire les parties basses
                subu    $v0, $a0, $a2 ; soustraire les parties hautes
; est-ce qu'une retenue a été générée lors de la soustraction des parties basses ?
; si oui, mettre $a0 à 1
                sltu    $a1, $v1
                jr      $ra
; soustraire 1 à la partie haute du résultat si la retenue doit être générée :
                subu    $v0, $a1 ; slot de délai de branchement
; $v0 - partie haute du résultat
; $v1 - partie basse du résultat

f_add_test :

var_10          = -0x10
var_4           = -4

                lui     $gp, (__gnu_local_gp >> 16)
                addiu   $sp, -0x20
                la      $gp, (__gnu_local_gp & 0xFFFF)
                sw      $ra, 0x20+var_4($sp)
                sw      $gp, 0x20+var_10($sp)
                lui     $a1, 0x73CE
                lui     $a3, 0x7593
                li      $a0, 0xB3A
                li      $a3, 0x75939F79
                li      $a2, 0x1555
                jal     f_add
                li      $a1, 0x73CE2FF2
                lw      $gp, 0x20+var_10($sp)
                lui     $a0, ($LC0 >> 16)  # "%lld\n"
                lw      $t9, (printf & 0xFFFF)($gp)
                lw      $ra, 0x20+var_4($sp)
                la      $a0, ($LC0 & 0xFFFF)  # "%lld\n"
                move    $a3, $v1
                move    $a2, $v0
                jr      $t9
                addiu   $sp, 0x20

$LC0 :           .ascii "%lld\n"<0>
```

MIPS n'a pas de registre de flags, donc il n'y a pas cette information après l'exécution des opérations arithmétiques. Donc il n'y a pas d'instructions comme ADC et SBB du x86. Pour savoir si le flag de retenue serait mis, une comparaison est faite (en utilisant l'instruction SLTU), qui met le registre de destination à 1 ou 0. Ce 1 ou ce 0 est ensuite ajouté ou soustrait au/du résultat final.

1.28.3 Multiplication, division

```
#include <stdint.h>

uint64_t f_mul (uint64_t a, uint64_t b)
```

```
{
        return a*b;
};

uint64_t f_div (uint64_t a, uint64_t b)
{
        return a/b;
};

uint64_t f_rem (uint64_t a, uint64_t b)
{
        return a % b;
};
```

x86

Listing 1.375: MSVC 2013 /Ob1 avec optimisation

```
_a$ = 8 ; signe = 8
_b$ = 16 ; signe = 8
_f_mul  PROC
        push    ebp
        mov     ebp, esp
        mov     eax, DWORD PTR _b$[ebp+4]
        push    eax
        mov     ecx, DWORD PTR _b$[ebp]
        push    ecx
        mov     edx, DWORD PTR _a$[ebp+4]
        push    edx
        mov     eax, DWORD PTR _a$[ebp]
        push    eax
        call    __allmul ; multiplication long long
        pop     ebp
        ret     0
_f_mul  ENDP

_a$ = 8 ; signe = 8
_b$ = 16 ; signe = 8
_f_div  PROC
        push    ebp
        mov     ebp, esp
        mov     eax, DWORD PTR _b$[ebp+4]
        push    eax
        mov     ecx, DWORD PTR _b$[ebp]
        push    ecx
        mov     edx, DWORD PTR _a$[ebp+4]
        push    edx
        mov     eax, DWORD PTR _a$[ebp]
        push    eax
        call    __aulldiv ; division long long non signée
        pop     ebp
        ret     0
_f_div  ENDP

_a$ = 8 ; signe = 8
_b$ = 16 ; signe = 8
_f_rem  PROC
        push    ebp
        mov     ebp, esp
        mov     eax, DWORD PTR _b$[ebp+4]
        push    eax
        mov     ecx, DWORD PTR _b$[ebp]
        push    ecx
        mov     edx, DWORD PTR _a$[ebp+4]
        push    edx
        mov     eax, DWORD PTR _a$[ebp]
        push    eax
        call    __aullrem ; reste long long non signé
```

```
        pop     ebp
        ret     0
_f_rem  ENDP
```

La multiplication et la division sont des opérations plus complexes, donc en général le compilateur embarque des appels à des fonctions de bibliothèque les effectuant.

Ces fonctions sont décrites ici: **??** on page ??.

Listing 1.376: GCC 4.8.1 -fno-inline avec optimisation

```
_f_mul :
        push    ebx
        mov     edx, DWORD PTR [esp+8]
        mov     eax, DWORD PTR [esp+16]
        mov     ebx, DWORD PTR [esp+12]
        mov     ecx, DWORD PTR [esp+20]
        imul    ebx, eax
        imul    ecx, edx
        mul     edx
        add     ecx, ebx
        add     edx, ecx
        pop     ebx
        ret

_f_div :
        sub     esp, 28
        mov     eax, DWORD PTR [esp+40]
        mov     edx, DWORD PTR [esp+44]
        mov     DWORD PTR [esp+8], eax
        mov     eax, DWORD PTR [esp+32]
        mov     DWORD PTR [esp+12], edx
        mov     edx, DWORD PTR [esp+36]
        mov     DWORD PTR [esp], eax
        mov     DWORD PTR [esp+4], edx
        call    ___udivdi3 ; division non signé
        add     esp, 28
        ret

_f_rem :
        sub     esp, 28
        mov     eax, DWORD PTR [esp+40]
        mov     edx, DWORD PTR [esp+44]
        mov     DWORD PTR [esp+8], eax
        mov     eax, DWORD PTR [esp+32]
        mov     DWORD PTR [esp+12], edx
        mov     edx, DWORD PTR [esp+36]
        mov     DWORD PTR [esp], eax
        mov     DWORD PTR [esp+4], edx
        call    ___umoddi3 ; modulo non signé
        add     esp, 28
        ret
```

GCC fait ce que l'on attend, mais le code multiplication est mis en ligne (inlined) directement dans la fonction, pensant que ça peut être plus efficace. GCC a des noms de fonctions de bibliothèque différents: **??** on page ??.

ARM

Keil pour mode Thumb insère des appels à des sous-routines de bibliothèque:

Listing 1.377: avec optimisation Keil 6/2013 (Mode Thumb)

```
||f_mul|| PROC
        PUSH    {r4,lr}
        BL      __aeabi_lmul
        POP     {r4,pc}
        ENDP
```

```
||f_div|| PROC
        PUSH     {r4,lr}
        BL       __aeabi_uldivmod
        POP      {r4,pc}
        ENDP

||f_rem|| PROC
        PUSH     {r4,lr}
        BL       __aeabi_uldivmod
        MOVS     r0,r2
        MOVS     r1,r3
        POP      {r4,pc}
        ENDP
```

Keil pour mode ARM, d'un autre côté, est capable de produire le code de la multiplication 64-bit:

Listing 1.378: avec optimisation Keil 6/2013 (Mode ARM)

```
||f_mul|| PROC
        PUSH     {r4,lr}
        UMULL    r12,r4,r0,r2
        MLA      r1,r2,r1,r4
        MLA      r1,r0,r3,r1
        MOV      r0,r12
        POP      {r4,pc}
        ENDP

||f_div|| PROC
        PUSH     {r4,lr}
        BL       __aeabi_uldivmod
        POP      {r4,pc}
        ENDP

||f_rem|| PROC
        PUSH     {r4,lr}
        BL       __aeabi_uldivmod
        MOV      r0,r2
        MOV      r1,r3
        POP      {r4,pc}
        ENDP
```

MIPS

GCC avec optimisation pour MIPS peut générer du code pour la multiplication 64-bit, mais doit appeler une routine de bibliothèque pour la division 64-bit:

Listing 1.379: GCC 4.4.5 avec optimisation (IDA)

```
f_mul :
       mult    $a2, $a1
       mflo    $v0
       or      $at, $zero  ; NOP
       or      $at, $zero  ; NOP
       mult    $a0, $a3
       mflo    $a0
       addu    $v0, $a0
       or      $at, $zero  ; NOP
       multu   $a3, $a1
       mfhi    $a2
       mflo    $v1
       jr      $ra
       addu    $v0, $a2

f_div :

var_10 = -0x10
var_4  = -4

       lui     $gp, (__gnu_local_gp >> 16)
```

```
        addiu   $sp, -0x20
        la      $gp, (__gnu_local_gp & 0xFFFF)
        sw      $ra, 0x20+var_4($sp)
        sw      $gp, 0x20+var_10($sp)
        lw      $t9, (__udivdi3 & 0xFFFF)($gp)
        or      $at, $zero
        jalr    $t9
        or      $at, $zero
        lw      $ra, 0x20+var_4($sp)
        or      $at, $zero
        jr      $ra
        addiu   $sp, 0x20

f_rem :

var_10 = -0x10
var_4  = -4

        lui     $gp, (__gnu_local_gp >> 16)
        addiu   $sp, -0x20
        la      $gp, (__gnu_local_gp & 0xFFFF)
        sw      $ra, 0x20+var_4($sp)
        sw      $gp, 0x20+var_10($sp)
        lw      $t9, (__umoddi3 & 0xFFFF)($gp)
        or      $at, $zero
        jalr    $t9
        or      $at, $zero
        lw      $ra, 0x20+var_4($sp)
        or      $at, $zero
        jr      $ra
        addiu   $sp, 0x20
```

Il y a beaucoup de NOPs, sans doute des slots de délai de remplissage après l'instruction de multiplication (c'est plus lent que les autres instructions après tout).

1.28.4 Décalage à droite

```
#include <stdint.h>

uint64_t f (uint64_t a)
{
        return a>>7;
};
```

x86

Listing 1.380: MSVC 2012 /Ob1 avec optimisation

```
_a$ = 8         ; size = 8
_f      PROC
        mov     eax, DWORD PTR _a$[esp-4]
        mov     edx, DWORD PTR _a$[esp]
        shrd    eax, edx, 7
        shr     edx, 7
        ret     0
_f      ENDP
```

Listing 1.381: GCC 4.8.1 -fno-inline avec optimisation

```
_f :
        mov     edx, DWORD PTR [esp+8]
        mov     eax, DWORD PTR [esp+4]
        shrd    eax, edx, 7
        shr     edx, 7
        ret
```

Le décalage se produit en deux passes: tout d'abord la partie basse est décalée, puis la partie haute. Mais la partie basse est décalée avec l'aide de l'instruction SHRD, elle décale la valeur de EAX de 7 bits, mais tire les nouveaux bits de EDX, i.e., de la partie haute. En d'autres mots, la valeur 64-bit dans la paire de registres EDX:EAX, dans son entier, est décalée de 7 bits et les 32 bits bas du résultat sont placés dans EAX. La partie haute est décalée en utilisant l'instruction plus populaire SHR : en effet, les bits libérés dans la partie haute doivent être remplis avec des zéros.

ARM

ARM n'a pas une instruction telle que SHRD en x86, donc le compilateur Keil fait cela en utilisant des simples décalages et des opérations OR :

Listing 1.382: avec optimisation Keil 6/2013 (Mode ARM)

```
||f|| PROC
        LSR      r0,r0,#7
        ORR      r0,r0,r1,LSL #25
        LSR      r1,r1,#7
        BX       lr
        ENDP
```

Listing 1.383: avec optimisation Keil 6/2013 (Mode Thumb)

```
||f|| PROC
        LSLS     r2,r1,#25
        LSRS     r0,r0,#7
        ORRS     r0,r0,r2
        LSRS     r1,r1,#7
        BX       lr
        ENDP
```

MIPS

GCC pour MIPS suit le même algorithme que Keil fait pour le mode Thumb:

Listing 1.384: GCC 4.4.5 avec optimisation (IDA)

```
f :
       sll     $v0, $a0, 25
       srl     $v1, $a1, 7
       or      $v1, $v0, $v1
       jr      $ra
       srl     $v0, $a0, 7
```

1.28.5 Convertir une valeur 32-bit en 64-bit

```
#include <stdint.h>

int64_t f (int32_t a)
{
        return a;
};
```

x86

Listing 1.385: MSVC 2012 avec optimisation

```
_a$ = 8
_f      PROC
        mov     eax, DWORD PTR _a$[esp-4]
        cdq
        ret     0
_f      ENDP
```

Ici, nous nous heurtons à la nécessité d'étendre une valeur 32-bit signée en une 64-bit signée. Les valeurs non signées sont converties directement: tous les bits de la partie haute doivent être mis à 0. Mais ce n'est pas approprié pour les types de donnée signée: le signe doit être copié dans la partie haute du nombre résultant.

L'instruction CDQ fait cela ici, elle prend sa valeur d'entrée dans EAX, étend le signe sur 64-bit et laisse le résultat dans la paire de registres EDX :EAX. En d'autres mots, CDQ prend le signe du nombre dans EAX (en prenant le bit le plus significatif dans EAX), et suivant sa valeur, met tous les 32 bits de EDX à 0 ou 1. Cette opération est quelque peu similaire à l'instruction MOVSX.

ARM

Listing 1.386: avec optimisation Keil 6/2013 (Mode ARM)

```
||f|| PROC
        ASR      r1,r0,#31
        BX       lr
        ENDP
```

Keil pour ARM est différent: il décale simplement arithmétiquement de 31 bits vers la droite la valeur en entrée. Comme nous le savons, le bit de signe est le MSB, et le décalage arithmétique copie le bit de signe dans les bits « apparus ». Donc après « ASR r1,r0,#31 », R1 contient 0xFFFFFFFF si la valeur d'entrée était négative et 0 sinon. R1 contient la partie haute de la valeur 64-bit résultante. En d'autres mots, ce code copie juste le MSB (bit de signe) de la valeur d'entrée dans R0 dans tous les bits de la partie haute 32-bit de la valeur 64-bit résultante.

MIPS

GCC pour MIPS fait la même chose que Keil a fait pour le mode ARM:

Listing 1.387: GCC 4.4.5 avec optimisation (IDA)

```
f :
    sra     $v0, $a0, 31
    jr      $ra
    move    $v1, $a0
```

1.29 SIMD

SIMD est un acronyme: *Single Instruction, Multiple Data* (simple instruction, multiple données).

Comme son nom le laisse entendre, cela traite des données multiples avec une seule instruction.

Comme le FPU, ce sous-système du CPU ressemble à un processeur séparé à l'intérieur du x86.

SIMD a commencé avec le MMX en x86. 8 nouveaux registres apparurent: MM0-MM7.

Chaque registre MMX contient 2 valeurs 32-bit, 4 valeurs 16-bit ou 8 octets. Par exemple, il est possible d'ajouter 8 valeurs 8-bit (octets) simultanément en ajoutant deux valeurs dans des registres MMX.

Un exemple simple est un éditeur graphique qui représente une image comme un tableau à deux dimensions. Lorsque l'utilisateur change la luminosité de l'image, l'éditeur doit ajouter ou soustraire un coefficient à/de chaque valeur du pixel. Dans un soucis de concision, si l'on dit que l'image est en niveau de gris et que chaque pixel est défini par un octet de 8-bit, alors il est possible de changer la luminosité de 8 pixels simultanément.

À propos, c'est la raison pour laquelle les instructions de *saturation* sont présentes en SIMD.

Lorsque l'utilisateur change la luminosité dans l'éditeur graphique, les dépassements au dessus ou en dessous ne sont pas souhaitables, donc il y a des instructions d'addition en SIMD qui n'additionnent pas si la valeur maximum est atteinte, etc.

Lorsque le MMX est apparu, ces registres étaient situés dans les registres du FPU. Il était seulement possible d'utiliser soit le FPU ou soit le MMX. On peut penser qu'Intel économisait des transistors, mais en fait, la raison d'une telle symbiose était plus simple —les anciens OSes qui n'étaient pas au courant de

ces registres supplémentaires et ne les sauvaient pas lors du changement de contexte, mais sauvaient les registres FPU. Ainsi, CPU avec MMX + ancien OS + processus utilisant les capacités MMX fonctionnait toujours.

SSE—est une extension des registres SIMD à 128 bits, maintenant séparé du FPU.

AVX—une autre extension, à 256 bits.

Parlons maintenant de l'usage pratique.

Bien sûr, il s'agit de routines de copie en mémoire (`memcpy`), de comparaison de mémoire (`memcmp`) et ainsi de suite.

Un autre exemple: l'algorithme de chiffrement DES prend un bloc de 64-bit et une clef de 56-bit, chiffre le bloc et produit un résultat de 64-bit. L'algorithme DES peut être considéré comme un grand circuit électronique, avec des fils et des portes AND/OR/NOT.

Le bitslice DES[181] —est l'idée de traiter des groupes de blocs et de clés simultanément. Disons, une variable de type *unsigned int* en x86 peut contenir jusqu'à 32-bit, donc il est possible d'y stocker des résultats intermédiaires pour 32 paires de blocs-clé simultanément, en utilisant 64+56 variables de type *unsigned int*.

Il existe un utilitaire pour brute-forcer les mots de passe/hashes d'Oracle RDBMS (certains basés sur DES) en utilisant un algorithme bitslice DES légèrement modifié pour SSE2 et AVX—maintenant il est possible de chiffrer 128 ou 256 paires de blocs-clé simultanément.

http://go.yurichev.com/17313

1.29.1 Vectorisation

La vectorisation[182], c'est lorsque, par exemple, vous avez une boucle qui prend une paire de tableaux en entrée et produit un tableau. Le corps de la boucle prend les valeurs dans les tableaux en entrée, fait quelque chose et met le résultat dans le tableau de sortie. La vectorisation est le fait de traiter plusieurs éléments simultanément.

La vectorisation n'est pas une nouvelle technologie: l'auteur de ce livre l'a vu au moins sur la série du super-calculateur Cray Y-MP de 1988 lorsqu'il jouait avec sa version « lite » le Cray Y-MP EL[183].

Par exemple:

```
for (i = 0; i < 1024; i++)
{
    C[i] = A[i]*B[i];
}
```

Ce morceau de code prend des éléments de A et de B, les multiplie et sauve le résultat dans C.

Si chaque élément du tableau que nous avons est un *int* 32-bit, alors il est possible de charger 4 éléments de A dans un registre XMM 128-bit, 4 de B dans un autre registre XMM, et en exécutant *PMULLD* (*Multiply Packed Signed Dword Integers and Store Low Result*) et `PMULHW` (*Multiply Packed Signed Integers and Store High Result*), il est possible d'obtenir 4 produits 64-bit en une fois.

Ainsi, le nombre d'exécution du corps de la boucle est $1024/4$ au lieu de 1024, ce qui est 4 fois moins et, bien sûr, est plus rapide.

Exemple d'addition

Certains compilateurs peuvent effectuer la vectorisation automatiquement dans des cas simples, e.g., Intel C++[184].

Voici une fonction minuscule:

```
int f (int sz, int *ar1, int *ar2, int *ar3)
{
        for (int i=0; i<sz; i++)
```

[181] http://go.yurichev.com/17329
[182] Wikipédia: vectorisation
[183] À distance. Il est installé dans le musée des super-calculateurs: http://go.yurichev.com/17081
[184] Sur la vectorisation automatique d'Intel C++: Extrait: Vectorisation automatique efficace

```
                ar3[i]=ar1[i]+ar2[i];

        return 0;
};
```

Intel C++

Compilons la avec Intel C++ 11.1.051 win32:

`icl intel.cpp /QaxSSE2 /Faintel.asm /Ox`

Nous obtenons (dans IDA):

```
; int __cdecl f(int, int *, int *, int *)
                public ?f@@YAHHPAH00@Z
?f@@YAHHPAH00@Z proc near

var_10 = dword ptr -10h
sz     = dword ptr  4
ar1    = dword ptr  8
ar2    = dword ptr  0Ch
ar3    = dword ptr  10h

        push    edi
        push    esi
        push    ebx
        push    esi
        mov     edx, [esp+10h+sz]
        test    edx, edx
        jle     loc_15B
        mov     eax, [esp+10h+ar3]
        cmp     edx, 6
        jle     loc_143
        cmp     eax, [esp+10h+ar2]
        jbe     short loc_36
        mov     esi, [esp+10h+ar2]
        sub     esi, eax
        lea     ecx, ds :0[edx*4]
        neg     esi
        cmp     ecx, esi
        jbe     short loc_55

loc_36 : ; CODE XREF : f(int,int *,int *,int *)+21
        cmp     eax, [esp+10h+ar2]
        jnb     loc_143
        mov     esi, [esp+10h+ar2]
        sub     esi, eax
        lea     ecx, ds :0[edx*4]
        cmp     esi, ecx
        jb      loc_143

loc_55 : ; CODE XREF : f(int,int *,int *,int *)+34
        cmp     eax, [esp+10h+ar1]
        jbe     short loc_67
        mov     esi, [esp+10h+ar1]
        sub     esi, eax
        neg     esi
        cmp     ecx, esi
        jbe     short loc_7F

loc_67 : ; CODE XREF : f(int,int *,int *,int *)+59
        cmp     eax, [esp+10h+ar1]
        jnb     loc_143
        mov     esi, [esp+10h+ar1]
        sub     esi, eax
        cmp     esi, ecx
        jb      loc_143
```

```
loc_7F : ; CODE XREF : f(int,int *,int *,int *)+65
       mov     edi, eax        ; edi = ar3
       and     edi, 0Fh        ; est-ce que ar3 est aligné sur 16-octets ?
       jz      short loc_9A    ; oui
       test    edi, 3
       jnz     loc_162
       neg     edi
       add     edi, 10h
       shr     edi, 2

loc_9A : ; CODE XREF : f(int,int *,int *,int *)+84
       lea     ecx, [edi+4]
       cmp     edx, ecx
       jl      loc_162
       mov     ecx, edx
       sub     ecx, edi
       and     ecx, 3
       neg     ecx
       add     ecx, edx
       test    edi, edi
       jbe     short loc_D6
       mov     ebx, [esp+10h+ar2]
       mov     [esp+10h+var_10], ecx
       mov     ecx, [esp+10h+ar1]
       xor     esi, esi

loc_C1 : ; CODE XREF : f(int,int *,int *,int *)+CD
       mov     edx, [ecx+esi*4]
       add     edx, [ebx+esi*4]
       mov     [eax+esi*4], edx
       inc     esi
       cmp     esi, edi
       jb      short loc_C1
       mov     ecx, [esp+10h+var_10]
       mov     edx, [esp+10h+sz]

loc_D6 : ; CODE XREF : f(int,int *,int *,int *)+B2
       mov     esi, [esp+10h+ar2]
       lea     esi, [esi+edi*4] ; est-ce que ar2+i*4 est aligné sur 16-octets ?
       test    esi, 0Fh
       jz      short loc_109   ; oui !
       mov     ebx, [esp+10h+ar1]
       mov     esi, [esp+10h+ar2]

loc_ED : ; CODE XREF : f(int,int *,int *,int *)+105
       movdqu  xmm1, xmmword ptr [ebx+edi*4] ; ar1+i*4
       movdqu  xmm0, xmmword ptr [esi+edi*4] ; ar2+i*4 n'est pas aligné sur 16-octet, donc le ↙
    ↳ charger dans XMM0
       paddd   xmm1, xmm0
       movdqa  xmmword ptr [eax+edi*4], xmm1 ; ar3+i*4
       add     edi, 4
       cmp     edi, ecx
       jb      short loc_ED
       jmp     short loc_127

loc_109 : ; CODE XREF : f(int,int *,int *,int *)+E3
       mov     ebx, [esp+10h+ar1]
       mov     esi, [esp+10h+ar2]

loc_111 : ; CODE XREF : f(int,int *,int *,int *)+125
       movdqu  xmm0, xmmword ptr [ebx+edi*4]
       paddd   xmm0, xmmword ptr [esi+edi*4]
       movdqa  xmmword ptr [eax+edi*4], xmm0
       add     edi, 4
       cmp     edi, ecx
       jb      short loc_111

loc_127 : ; CODE XREF : f(int,int *,int *,int *)+107
         ; f(int,int *,int *,int *)+164
       cmp     ecx, edx
```

```
        jnb     short loc_15B
        mov     esi, [esp+10h+ar1]
        mov     edi, [esp+10h+ar2]

loc_133 : ; CODE XREF : f(int,int *,int *,int *)+13F
        mov     ebx, [esi+ecx*4]
        add     ebx, [edi+ecx*4]
        mov     [eax+ecx*4], ebx
        inc     ecx
        cmp     ecx, edx
        jb      short loc_133
        jmp     short loc_15B

loc_143 : ; CODE XREF : f(int,int *,int *,int *)+17
          ; f(int,int *,int *,int *)+3A ...
        mov     esi, [esp+10h+ar1]
        mov     edi, [esp+10h+ar2]
        xor     ecx, ecx

loc_14D : ; CODE XREF : f(int,int *,int *,int *)+159
        mov     ebx, [esi+ecx*4]
        add     ebx, [edi+ecx*4]
        mov     [eax+ecx*4], ebx
        inc     ecx
        cmp     ecx, edx
        jb      short loc_14D

loc_15B : ; CODE XREF : f(int,int *,int *,int *)+A
          ; f(int,int *,int *,int *)+129 ...
        xor     eax, eax
        pop     ecx
        pop     ebx
        pop     esi
        pop     edi
        retn

loc_162 : ; CODE XREF : f(int,int *,int *,int *)+8C
          ; f(int,int *,int *,int *)+9F
        xor     ecx, ecx
        jmp     short loc_127
?f@@YAHHPAH00@Z endp
```

Les instructions relatives à SSE2 sont:

- MOVDQU (*Move Unaligned Double Quadword* déplacer double quadruple mot non alignés)—charge juste 16 octets depuis la mémoire dans un registre XMM.
- PADDD (*Add Packed Integers* ajouter entier packé)—ajoute 4 paires de nombres 32-bit et laisse le résultat dans le premier opérande. À propos, aucune exception n'est levée en cas de débordement et aucun flag n'est mis, seuls les 32-bit bas du résultat sont stockés. Si un des opérandes de PADDD est l'adresse d'une valeur en mémoire, alors l'adresse doit être alignée sur une limite de 16 octets. Si elle n'est pas alignée, une exception est levée[185].
- MOVDQA (*Move Aligned Double Quadword*) est la même chose que MOVDQU, mais nécessite que l'adresse de la valeur en mémoire soit alignée sur une limite de 16 octets. Si elle n'est pas alignée, une exception est levée. MOVDQA fonctionne plus vite que MOVDQU, mais nécessite la condition qui vient d'être écrite.

Donc, ces instructions SSE2 sont exécutées seulement dans le cas où il y a plus de 4 paires à traiter et que le pointeur ar3 est aligné sur une limite de 16 octets.

Ainsi, si ar2 est également aligné sur une limite de 16 octets, ce morceau de code sera exécuté:

```
movdqu  xmm0, xmmword ptr [ebx+edi*4] ; ar1+i*4
paddd   xmm0, xmmword ptr [esi+edi*4] ; ar2+i*4
movdqa  xmmword ptr [eax+edi*4], xmm0 ; ar3+i*4
```

Autrement, la valeur de ar2 est chargée dans XMM0 avec MOVDQU, qui ne nécessite pas que le pointeur soit aligné, mais peut s'exécuter plus lentement.

[185] En savoir plus sur l'alignement des données: Wikipedia: Alignement en mémoire

```
movdqu  xmm1, xmmword ptr [ebx+edi*4] ; ar1+i*4
movdqu  xmm0, xmmword ptr [esi+edi*4] ; ar2+i*4 n'est pas aligné sur 16-octet, donc le charger ↙
    ↳ dans XMM0
paddd   xmm1, xmm0
movdqa  xmmword ptr [eax+edi*4], xmm1 ; ar3+i*4
```

Dans tous les autres cas, le code non-SSE2 sera exécuté.

GCC

GCC peut aussi vectoriser dans des cas simples[186], si l'option `-O3` est utilisée et le support de SSE2 activé: `-msse2`.

Ce que nous obtenons (GCC 4.4.1):

```
; f(int, int *, int *, int *)
            public _Z1fiPiS_S_
_Z1fiPiS_S_ proc near

var_18      = dword ptr -18h
var_14      = dword ptr -14h
var_10      = dword ptr -10h
arg_0       = dword ptr  8
arg_4       = dword ptr  0Ch
arg_8       = dword ptr  10h
arg_C       = dword ptr  14h

            push    ebp
            mov     ebp, esp
            push    edi
            push    esi
            push    ebx
            sub     esp, 0Ch
            mov     ecx, [ebp+arg_0]
            mov     esi, [ebp+arg_4]
            mov     edi, [ebp+arg_8]
            mov     ebx, [ebp+arg_C]
            test    ecx, ecx
            jle     short loc_80484D8
            cmp     ecx, 6
            lea     eax, [ebx+10h]
            ja      short loc_80484E8

loc_80484C1 : ; CODE XREF : f(int,int *,int *,int *)+4B
              ; f(int,int *,int *,int *)+61 ...
            xor     eax, eax
            nop
            lea     esi, [esi+0]

loc_80484C8 : ; CODE XREF : f(int,int *,int *,int *)+36
            mov     edx, [edi+eax*4]
            add     edx, [esi+eax*4]
            mov     [ebx+eax*4], edx
            add     eax, 1
            cmp     eax, ecx
            jnz     short loc_80484C8

loc_80484D8 : ; CODE XREF : f(int,int *,int *,int *)+17
              ; f(int,int *,int *,int *)+A5
            add     esp, 0Ch
            xor     eax, eax
            pop     ebx
            pop     esi
            pop     edi
            pop     ebp
            retn
```

[186] Plus sur le support de la vectorisation dans GCC: http://go.yurichev.com/17083

```
            align 8

loc_80484E8 : ; CODE XREF : f(int,int *,int *,int *)+1F
            test    bl, 0Fh
            jnz     short loc_80484C1
            lea     edx, [esi+10h]
            cmp     ebx, edx
            jbe     loc_8048578

loc_80484F8 : ; CODE XREF : f(int,int *,int *,int *)+E0
            lea     edx, [edi+10h]
            cmp     ebx, edx
            ja      short loc_8048503
            cmp     edi, eax
            jbe     short loc_80484C1

loc_8048503 : ; CODE XREF : f(int,int *,int *,int *)+5D
            mov     eax, ecx
            shr     eax, 2
            mov     [ebp+var_14], eax
            shl     eax, 2
            test    eax, eax
            mov     [ebp+var_10], eax
            jz      short loc_8048547
            mov     [ebp+var_18], ecx
            mov     ecx, [ebp+var_14]
            xor     eax, eax
            xor     edx, edx
            nop

loc_8048520 : ; CODE XREF : f(int,int *,int *,int *)+9B
            movdqu  xmm1, xmmword ptr [edi+eax]
            movdqu  xmm0, xmmword ptr [esi+eax]
            add     edx, 1
            paddd   xmm0, xmm1
            movdqa  xmmword ptr [ebx+eax], xmm0
            add     eax, 10h
            cmp     edx, ecx
            jb      short loc_8048520
            mov     ecx, [ebp+var_18]
            mov     eax, [ebp+var_10]
            cmp     ecx, eax
            jz      short loc_80484D8

loc_8048547 : ; CODE XREF : f(int,int *,int *,int *)+73
            lea     edx, ds :0[eax*4]
            add     esi, edx
            add     edi, edx
            add     ebx, edx
            lea     esi, [esi+0]

loc_8048558 : ; CODE XREF : f(int,int *,int *,int *)+CC
            mov     edx, [edi]
            add     eax, 1
            add     edi, 4
            add     edx, [esi]
            add     esi, 4
            mov     [ebx], edx
            add     ebx, 4
            cmp     ecx, eax
            jg      short loc_8048558
            add     esp, 0Ch
            xor     eax, eax
            pop     ebx
            pop     esi
            pop     edi
            pop     ebp
            retn
```

```
loc_8048578 : ; CODE XREF : f(int,int *,int *,int *)+52
            cmp     eax, esi
            jnb     loc_80484C1
            jmp     loc_80484F8
_Z1fiPiS_S_ endp
```

Presque le même, toutefois, pas aussi méticuleux qu'Intel C++.

Exemple de copie de mémoire

Revoyons le simple exemple memcpy() (1.16.2 on page 195):

```
#include <stdio.h>

void my_memcpy (unsigned char* dst, unsigned char* src, size_t cnt)
{
        size_t i;
        for (i=0; i<cnt; i++)
                dst[i]=src[i];
};
```

Et ce que les optimisations de GCC 4.9.1 font:

Listing 1.388: GCC 4.9.1 x64 avec optimisation

```
my_memcpy :
; RDI = adresse de destination
; RSI = adresse source
; RDX = taille du bloc
        test    rdx, rdx
        je      .L41
        lea     rax, [rdi+16]
        cmp     rsi, rax
        lea     rax, [rsi+16]
        setae   cl
        cmp     rdi, rax
        setae   al
        or      cl, al
        je      .L13
        cmp     rdx, 22
        jbe     .L13
        mov     rcx, rsi
        push    rbp
        push    rbx
        neg     rcx
        and     ecx, 15
        cmp     rcx, rdx
        cmova   rcx, rdx
        xor     eax, eax
        test    rcx, rcx
        je      .L4
        movzx   eax, BYTE PTR [rsi]
        cmp     rcx, 1
        mov     BYTE PTR [rdi], al
        je      .L15
        movzx   eax, BYTE PTR [rsi+1]
        cmp     rcx, 2
        mov     BYTE PTR [rdi+1], al
        je      .L16
        movzx   eax, BYTE PTR [rsi+2]
        cmp     rcx, 3
        mov     BYTE PTR [rdi+2], al
        je      .L17
        movzx   eax, BYTE PTR [rsi+3]
        cmp     rcx, 4
        mov     BYTE PTR [rdi+3], al
        je      .L18
        movzx   eax, BYTE PTR [rsi+4]
        cmp     rcx, 5
```

```
        mov     BYTE PTR [rdi+4], al
        je      .L19
        movzx   eax, BYTE PTR [rsi+5]
        cmp     rcx, 6
        mov     BYTE PTR [rdi+5], al
        je      .L20
        movzx   eax, BYTE PTR [rsi+6]
        cmp     rcx, 7
        mov     BYTE PTR [rdi+6], al
        je      .L21
        movzx   eax, BYTE PTR [rsi+7]
        cmp     rcx, 8
        mov     BYTE PTR [rdi+7], al
        je      .L22
        movzx   eax, BYTE PTR [rsi+8]
        cmp     rcx, 9
        mov     BYTE PTR [rdi+8], al
        je      .L23
        movzx   eax, BYTE PTR [rsi+9]
        cmp     rcx, 10
        mov     BYTE PTR [rdi+9], al
        je      .L24
        movzx   eax, BYTE PTR [rsi+10]
        cmp     rcx, 11
        mov     BYTE PTR [rdi+10], al
        je      .L25
        movzx   eax, BYTE PTR [rsi+11]
        cmp     rcx, 12
        mov     BYTE PTR [rdi+11], al
        je      .L26
        movzx   eax, BYTE PTR [rsi+12]
        cmp     rcx, 13
        mov     BYTE PTR [rdi+12], al
        je      .L27
        movzx   eax, BYTE PTR [rsi+13]
        cmp     rcx, 15
        mov     BYTE PTR [rdi+13], al
        jne     .L28
        movzx   eax, BYTE PTR [rsi+14]
        mov     BYTE PTR [rdi+14], al
        mov     eax, 15
.L4 :
        mov     r10, rdx
        lea     r9, [rdx-1]
        sub     r10, rcx
        lea     r8, [r10-16]
        sub     r9, rcx
        shr     r8, 4
        add     r8, 1
        mov     r11, r8
        sal     r11, 4
        cmp     r9, 14
        jbe     .L6
        lea     rbp, [rsi+rcx]
        xor     r9d, r9d
        add     rcx, rdi
        xor     ebx, ebx
.L7 :
        movdqa  xmm0, XMMWORD PTR [rbp+0+r9]
        add     rbx, 1
        movups  XMMWORD PTR [rcx+r9], xmm0
        add     r9, 16
        cmp     rbx, r8
        jb      .L7
        add     rax, r11
        cmp     r10, r11
        je      .L1
.L6 :
        movzx   ecx, BYTE PTR [rsi+rax]
        mov     BYTE PTR [rdi+rax], cl
```

```
        lea     rcx, [rax+1]
        cmp     rdx, rcx
        jbe     .L1
        movzx   ecx, BYTE PTR [rsi+1+rax]
        mov     BYTE PTR [rdi+1+rax], cl
        lea     rcx, [rax+2]
        cmp     rdx, rcx
        jbe     .L1
        movzx   ecx, BYTE PTR [rsi+2+rax]
        mov     BYTE PTR [rdi+2+rax], cl
        lea     rcx, [rax+3]
        cmp     rdx, rcx
        jbe     .L1
        movzx   ecx, BYTE PTR [rsi+3+rax]
        mov     BYTE PTR [rdi+3+rax], cl
        lea     rcx, [rax+4]
        cmp     rdx, rcx
        jbe     .L1
        movzx   ecx, BYTE PTR [rsi+4+rax]
        mov     BYTE PTR [rdi+4+rax], cl
        lea     rcx, [rax+5]
        cmp     rdx, rcx
        jbe     .L1
        movzx   ecx, BYTE PTR [rsi+5+rax]
        mov     BYTE PTR [rdi+5+rax], cl
        lea     rcx, [rax+6]
        cmp     rdx, rcx
        jbe     .L1
        movzx   ecx, BYTE PTR [rsi+6+rax]
        mov     BYTE PTR [rdi+6+rax], cl
        lea     rcx, [rax+7]
        cmp     rdx, rcx
        jbe     .L1
        movzx   ecx, BYTE PTR [rsi+7+rax]
        mov     BYTE PTR [rdi+7+rax], cl
        lea     rcx, [rax+8]
        cmp     rdx, rcx
        jbe     .L1
        movzx   ecx, BYTE PTR [rsi+8+rax]
        mov     BYTE PTR [rdi+8+rax], cl
        lea     rcx, [rax+9]
        cmp     rdx, rcx
        jbe     .L1
        movzx   ecx, BYTE PTR [rsi+9+rax]
        mov     BYTE PTR [rdi+9+rax], cl
        lea     rcx, [rax+10]
        cmp     rdx, rcx
        jbe     .L1
        movzx   ecx, BYTE PTR [rsi+10+rax]
        mov     BYTE PTR [rdi+10+rax], cl
        lea     rcx, [rax+11]
        cmp     rdx, rcx
        jbe     .L1
        movzx   ecx, BYTE PTR [rsi+11+rax]
        mov     BYTE PTR [rdi+11+rax], cl
        lea     rcx, [rax+12]
        cmp     rdx, rcx
        jbe     .L1
        movzx   ecx, BYTE PTR [rsi+12+rax]
        mov     BYTE PTR [rdi+12+rax], cl
        lea     rcx, [rax+13]
        cmp     rdx, rcx
        jbe     .L1
        movzx   ecx, BYTE PTR [rsi+13+rax]
        mov     BYTE PTR [rdi+13+rax], cl
        lea     rcx, [rax+14]
        cmp     rdx, rcx
        jbe     .L1
        movzx   edx, BYTE PTR [rsi+14+rax]
        mov     BYTE PTR [rdi+14+rax], dl
```

```
.L1 :
        pop     rbx
        pop     rbp
.L41 :
        rep ret
.L13 :
        xor     eax, eax
.L3 :
        movzx   ecx, BYTE PTR [rsi+rax]
        mov     BYTE PTR [rdi+rax], cl
        add     rax, 1
        cmp     rax, rdx
        jne     .L3
        rep ret
.L28 :
        mov     eax, 14
        jmp     .L4
.L15 :
        mov     eax, 1
        jmp     .L4
.L16 :
        mov     eax, 2
        jmp     .L4
.L17 :
        mov     eax, 3
        jmp     .L4
.L18 :
        mov     eax, 4
        jmp     .L4
.L19 :
        mov     eax, 5
        jmp     .L4
.L20 :
        mov     eax, 6
        jmp     .L4
.L21 :
        mov     eax, 7
        jmp     .L4
.L22 :
        mov     eax, 8
        jmp     .L4
.L23 :
        mov     eax, 9
        jmp     .L4
.L24 :
        mov     eax, 10
        jmp     .L4
.L25 :
        mov     eax, 11
        jmp     .L4
.L26 :
        mov     eax, 12
        jmp     .L4
.L27 :
        mov     eax, 13
        jmp     .L4
```

1.29.2 Implémentation SIMD de `strlen()`

Il faut noter que les instructions SIMD peuvent être insérées en code C/C++ via des macros[187] spécifiques. Pour MSVC, certaines d'entre elles se trouvent dans le fichier `intrin.h`.

Il est possible d'implémenter la fonction `strlen()`[188] en utilisant des instructions SIMD qui fonctionne 2-2.5 fois plus vite que l'implémentation habituelle. Cette fonction charge 16 caractères dans un registre

[187] MSDN: particularités MMX, SSE, et SSE2

[188] strlen() —fonction de la bibliothèque C standard pour calculer la longueur d'une chaîne

XMM et teste chacun d'eux avec zéro. [189].

```
size_t strlen_sse2(const char *str)
{
    register size_t len = 0;
    const char *s=str;
    bool str_is_aligned=(((unsigned int)str)&0xFFFFFFF0) == (unsigned int)str;

    if (str_is_aligned==false)
        return strlen (str);

    __m128i xmm0 = _mm_setzero_si128();
    __m128i xmm1;
    int mask = 0;

    for (;;)
    {
        xmm1 = _mm_load_si128((__m128i *)s);
        xmm1 = _mm_cmpeq_epi8(xmm1, xmm0);
        if ((mask = _mm_movemask_epi8(xmm1)) != 0)
        {
            unsigned long pos;
            _BitScanForward(&pos, mask);
            len += (size_t)pos;

            break;
        }
        s += sizeof(__m128i);
        len += sizeof(__m128i);
    };

    return len;
}
```

Compilons la avec MSVC 2010 avec l'option /0x :

Listing 1.389: MSVC 2010 avec optimisation

```
_pos$75552 = -4          ; taille = 4
_str$ = 8                ; taille = 4
?strlen_sse2@@YAIPBD@Z PROC ; strlen_sse2

    push     ebp
    mov      ebp, esp
    and      esp, -16            ; fffffff0H
    mov      eax, DWORD PTR _str$[ebp]
    sub      esp, 12             ; 0000000cH
    push     esi
    mov      esi, eax
    and      esi, -16            ; fffffff0H
    xor      edx, edx
    mov      ecx, eax
    cmp      esi, eax
    je       SHORT $LN4@strlen_sse
    lea      edx, DWORD PTR [eax+1]
    npad     3 ; aligner le prochain label
$LL11@strlen_sse :
    mov      cl, BYTE PTR [eax]
    inc      eax
    test     cl, cl
    jne      SHORT $LL11@strlen_sse
    sub      eax, edx
    pop      esi
    mov      esp, ebp
    pop      ebp
    ret      0
$LN4@strlen_sse :
    movdqa   xmm1, XMMWORD PTR [eax]
    pxor     xmm0, xmm0
```

[189] L'exemple est basé sur le code source de: http://go.yurichev.com/17330.

```
    pcmpeqb  xmm1, xmm0
    pmovmskb eax, xmm1
    test     eax, eax
    jne      SHORT $LN9@strlen_sse
$LL3@strlen_sse :
    movdqa   xmm1, XMMWORD PTR [ecx+16]
    add      ecx, 16                    ; 00000010H
    pcmpeqb  xmm1, xmm0
    add      edx, 16                    ; 00000010H
    pmovmskb eax, xmm1
    test     eax, eax
    je       SHORT $LL3@strlen_sse
$LN9@strlen_sse :
    bsf      eax, eax
    mov      ecx, eax
    mov      DWORD PTR _pos$75552[esp+16], eax
    lea      eax, DWORD PTR [ecx+edx]
    pop      esi
    mov      esp, ebp
    pop      ebp
    ret      0
?strlen_sse2@@YAIPBD@Z ENDP                ; strlen_sse2
```

Comment est-ce que ça fonctionne? Premièrement, nous devons comprendre la but de la fonction. Elle calcule la longueur de la chaîne C, mais nous pouvons utiliser différents termes: sa tâche est de chercher l'octet zéro, et de calculer sa position relativement au début de la chaîne.

Premièrement, nous testons si le pointeur `str` est aligné sur une limite de 16 octets. Si non, nous appelons l'implémentation générique de `strlen()`.

Puis, nous chargeons les 16 octets suivants dans le registre `XMM1` en utilisant `MOVDQA`.

Un lecteur observateur pourrait demander, pourquoi `MOVDQU` ne pourrait pas être utilisé ici, puisqu'il peut charger des données depuis la mémoire quelque soit l'alignement du pointeur?

Oui, cela pourrait être fait comme ça: si le pointeur est aligné, charger les données en utilisant `MOVDQA`, si non —utiliser `MOVDQU` moins rapide.

Mais ici nous pourrions rencontrer une autre restriction:

Dans la série d'OS Windows NT (mais pas seulement), la mémoire est allouée par pages de 4 KiB (4096 octets). Chaque processus win32 a 4GiB de disponible, mais en fait, seulement une partie de l'espace d'adressage est connecté à de la mémoire réelle. Si le processus accède a un bloc mémoire absent, une exception est levée. C'est comme cela que fonctionnent les VM[190].

Donc, une fonction qui charge 16 octets à la fois peut dépasser la limite d'un bloc de mémoire allouée. Disons que l'OS a alloué 8192 (0x2000) octets à l'adresse 0x008c0000. Ainsi, le bloc comprend les octets démarrant à l'adresse 0x008c0000 jusqu'à 0x008c1fff inclus.

Après ce bloc, c'est à dire à partir de l'adresse 0x008c2000 il n'y a rien du tout, e.g. l'OS n'y a pas alloué de mémoire. Toutes tentatives d'accéder à la mémoire à partir de cette adresse va déclencher une exception.

Et maintenant, considérons l'exemple dans lequel le programme possède une chaîne contenant 5 caractères presque à la fin d'un bloc, et ce n'est pas un crime.

0x008c1ff8	'h'
0x008c1ff9	'e'
0x008c1ffa	'l'
0x008c1ffb	'l'
0x008c1ffc	'o'
0x008c1ffd	'\x00'
0x008c1ffe	random noise
0x008c1fff	random noise

Donc, dans des conditions normales, le programme appelle `strlen()`, en lui passant un pointeur sur la chaîne `'hello'` se trouvant en mémoire à l'adresse 0x008c1ff8. `strlen()` lit un octet à la fois jusqu'à 0x008c1ffd, où se trouve un octet à zéro, et puis s'arrête.

Maintenant, si nous implémentons notre propre `strlen()` lisant 16 octets à la fois, à partir de n'importe quelle adresse, alignée ou pas, `MOVDQU` pourrait essayer de charger 16 octets à la fois depuis l'adresse

[190] Wikipédia

0x008c1ff8 jusqu'à 0x008c2008, et ainsi déclencherait une exception. Cette situation peut être évitée, bien sûr.

Nous allons donc ne travailler qu'avec des adresses alignées sur une limite de 16 octets, ce qui en combinaison avec la connaissance que les pages de l'OS sont en général alignées sur une limite de 16 octets nous donne quelques garanties que notre fonction ne va pas lire de la mémoire non allouée.

Retournons à notre fonction.

`_mm_setzero_si128()`—est une macro générant `pxor xmm0, xmm0` —elle efface juste le registre XMM0.

`_mm_load_si128()`—est une macro pour MOVDQA, elle charge 16 octets depuis l'adresse dans le registre XMM1.

`_mm_cmpeq_epi8()`—est une macro pour PCMPEQB, une instruction qui compare deux registres XMM par octet.

Et si l'un des octets est égal à celui dans l'autre registre, il y aura `0xff` à ce point dans le résultat ou 0 sinon.

Par exemple:

```
XMM1: 0x11223344556677880000000000000000
XMM0: 0x11ab3444007877881111111111111111
```

Après l'exécution de `pcmpeqb xmm1, xmm0`, le registre XMM1 contient:

```
XMM1: 0xff0000ff0000ffff0000000000000000
```

Dans notre cas, cette instruction compare chacun des 16 octets avec un bloc de 16 octets à zéro, qui ont été mis dans le registre XMM0 par `pxor xmm0, xmm0`.

La macro suivante est `_mm_movemask_epi8()` —qui est l'instruction PMOVMSKB.

Elle est très utile avec PCMPEQB.

```
pmovmskb eax, xmm1
```

Cette instruction met d'abord le premier bit d'EAX à 1 si le bit le plus significatif du premier octet dans XMM1 est 1. En d'autres mots, si le premier octet du registre XMM1 est `0xff`, alors le premier bit de EAX sera 1 aussi.

Si le second octet du registre XMM1 est `0xff`, alors le second bit de EAX sera mis à 1 aussi. En d'autres mots, cette instruction répond à la question « quels octets de XMM1 ont le bit le plus significatif à 1 ou sont plus grand que `0x7f` ? » et renvoie 16 bits dans le registre EAX. Les autres bits du registre EAX sont mis à zéro.

À propos, ne pas oublier cette bizarrerie dans notre algorithme. Il pourrait y avoir 16 octets dans l'entrée, comme:

15	14	13	12	11	10	9 ... 3	2	1	0
'h'	'e'	'l'	'l'	'o'	0	déchets	0	déchets	

Il s'agit de la chaîne `'hello'`, terminée par un zéro, et du bruit aléatoire dans la mémoire.

Si nous chargeons ces 16 octets dans XMM1 et les comparons avec ceux à zéro dans XMM0, nous obtenons quelque chose comme [191] :

```
XMM1: 0x0000ff00000000000000ff0000000000
```

Cela signifie que cette instruction a trouvé deux octets à zéro, et ce n'est pas surprenant.

PMOVMSKB dans notre cas va mettre EAX à *0b0010000000100000*.

Bien sûr, notre fonction doit seulement prendre le premier octet à zéro et ignorer le reste.

L'instruction suivante est BSF (*Bit Scan Forward*).

Cette instruction trouve le premier bit mis à 1 et met sa position dans le premier opérande.

```
EAX=0b0010000000100000
```

Après l'exécution de `bsf eax, eax`, EAX contient 5, signifiant que 1 a été trouvé au bit d'index 5 (en commençant à zéro).

MSVC a une macro pour cette instruction: `_BitScanForward`.

[191] Un ordre de MSB à LSB[192] est utilisé ici.

Maintenant c'est simple. Si un octet à zéro a été trouvé, sa position est ajoutée à ce que nous avions déjà compté et nous pouvons renvoyer le résultat.

Presque tout.

À propos, il faut aussi noter que le compilateur MSVC a généré deux corps de boucle côte-à-côte, afin d'optimiser.

Et aussi, SSE 4.2 (apparu dans les Intel Core i7) offre encore plus d'instructions avec lesquelles ces manipulations de chaîne sont encore plus facile: http://go.yurichev.com/17331

1.30 64 bits

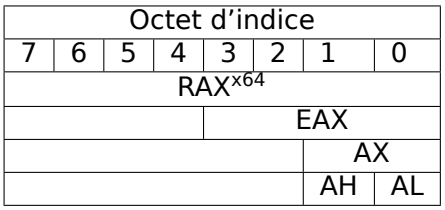
1.30.1 x86-64

Il s'agit d'une extension à 64 bits de l'architecture x86.

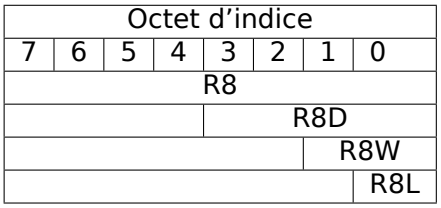
Pour l'ingénierie inverse, les changements les plus significatifs sont:

- La plupart des registres (à l'exception des registres FPU et SIMD) ont été étendus à 64 bits et leur nom préfixé de la lettre R. 8 registres ont également été ajoutés. Les GPR sont donc désormais: RAX, RBX, RCX, RDX, RBP, RSP, RSI, RDI, R8, R9, R10, R11, R12, R13, R14, R15.

  Les *anciens* registres restent accessibles de la manière habituelle. Ainsi, l'utilisation de EAX donne accès aux 32 bits de poids faible du registre RAX :



  Les nouveaux registres R8-R15 possèdent eux aussi des sous-parties : R8D-R15D (pour les 32 bits de poids faible), R8W-R15W (16 bits de poids faible), R8L-R15L (8 bits de poids faible).



  Les registres SIMD ont vu leur nombre passé de 8 à 16: XMM0-XMM15.

- En environnement Win64, la convention d'appel de fonctions est légèrement différente et ressemble à la convention fastcall (4.1.3 on page 545). Les 4 premiers arguments sont stockés dans les registres RCX, RDX, R8 et R9. Les arguments suivants —sur la pile. La fonction appelante doit également allouer 32 octets pour utilisation par la fonction appelée qui pourra y sauvegarder les registres contenant les 4 premiers arguments. Les fonctions les plus simples peuvent utiliser les arguments directement depuis les registres. En revanche, les fonctions plus complexes peuvent sauvegarder ces registres sur la pile.

  L'ABI System V AMD64 (Linux, *BSD, Mac OS X)[Michael Matz, Jan Hubicka, Andreas Jaeger, Mark Mitchell, *System V Application Binary Interface. AMD64 Architecture Processor Supplement*, (2013)] [193]ressemble elle aussi à la convention fastcall. Elle utilise 6 registres RDI, RSI, RDX, RCX, R8, R9 pour les 6 premiers arguments. Tous les suivants sont passés sur la pile.

  Référez-vous également à la section sur les conventions d'appel (4.1 on page 544).

- Pour des raisons de compatibilité, le type C/C++ *int* conserve sa taille de 32bits.

- Tous les pointeurs sont désormais sur 64 bits.

Dans la mesure où le nombre de registres a doublé, les compilateurs disposent de plus de marge de manœuvre en matière d'allocation des registres. Pour nous, il en résulte que le code généré contient moins de variables locales.

[193]Aussi disponible en https://software.intel.com/sites/default/files/article/402129/mpx-linux64-abi.pdf

Par exemple, la fonction qui calcule la première S-box de l'algorithme de chiffrage DES traite en une fois au moyen de la méthode DES bitslice des valeurs de 32/64/128/256 bits (en fonction du type DES_type (uint32, uint64, SSE2 or AVX)). Pour en savoir plus sur cette technique, voyez (1.29 on page 409):

```
/*
 * Generated S-box files.
 *
 * This software may be modified, redistributed, and used for any purpose,
 * so long as its origin is acknowledged.
 *
 * Produced by Matthew Kwan - March 1998
 */

#ifdef _WIN64
#define DES_type unsigned __int64
#else
#define DES_type unsigned int
#endif

void
s1 (
    DES_type    a1,
    DES_type    a2,
    DES_type    a3,
    DES_type    a4,
    DES_type    a5,
    DES_type    a6,
    DES_type    *out1,
    DES_type    *out2,
    DES_type    *out3,
    DES_type    *out4
) {
    DES_type    x1, x2, x3, x4, x5, x6, x7, x8;
    DES_type    x9, x10, x11, x12, x13, x14, x15, x16;
    DES_type    x17, x18, x19, x20, x21, x22, x23, x24;
    DES_type    x25, x26, x27, x28, x29, x30, x31, x32;
    DES_type    x33, x34, x35, x36, x37, x38, x39, x40;
    DES_type    x41, x42, x43, x44, x45, x46, x47, x48;
    DES_type    x49, x50, x51, x52, x53, x54, x55, x56;

    x1 = a3 & ~a5;
    x2 = x1 ^ a4;
    x3 = a3 & ~a4;
    x4 = x3 | a5;
    x5 = a6 & x4;
    x6 = x2 ^ x5;
    x7 = a4 & ~a5;
    x8 = a3 ^ a4;
    x9 = a6 & ~x8;
    x10 = x7 ^ x9;
    x11 = a2 | x10;
    x12 = x6 ^ x11;
    x13 = a5 ^ x5;
    x14 = x13 & x8;
    x15 = a5 & ~a4;
    x16 = x3 ^ x14;
    x17 = a6 | x16;
    x18 = x15 ^ x17;
    x19 = a2 | x18;
    x20 = x14 ^ x19;
    x21 = a1 & x20;
    x22 = x12 ^ ~x21;
    *out2 ^= x22;
    x23 = x1 | x5;
    x24 = x23 ^ x8;
    x25 = x18 & ~x2;
    x26 = a2 & ~x25;
    x27 = x24 ^ x26;
    x28 = x6 | x7;
    x29 = x28 ^ x25;
```

```
    x30 = x9 ^ x24;
    x31 = x18 & ~x30;
    x32 = a2 & x31;
    x33 = x29 ^ x32;
    x34 = a1 & x33;
    x35 = x27 ^ x34;
    *out4 ^= x35;
    x36 = a3 & x28;
    x37 = x18 & ~x36;
    x38 = a2 | x3;
    x39 = x37 ^ x38;
    x40 = a3 | x31;
    x41 = x24 & ~x37;
    x42 = x41 | x3;
    x43 = x42 & ~a2;
    x44 = x40 ^ x43;
    x45 = a1 & ~x44;
    x46 = x39 ^ ~x45;
    *out1 ^= x46;
    x47 = x33 & ~x9;
    x48 = x47 ^ x39;
    x49 = x4 ^ x36;
    x50 = x49 & ~x5;
    x51 = x42 | x18;
    x52 = x51 ^ a5;
    x53 = a2 & ~x52;
    x54 = x50 ^ x53;
    x55 = a1 | x54;
    x56 = x48 ^ ~x55;
    *out3 ^= x56;
}
```

Cette fonction contient de nombreuses variables locales, mais toutes ne se retrouveront pas dans la pile. Compilons ce fichier avec MSVC 2008 et l'option /0x :

Listing 1.390: MSVC 2008 avec optimisation

```
PUBLIC    _s1
; Function compile flags : /Ogtpy
_TEXT    SEGMENT
_x6$ = -20          ; size = 4
_x3$ = -16          ; size = 4
_x1$ = -12          ; size = 4
_x8$ = -8           ; size = 4
_x4$ = -4           ; size = 4
_a1$ = 8            ; size = 4
_a2$ = 12           ; size = 4
_a3$ = 16           ; size = 4
_x33$ = 20          ; size = 4
_x7$ = 20           ; size = 4
_a4$ = 20           ; size = 4
_a5$ = 24           ; size = 4
tv326 = 28          ; size = 4
_x36$ = 28          ; size = 4
_x28$ = 28          ; size = 4
_a6$ = 28           ; size = 4
_out1$ = 32         ; size = 4
_x24$ = 36          ; size = 4
_out2$ = 36         ; size = 4
_out3$ = 40         ; size = 4
_out4$ = 44         ; size = 4
_s1    PROC
    sub    esp, 20                    ; 00000014H
    mov    edx, DWORD PTR _a5$[esp+16]
    push   ebx
    mov    ebx, DWORD PTR _a4$[esp+20]
    push   ebp
    push   esi
    mov    esi, DWORD PTR _a3$[esp+28]
    push   edi
```

```
    mov    edi, ebx
    not    edi
    mov    ebp, edi
    and    edi, DWORD PTR _a5$[esp+32]
    mov    ecx, edx
    not    ecx
    and    ebp, esi
    mov    eax, ecx
    and    eax, esi
    and    ecx, ebx
    mov    DWORD PTR _x1$[esp+36], eax
    xor    eax, ebx
    mov    esi, ebp
    or     esi, edx
    mov    DWORD PTR _x4$[esp+36], esi
    and    esi, DWORD PTR _a6$[esp+32]
    mov    DWORD PTR _x7$[esp+32], ecx
    mov    edx, esi
    xor    edx, eax
    mov    DWORD PTR _x6$[esp+36], edx
    mov    edx, DWORD PTR _a3$[esp+32]
    xor    edx, ebx
    mov    ebx, esi
    xor    ebx, DWORD PTR _a5$[esp+32]
    mov    DWORD PTR _x8$[esp+36], edx
    and    ebx, edx
    mov    ecx, edx
    mov    edx, ebx
    xor    edx, ebp
    or     edx, DWORD PTR _a6$[esp+32]
    not    ecx
    and    ecx, DWORD PTR _a6$[esp+32]
    xor    edx, edi
    mov    edi, edx
    or     edi, DWORD PTR _a2$[esp+32]
    mov    DWORD PTR _x3$[esp+36], ebp
    mov    ebp, DWORD PTR _a2$[esp+32]
    xor    edi, ebx
    and    edi, DWORD PTR _a1$[esp+32]
    mov    ebx, ecx
    xor    ebx, DWORD PTR _x7$[esp+32]
    not    edi
    or     ebx, ebp
    xor    edi, ebx
    mov    ebx, edi
    mov    edi, DWORD PTR _out2$[esp+32]
    xor    ebx, DWORD PTR [edi]
    not    eax
    xor    ebx, DWORD PTR _x6$[esp+36]
    and    eax, edx
    mov    DWORD PTR [edi], ebx
    mov    ebx, DWORD PTR _x7$[esp+32]
    or     ebx, DWORD PTR _x6$[esp+36]
    mov    edi, esi
    or     edi, DWORD PTR _x1$[esp+36]
    mov    DWORD PTR _x28$[esp+32], ebx
    xor    edi, DWORD PTR _x8$[esp+36]
    mov    DWORD PTR _x24$[esp+32], edi
    xor    edi, ecx
    not    edi
    and    edi, edx
    mov    ebx, edi
    and    ebx, ebp
    xor    ebx, DWORD PTR _x28$[esp+32]
    xor    ebx, eax
    not    eax
    mov    DWORD PTR _x33$[esp+32], ebx
    and    ebx, DWORD PTR _a1$[esp+32]
    and    eax, ebp
    xor    eax, ebx
```

```
    mov    ebx, DWORD PTR _out4$[esp+32]
    xor    eax, DWORD PTR [ebx]
    xor    eax, DWORD PTR _x24$[esp+32]
    mov    DWORD PTR [ebx], eax
    mov    eax, DWORD PTR _x28$[esp+32]
    and    eax, DWORD PTR _a3$[esp+32]
    mov    ebx, DWORD PTR _x3$[esp+36]
    or     edi, DWORD PTR _a3$[esp+32]
    mov    DWORD PTR _x36$[esp+32], eax
    not    eax
    and    eax, edx
    or     ebx, ebp
    xor    ebx, eax
    not    eax
    and    eax, DWORD PTR _x24$[esp+32]
    not    ebp
    or     eax, DWORD PTR _x3$[esp+36]
    not    esi
    and    ebp, eax
    or     eax, edx
    xor    eax, DWORD PTR _a5$[esp+32]
    mov    edx, DWORD PTR _x36$[esp+32]
    xor    edx, DWORD PTR _x4$[esp+36]
    xor    ebp, edi
    mov    edi, DWORD PTR _out1$[esp+32]
    not    eax
    and    eax, DWORD PTR _a2$[esp+32]
    not    ebp
    and    ebp, DWORD PTR _a1$[esp+32]
    and    edx, esi
    xor    eax, edx
    or     eax, DWORD PTR _a1$[esp+32]
    not    ebp
    xor    ebp, DWORD PTR [edi]
    not    ecx
    and    ecx, DWORD PTR _x33$[esp+32]
    xor    ebp, ebx
    not    eax
    mov    DWORD PTR [edi], ebp
    xor    eax, ecx
    mov    ecx, DWORD PTR _out3$[esp+32]
    xor    eax, DWORD PTR [ecx]
    pop    edi
    pop    esi
    xor    eax, ebx
    pop    ebp
    mov    DWORD PTR [ecx], eax
    pop    ebx
    add    esp, 20
    ret    0
_s1    ENDP
```

Seules 5 variables ont été allouées dans la pile par le compilateur.

Essayons maintenant une compilation avec la version 64 bits de MSVC 2008:

Listing 1.391: MSVC 2008 avec optimisation

```
a1$ = 56
a2$ = 64
a3$ = 72
a4$ = 80
x36$1$ = 88
a5$ = 88
a6$ = 96
out1$ = 104
out2$ = 112
out3$ = 120
out4$ = 128
s1    PROC
$LN3 :
```

```
    mov    QWORD PTR [rsp+24], rbx
    mov    QWORD PTR [rsp+32], rbp
    mov    QWORD PTR [rsp+16], rdx
    mov    QWORD PTR [rsp+8], rcx
    push   rsi
    push   rdi
    push   r12
    push   r13
    push   r14
    push   r15
    mov    r15, QWORD PTR a5$[rsp]
    mov    rcx, QWORD PTR a6$[rsp]
    mov    rbp, r8
    mov    r10, r9
    mov    rax, r15
    mov    rdx, rbp
    not    rax
    xor    rdx, r9
    not    r10
    mov    r11, rax
    and    rax, r9
    mov    rsi, r10
    mov    QWORD PTR x36$1$[rsp], rax
    and    r11, r8
    and    rsi, r8
    and    r10, r15
    mov    r13, rdx
    mov    rbx, r11
    xor    rbx, r9
    mov    r9, QWORD PTR a2$[rsp]
    mov    r12, rsi
    or     r12, r15
    not    r13
    and    r13, rcx
    mov    r14, r12
    and    r14, rcx
    mov    rax, r14
    mov    r8, r14
    xor    r8, rbx
    xor    rax, r15
    not    rbx
    and    rax, rdx
    mov    rdi, rax
    xor    rdi, rsi
    or     rdi, rcx
    xor    rdi, r10
    and    rbx, rdi
    mov    rcx, rdi
    or     rcx, r9
    xor    rcx, rax
    mov    rax, r13
    xor    rax, QWORD PTR x36$1$[rsp]
    and    rcx, QWORD PTR a1$[rsp]
    or     rax, r9
    not    rcx
    xor    rcx, rax
    mov    rax, QWORD PTR out2$[rsp]
    xor    rcx, QWORD PTR [rax]
    xor    rcx, r8
    mov    QWORD PTR [rax], rcx
    mov    rax, QWORD PTR x36$1$[rsp]
    mov    rcx, r14
    or     rax, r8
    or     rcx, r11
    mov    r11, r9
    xor    rcx, rdx
    mov    QWORD PTR x36$1$[rsp], rax
    mov    r8, rsi
    mov    rdx, rcx
    xor    rdx, r13
```

```
    not    rdx
    and    rdx, rdi
    mov    r10, rdx
    and    r10, r9
    xor    r10, rax
    xor    r10, rbx
    not    rbx
    and    rbx, r9
    mov    rax, r10
    and    rax, QWORD PTR a1$[rsp]
    xor    rbx, rax
    mov    rax, QWORD PTR out4$[rsp]
    xor    rbx, QWORD PTR [rax]
    xor    rbx, rcx
    mov    QWORD PTR [rax], rbx
    mov    rbx, QWORD PTR x36$1$[rsp]
    and    rbx, rbp
    mov    r9, rbx
    not    r9
    and    r9, rdi
    or     r8, r11
    mov    rax, QWORD PTR out1$[rsp]
    xor    r8, r9
    not    r9
    and    r9, rcx
    or     rdx, rbp
    mov    rbp, QWORD PTR [rsp+80]
    or     r9, rsi
    xor    rbx, r12
    mov    rcx, r11
    not    rcx
    not    r14
    not    r13
    and    rcx, r9
    or     r9, rdi
    and    rbx, r14
    xor    r9, r15
    xor    rcx, rdx
    mov    rdx, QWORD PTR a1$[rsp]
    not    r9
    not    rcx
    and    r13, r10
    and    r9, r11
    and    rcx, rdx
    xor    r9, rbx
    mov    rbx, QWORD PTR [rsp+72]
    not    rcx
    xor    rcx, QWORD PTR [rax]
    or     r9, rdx
    not    r9
    xor    rcx, r8
    mov    QWORD PTR [rax], rcx
    mov    rax, QWORD PTR out3$[rsp]
    xor    r9, r13
    xor    r9, QWORD PTR [rax]
    xor    r9, r8
    mov    QWORD PTR [rax], r9
    pop    r15
    pop    r14
    pop    r13
    pop    r12
    pop    rdi
    pop    rsi
    ret    0
s1    ENDP
```

Le compilateur n'a pas eu besoin d'allouer de l'espace sur la pile. x36 est synonyme de a5.

Il existe cependant des CPUs qui possèdent beaucoup plus de GPR. Itanium possède ainsi 128 registres.

1.30.2 ARM

Les instructions 64 bits sont apparues avec ARMv8.

1.30.3 Nombres flottants

Le traitement des nombres flottants en environnement x86-64 est expliqué ici: 1.31.

1.30.4 Critiques concernant l'architecture 64 bits

Certains se sont parfois irrité du doublement de la taille des pointeurs, en particulier dans le cache puisque les CPUs x64 ne peuvent de toute manière adresser que des adresses RAM sur 48 bits.

> Les pointeurs ont perdu mes faveurs au point que j'en viens à les injurier. Si je cherche vraiment à utiliser au mieux les capacités de mon ordinateur 64 bits, j'en conclus que je ferais mieux de ne pas utiliser de pointeurs. Les registres de mon ordinateur sont sur 64 bits, mais je n'ai que 2Go de RAM. Les pointeurs n'ont donc jamais plus de 32 bits significatifs. Et pourtant, chaque fois que j'utilise un pointeur, il me coûte 64 bits ce qui double la taille de ma structure de données. Pire, ils atterrissent dans mon cache et en gaspillent la moitié et cela me coûte car le cache est cher.
>
> Donc, si je cherche à grappiller, j'en viens à utiliser des tableaux au lieu de pointeurs. Je rédige des macros compliquées qui peuvent laisser l'impression à tort que j'utilise des pointeurs.

(Donald Knuth dans "Coders at Work: Reflections on the Craft of Programming ".)

Certains en sont venus à fabriquer leurs propres allocateurs de mémoire.

Il est intéressant de se pencher sur le cas de CryptoMiniSat[194]. Ce programme qui utilise rarement plus de 4Go de RAM, fait un usage intensif des pointeurs. Il nécessite donc moins de mémoire sur les architectures 32 bits que sur celles à 64 bits. Pour remédier à ce problème, l'auteur a donc programmé son propre allocateur (dans *clauseallocator.(h|cpp)*), qui lui permet d'allouer de la mémoire en utilisant des identifiants sur 32 bits à la place de pointeurs sur 64 bits.

1.31 Travailler avec des nombres à virgule flottante en utilisant SIMD

Bien sûr. le FPU est resté dans les processeurs compatible x86 lorsque les extensions SIMD ont été ajoutées.

L'extension SIMD (SSE2) offre un moyen facile de travailler avec des nombres à virgule flottante.

Le format des nombres reste le même (IEEE 754).

Donc, les compilateurs modernes (incluant ceux générant pour x86-64) utilisent les instructions SIMD au lieu de celles pour FPU.

On peut dire que c'est une bonne nouvelle, car il est plus facile de travailler avec elles.

Nous allons ré-utiliser les exemples de la section FPU ici: 1.19 on page 218.

1.31.1 Simple exemple

```
#include <stdio.h>

double f (double a, double b)
{
        return a/3.14 + b*4.1;
};
```

[194] https://github.com/msoos/cryptominisat/

```
int main()
{
        printf ("%f\n", f(1.2, 3.4));
};
```

x64

Listing 1.392: MSVC 2012 x64 avec optimisation

```
__real@4010666666666666 DQ 04010666666666666r   ; 4.1
__real@40091eb851eb851f DQ 040091eb851eb851fr   ; 3.14

a$ = 8
b$ = 16
f       PROC
        divsd   xmm0, QWORD PTR __real@40091eb851eb851f
        mulsd   xmm1, QWORD PTR __real@4010666666666666
        addsd   xmm0, xmm1
        ret     0
f       ENDP
```

Les valeurs en virgule flottante entrées sont passées dans les registres XMM0-XMM3, tout le reste—via la pile [195].

a est passé dans XMM0, b—via XMM1.

Les registres XMM font 128-bit (comme nous le savons depuis la section à propos de SIMD : 1.29 on page 408), mais les valeurs *double* font 64-bit, donc seulement la moitié basse du registre est utilisée.

DIVSD est une instruction SSE qui signifie « Divide Scalar Double-Precision Floating-Point Values » (Diviser des nombres flottants double-précision), elle divise une valeur de type *double* par une autre, stockées dans la moitié basse des opérandes.

Les constantes sont encodées par le compilateur au format IEEE 754.

MULSD et ADDSD fonctionnent de même, mais font la multiplication et l'addition.

Le résultat de l'exécution de la fonction de type *double* est laissé dans le registre XMM0.

C'est ainsi que travaille MSVC sans optimisation:

Listing 1.393: MSVC 2012 x64

```
__real@4010666666666666 DQ 04010666666666666r   ; 4.1
__real@40091eb851eb851f DQ 040091eb851eb851fr   ; 3.14

a$ = 8
b$ = 16
f       PROC
        movsdx  QWORD PTR [rsp+16], xmm1
        movsdx  QWORD PTR [rsp+8], xmm0
        movsdx  xmm0, QWORD PTR a$[rsp]
        divsd   xmm0, QWORD PTR __real@40091eb851eb851f
        movsdx  xmm1, QWORD PTR b$[rsp]
        mulsd   xmm1, QWORD PTR __real@4010666666666666
        addsd   xmm0, xmm1
        ret     0
f       ENDP
```

Légèrement redondant. Les arguments en entrée sont sauvés dans le « shadow space » (1.10.2 on page 101), mais seule leur moitié inférieure, i.e., seulement la valeur 64-bit de type *double*. GCC produit le même code.

[195] MSDN: Parameter Passing

x86

Compilons cet exemple pour x86. Bien qu'il compile pour x86, MSVC 2012 utilise des instructions SSE2:

Listing 1.394: MSVC 2012 x86 sans optimisation

```
tv70 = -8       ; size = 8
_a$ = 8         ; size = 8
_b$ = 16        ; size = 8
_f      PROC
        push    ebp
        mov     ebp, esp
        sub     esp, 8
        movsd   xmm0, QWORD PTR _a$[ebp]
        divsd   xmm0, QWORD PTR __real@40091eb851eb851f
        movsd   xmm1, QWORD PTR _b$[ebp]
        mulsd   xmm1, QWORD PTR __real@4010666666666666
        addsd   xmm0, xmm1
        movsd   QWORD PTR tv70[ebp], xmm0
        fld     QWORD PTR tv70[ebp]
        mov     esp, ebp
        pop     ebp
        ret     0
_f      ENDP
```

Listing 1.395: MSVC 2012 x86 avec optimisation

```
tv67 = 8        ; size = 8
_a$ = 8         ; size = 8
_b$ = 16        ; size = 8
_f      PROC
        movsd   xmm1, QWORD PTR _a$[esp-4]
        divsd   xmm1, QWORD PTR __real@40091eb851eb851f
        movsd   xmm0, QWORD PTR _b$[esp-4]
        mulsd   xmm0, QWORD PTR __real@4010666666666666
        addsd   xmm1, xmm0
        movsd   QWORD PTR tv67[esp-4], xmm1
        fld     QWORD PTR tv67[esp-4]
        ret     0
_f      ENDP
```

C'est presque le même code, toutefois, il y a quelques différences relatives aux conventions d'appel: 1) les arguments ne sont pas passés dans des registres XMM, mais par la pile, comme dans les exemples FPU (1.19 on page 218); 2) le résultat de la fonction est renvoyé dans `ST(0)` — afin de faire cela, il est copié (à travers la variable locale `tv`) depuis un des registres XMM dans `ST(0)`.

Essayons l'exemple optimisé dans OllyDbg :

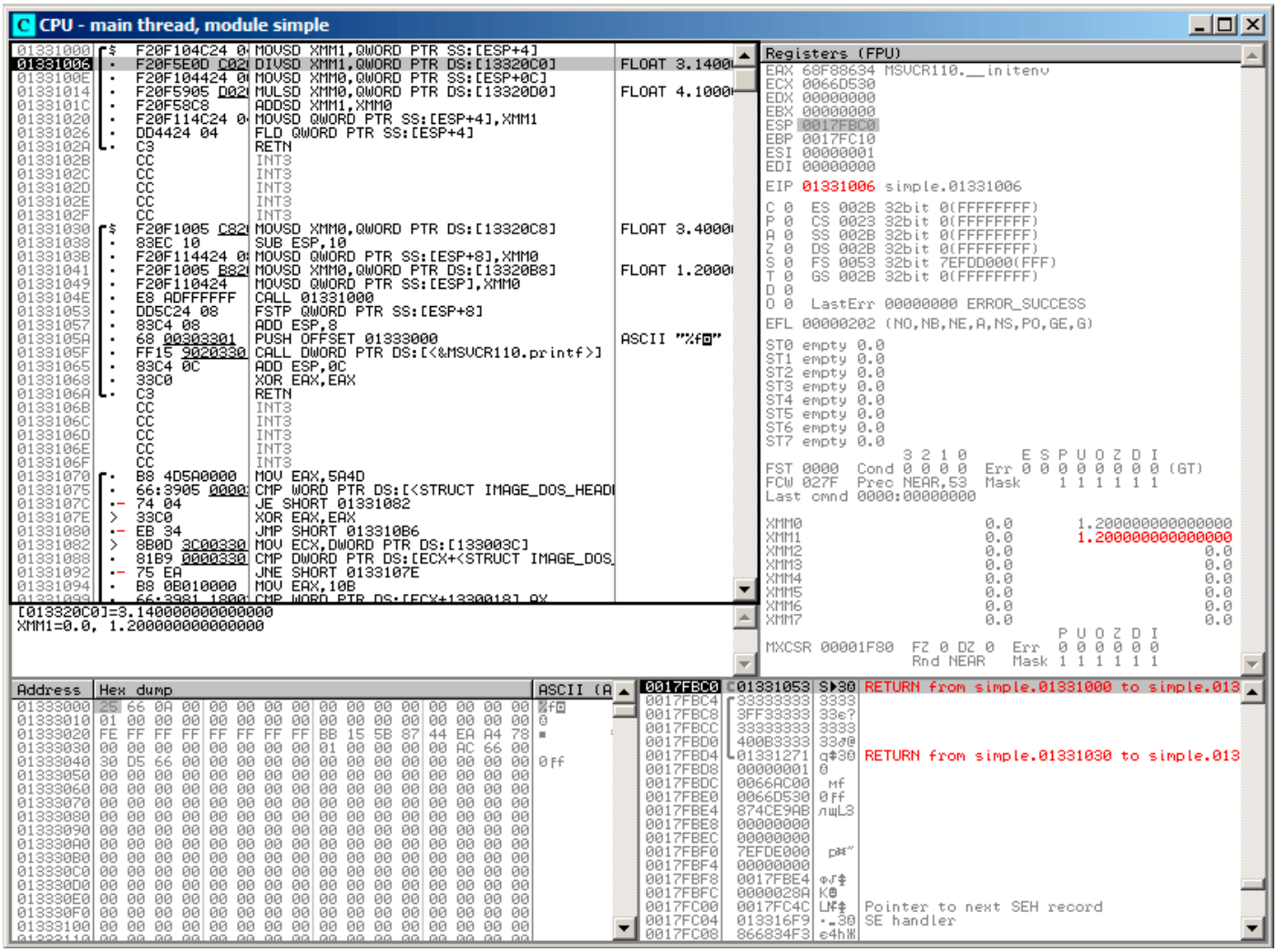


Fig. 1.113: OllyDbg : MOVSD charge la valeur de a dans XMM1

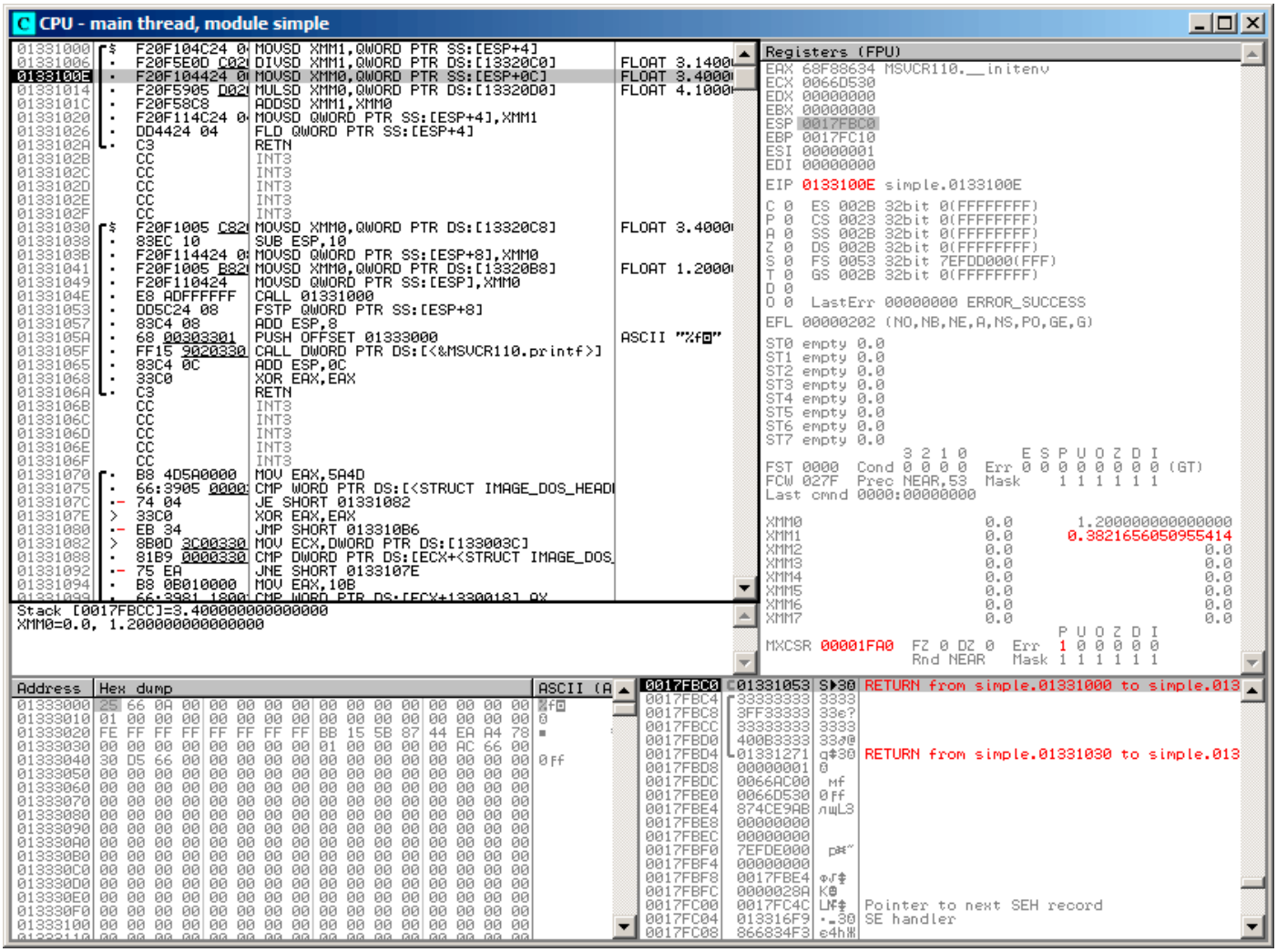


Fig. 1.114: OllyDbg : DIVSD a calculé le quotient et l'a stocké dans XMM1

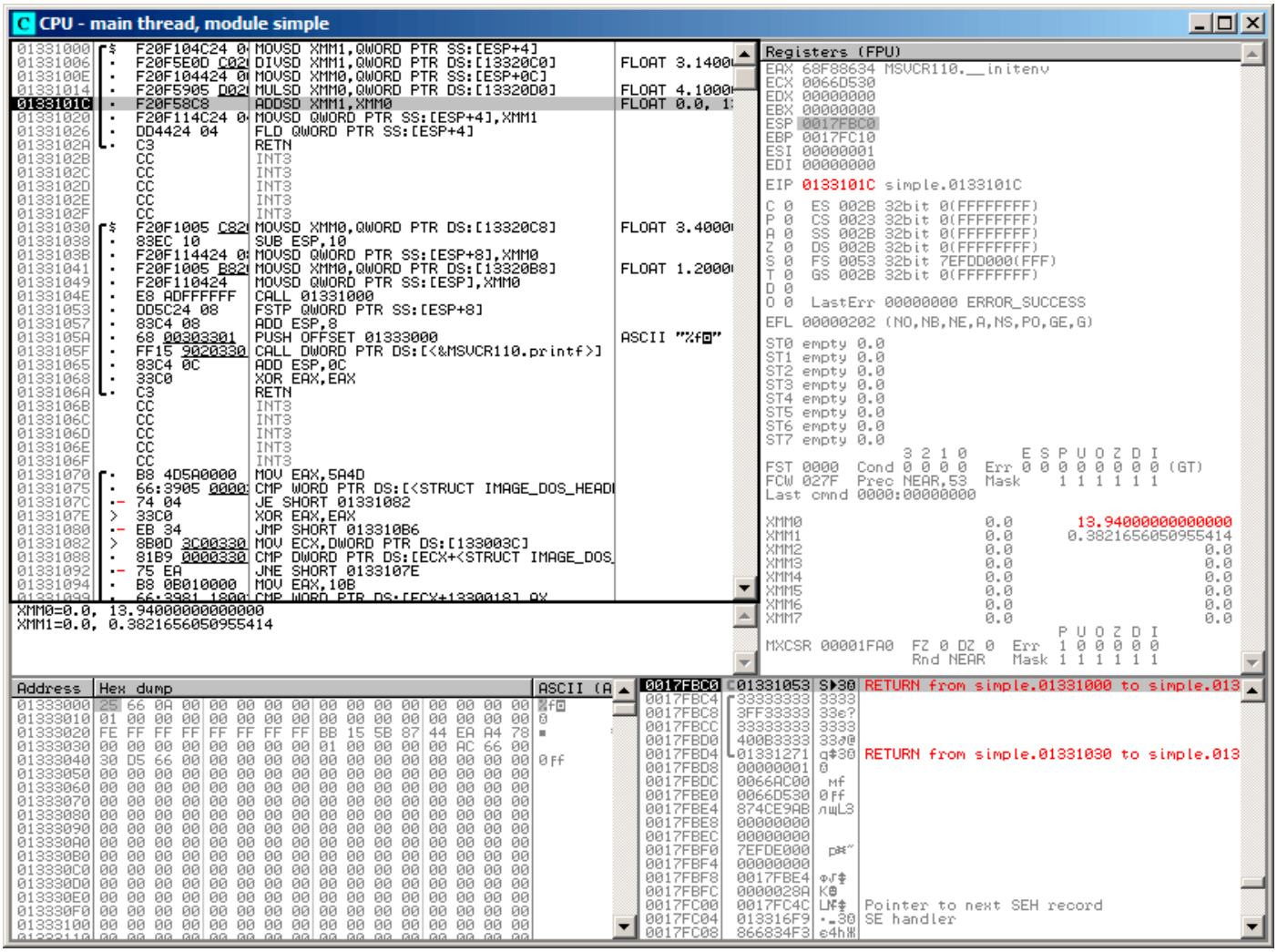


Fig. 1.115: OllyDbg : MULSD a calculé le produit et l'a stocké dans XMM0

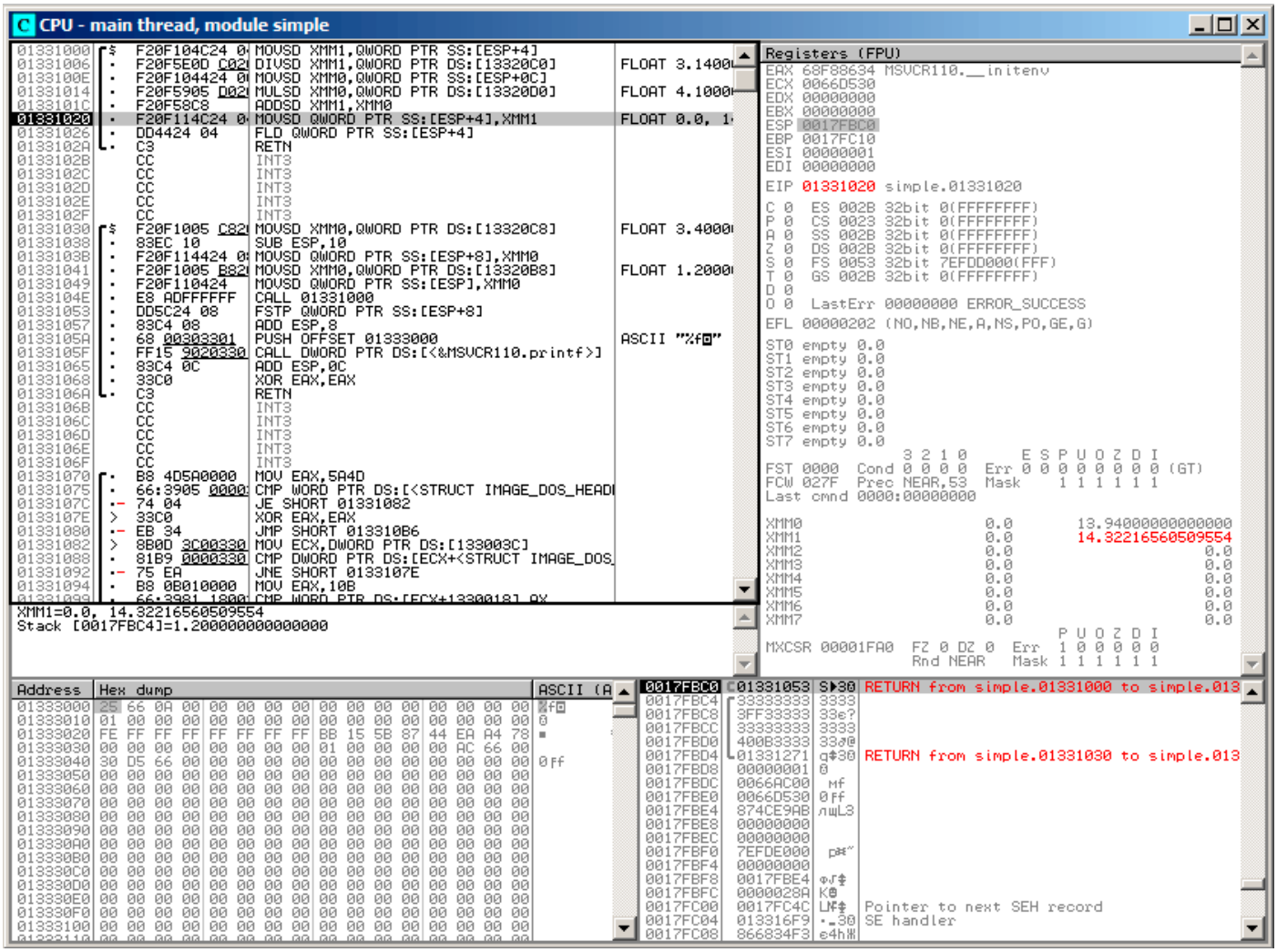


Fig. 1.116: OllyDbg : ADDSD ajoute la valeur dans XMM0 à celle dans XMM1

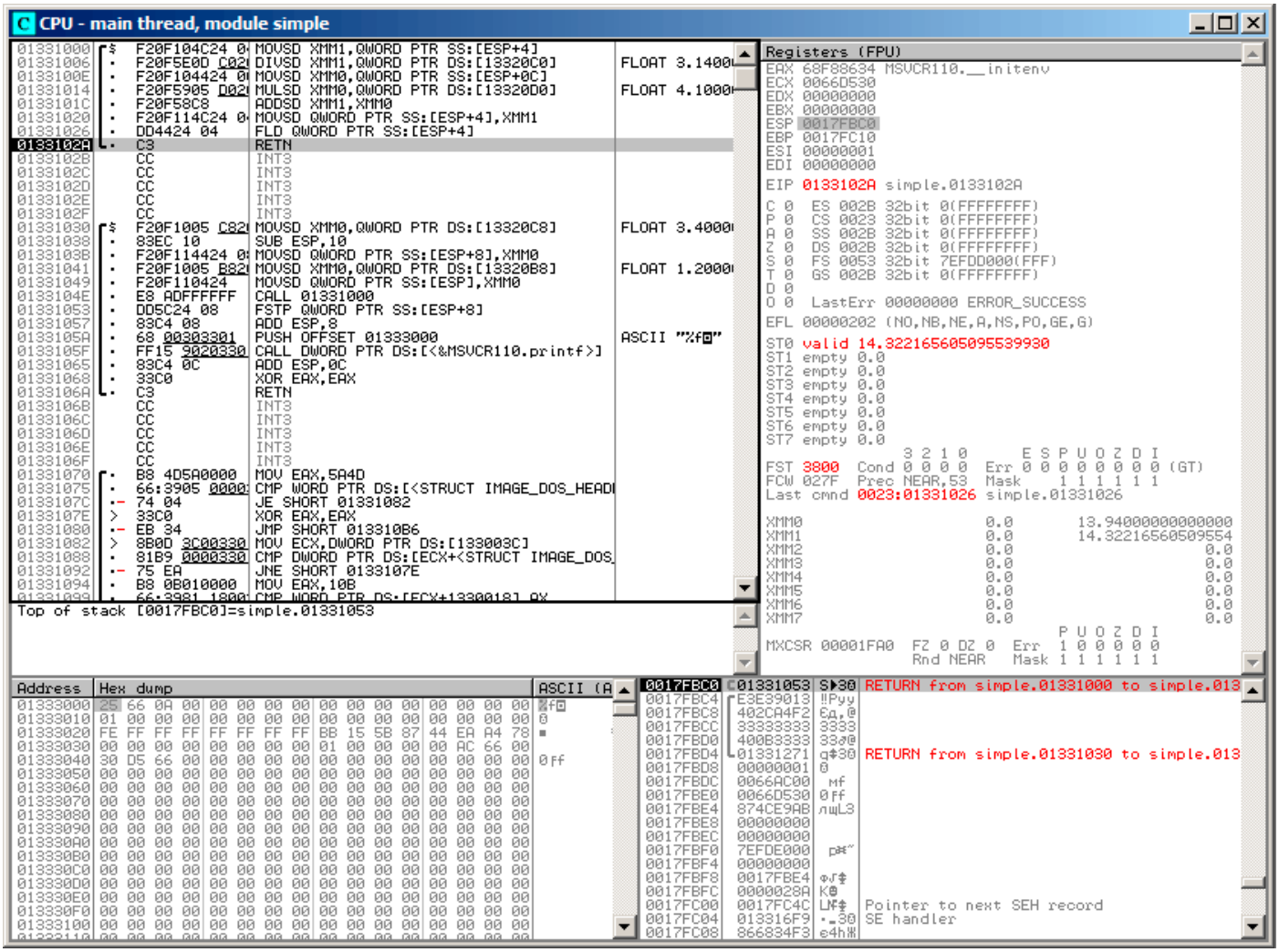


Fig. 1.117: OllyDbg : FLD laisse le résultat de la fonction dans ST(0)

Nous voyons qu'OllyDbg montre les registres XMM comme des paires de nombres *double*, mais seule la partie *basse* est utilisée.

Apparemment, OllyDbg les montre dans ce format car les instructions SSE2 (suffixées avec -SD) sont exécutées actuellement.

Mais bien sûr, il est possible de changer le format du registre et de voir le contenu comme 4 nombres *float* ou juste comme 16 octets.

1.31.2 Passer des nombres à virgule flottante via les arguments

```
#include <math.h>
#include <stdio.h>

int main ()
{
        printf ("32.01 ^ 1.54 = %lf\n", pow (32.01,1.54));

        return 0;
}
```

Ils sont passés dans la moitié basse des registres XMM0-XMM3.

Listing 1.396: MSVC 2012 x64 avec optimisation

```
$SG1354 DB      '32.01 ^ 1.54 = %lf', 0aH, 00H

__real@40400147ae147ae1 DQ 040400147ae147ae1r    ; 32.01
__real@3ff8a3d70a3d70a4 DQ 03ff8a3d70a3d70a4r    ; 1.54

main    PROC
        sub     rsp, 40                                 ; 00000028H
        movsdx  xmm1, QWORD PTR __real@3ff8a3d70a3d70a4
        movsdx  xmm0, QWORD PTR __real@40400147ae147ae1
        call    pow
        lea     rcx, OFFSET FLAT :$SG1354
        movaps  xmm1, xmm0
        movd    rdx, xmm1
        call    printf
        xor     eax, eax
        add     rsp, 40                                 ; 00000028H
        ret     0
main    ENDP
```

Il n'y a pas d'instruction MOVSDX dans les manuels Intel et AMD (8.1.4 on page 649), elle y est appelée MOVSD. Donc il y a deux instructions qui partagent le même nom en x86 (à propos de l'autre lire: **??** on page ??). Apparemment, les développeurs de Microsoft voulaient arrêter cette pagaille, donc ils l'ont renommée MOVSDX. Elle charge simplement une valeur dans la moitié inférieure d'un registre XMM.

pow() prends ses arguments de XMM0 et XMM1, et renvoie le résultat dans XMM0. Il est ensuite déplacé dans RDX pour printf(). Pourquoi? Peut-être parce que printf()—est une fonction avec un nombre variable d'arguments?

Listing 1.397: GCC 4.4.6 x64 avec optimisation

```
.LC2 :
        .string "32.01 ^ 1.54 = %lf\n"
main :
        sub     rsp, 8
        movsd   xmm1, QWORD PTR .LC0[rip]
        movsd   xmm0, QWORD PTR .LC1[rip]
        call    pow
        ; le résultat est maintenant dans XMM0
        mov     edi, OFFSET FLAT :.LC2
        mov     eax, 1 ; nombre de registres vecteur passé
        call    printf
        xor     eax, eax
        add     rsp, 8
        ret
.LC0 :
        .long   171798692
        .long   1073259479
.LC1 :
        .long   2920577761
        .long   1077936455
```

GCC génère une sortie plus claire. La valeur pour printf() est passée dans XMM0. À propos, il y a un cas lorsque 1 est écrit dans EAX pour printf()—ceci implique qu'un argument sera passé dans des registres

vectoriels, comme le requiert le standard [Michael Matz, Jan Hubicka, Andreas Jaeger, Mark Mitchell, *System V Application Binary Interface. AMD64 Architecture Processor Supplement*, (2013)] [196].

1.31.3 Exemple de comparaison

```
#include <stdio.h>

double d_max (double a, double b)
{
        if (a>b)
                return a;

        return b;
};

int main()
{
        printf ("%f\n", d_max (1.2, 3.4));
        printf ("%f\n", d_max (5.6, -4));
};
```

x64

Listing 1.398: MSVC 2012 x64 avec optimisation

```
a$ = 8
b$ = 16
d_max   PROC
        comisd  xmm0, xmm1
        ja      SHORT $LN2@d_max
        movaps  xmm0, xmm1
$LN2@d_max :
        fatret  0
d_max   ENDP
```

MSVC avec optimisation génère un code très facile à comprendre.

`COMISD` is « Compare Scalar Ordered Double-Precision Floating-Point Values and Set EFLAGS » (comparer des valeurs double précision en virgule flottante scalaire ordrées et mettre les EFLAGS). Pratiquement, c'est ce qu'elle fait.

MSVC sans optimisation génère plus de code redondant, mais il n'est toujours pas très difficile à comprendre:

Listing 1.399: MSVC 2012 x64

```
a$ = 8
b$ = 16
d_max   PROC
        movsdx  QWORD PTR [rsp+16], xmm1
        movsdx  QWORD PTR [rsp+8], xmm0
        movsdx  xmm0, QWORD PTR a$[rsp]
        comisd  xmm0, QWORD PTR b$[rsp]
        jbe     SHORT $LN1@d_max
        movsdx  xmm0, QWORD PTR a$[rsp]
        jmp     SHORT $LN2@d_max
$LN1@d_max :
        movsdx  xmm0, QWORD PTR b$[rsp]
$LN2@d_max :
        fatret  0
d_max   ENDP
```

Toutefois, GCC 4.4.6 effectue plus d'optimisations et utilise l'instruction `MAXSD` (« Return Maximum Scalar Double-Precision Floating-Point Value ») qui choisit la valeur maximum!

[196] Aussi disponible en https://software.intel.com/sites/default/files/article/402129/mpx-linux64-abi.pdf

Listing 1.400: GCC 4.4.6 x64 avec optimisation

```
d_max :
        maxsd   xmm0, xmm1
        ret
```

x86

Compilons cet exemple dans MSVC 2012 avec l'optimisation activée:

Listing 1.401: MSVC 2012 x86 avec optimisation

```
_a$ = 8          ; size = 8
_b$ = 16         ; size = 8
_d_max  PROC
        movsd   xmm0, QWORD PTR _a$[esp-4]
        comisd  xmm0, QWORD PTR _b$[esp-4]
        jbe     SHORT $LN1@d_max
        fld     QWORD PTR _a$[esp-4]
        ret     0
$LN1@d_max :
        fld     QWORD PTR _b$[esp-4]
        ret     0
_d_max  ENDP
```

Presque la même chose, mais les valeurs de a et b sont prises depuis la pile et le résultat de la fonction est laissé dans `ST(0)`.

Si nous chargeons cet exemple dans OllyDbg, nous pouvons voir comment l'instruction `COMISD` compare les valeurs et met/efface les flags `CF` et `PF` :

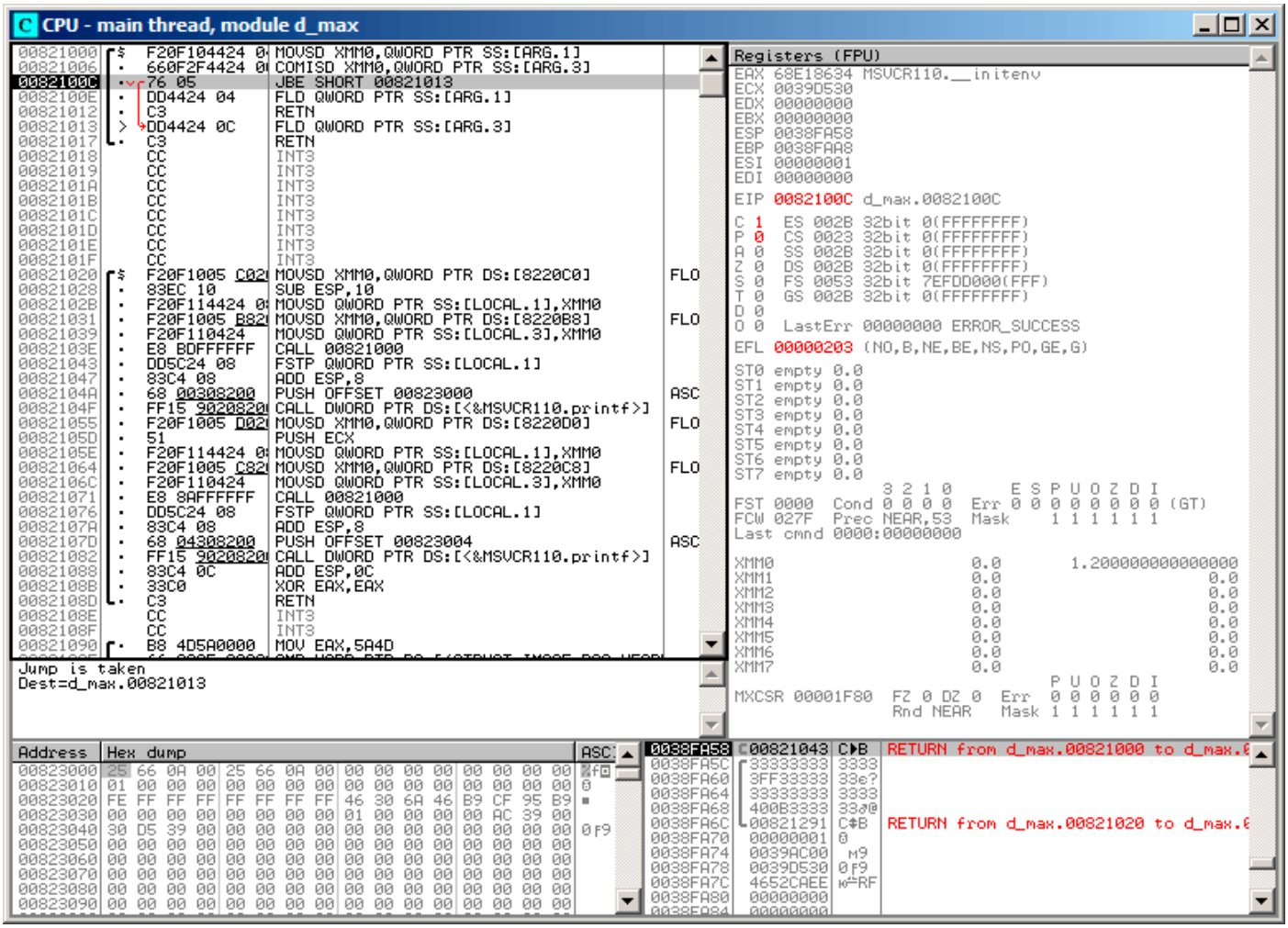


Fig. 1.118: OllyDbg : `COMISD` a changé les flags `CF` et `PF`

1.31.4 Calcul de l'epsilon de la machine: x64 et SIMD

Revoyons l'exemple « calcul de l'epsilon de la machine » pour *double* liste.1.25.2.

Maintenant nous compilons pour x64:

Listing 1.402: MSVC 2012 x64 avec optimisation

```
v$ = 8
calculate_machine_epsilon PROC
        movsdx  QWORD PTR v$[rsp], xmm0
        movaps  xmm1, xmm0
        inc     QWORD PTR v$[rsp]
        movsdx  xmm0, QWORD PTR v$[rsp]
        subsd   xmm0, xmm1
        ret     0
calculate_machine_epsilon ENDP
```

Il n'y a pas moyen d'ajouter 1 à une valeur dans un registre XMM 128-bit, donc il doit être placé en mémoire.

Il y a toutefois l'instruction ADDSD (*Add Scalar Double-Precision Floating-Point Values* ajouter des valeurs scalaires à virgule flottante double-précision), qui peut ajouter une valeur dans la moitié 64-bit basse d'un registre XMM en ignorant celle du haut, mais MSVC 2012 n'est probablement pas encore assez bon [197].

Néanmoins, la valeur est ensuite rechargée dans un registre XMM et la soustraction est effectuée. SUBSD est « Subtract Scalar Double-Precision Floating-Point Values » (soustraire des valeurs en virgule flottante double-précision), i.e., elle opère sur la partie 64-bit basse d'un registre XMM 128-bit. Le résultat est renvoyé dans le registre XMM0.

1.31.5 Exemple de générateur de nombre pseudo-aléatoire revisité

Revoyons l'exemple de « générateur de nombre pseudo-aléatoire » liste.1.25.1.

Si nous compilons ceci en MSVC 2012, il va utiliser les instructions SIMD pour le FPU.

Listing 1.403: MSVC 2012 avec optimisation

```
__real@3f800000 DD 03f800000r   ; 1

tv128 = -4
_tmp$ = -4
?float_rand@@YAMXZ PROC
        push    ecx
        call    ?my_rand@@YAIXZ
; EAX=valeur pseudo-aléatoire
        and     eax, 8388607    ; 007fffffH
        or      eax, 1065353216 ; 3f800000H
; EAX=valeur pseudo-aléatoire & 0x007fffff | 0x3f800000
; la stocker dans la pile locale :
        mov     DWORD PTR _tmp$[esp+4], eax
; la recharger comme un nombre à virgule flottante :
        movss   xmm0, DWORD PTR _tmp$[esp+4]
; soustraire 1.0:
        subss   xmm0, DWORD PTR __real@3f800000
; mettre la valeur dans ST0 en la plaçant dans une variable temporaire...
        movss   DWORD PTR tv128[esp+4], xmm0
; ... et en la rechargeant dans ST0 :
        fld     DWORD PTR tv128[esp+4]
        pop     ecx
        ret     0
?float_rand@@YAMXZ ENDP
```

Toutes les instructions ont le suffixe -SS, qui signifie « Scalar Single » (scalaire simple).

« Scalar » (scalaire) implique qu'une seule valeur est stockée dans le registre.

« Single » (simple[198]) signifie un type de donnée *float*.

[197] À titre d'exercice, vous pouvez retravailler ce code pour éliminer l'usage de la pile locale

[198] pour simple précision

1.31.6 Résumé

Seule la moitié basse des registres XMM est utilisée dans tous les exemples ici, pour stocker un nombre au format IEEE 754.

Pratiquement, toutes les instructions préfixées par `-SD` (« Scalar Double-Precision »)—sont des instructions travaillant avec des nombres à virgule flottante au format IEEE 754, stockés dans la moitié 64-bit basse d'un registre XMM.

Et c'est plus facile que dans le FPU, sans doute parce que les extensions SIMD ont évolué dans un chemin moins chaotique que celles FPU dans le passé. Le modèle de pile de registre n'est pas utilisé.

Si vous voulez, essayez de remplacer *double* avec *float*

dans ces exemples, la même instruction sera utilisée, mais préfixée avec `-SS` (« Scalar Single-Precision » scalaire simple-précision), par exemple, `MOVSS`, `COMISS`, `ADDSS`, etc.

« Scalaire » implique que le registre SIMD contienne seulement une valeur au lieu de plusieurs.

Les instructions travaillant avec plusieurs valeurs dans un registre simultanément ont « Packed » dans leur nom.

Inutile de dire que les instructions SSE2 travaillent avec des nombres 64-bit au format IEEE 754 (*double*), alors que la représentation interne des nombres à virgule flottante dans le FPU est sur 80-bit.

C'est pourquoi la FPU produit moins d'erreur d'arrondi et par conséquent, le FPU peut donner des résultats de calcul plus précis.

1.32 Détails spécifiques à ARM

1.32.1 Signe (#) avant un nombre

Le compilateur Keil, IDA et objdump font précéder tous les nombres avec le signe « # », par exemple: liste.1.16.1.

Mais lorsque GCC 4.9 génère une sortie en langage d'assemblage, il ne le fait pas, par exemple: liste.3.15.

Les listings ARM dans ce livre sont quelque peu mélangés.

Il est difficile de dire quelle méthode est juste. Censément, on doit suivre les règles de l'environnement dans lequel on travaille.

1.32.2 Modes d'adressage

Cette instruction est possible en ARM64:

```
ldr     x0, [x29,24]
```

Ceci signifie ajouter 24 à la valeur dans X29 et charger la valeur à cette adresse.

Notez s'il vous plaît que 24 est à l'intérieur des parenthèses. La signification est différente si le nombre est à l'extérieur des parenthèses:

```
ldr     w4, [x1],28
```

Ceci signifie charger la valeur à l'adresse dans X1, puis ajouter 28 à X1.

ARM permet d'ajouter ou de soustraire une constante à/de l'adresse utilisée pour charger.

Et il est possible de faire cela à la fois avant et après le chargement.

Il n'y a pas de tels modes d'adressage en x86, mais ils sont présents dans d'autres processeurs, même sur le PDP-11.

Il y a une légende disant que les modes pré-incrémentation, post-incrémentation, pré-décrémentation et post-décrémentation du PDP-11, sont « responsables » de l'apparition du genre de constructions en langage C (qui a été développé sur PDP-11) comme *ptr++, *++ptr, *ptr--, *--ptr.

À propos, ce sont des caractéristiques de C difficiles à mémoriser. Voici comment ça se passe:

terme C	terme ARM	déclaration C	ce que ça fait
Post-incrémentation	adressage post-indexé	`*ptr++`	utiliser la valeur `*ptr`, puis incrémenter le pointeur `ptr`
Post-décrémentation	adressage post-indexé	`*ptr--`	utiliser la valeur `*ptr`, puis décrémenter le pointeur `ptr`
Pré-incrémentation	adressage pré-indexé	`*++ptr`	incrémenter le pointeur `ptr`, puis utiliser la valeur `*ptr`
Pré-décrémentation	adressage pré-indexé	`*--ptr`	décrémenter le pointeur `ptr`, puis utiliser la valeur `*ptr`

La pré-indexation est marquée avec un point d'exclamation en langage d'assemblage ARM. Par exemple, voir ligne 2 dans liste.1.28.

Dennis Ritchie (un des créateurs du langage C) a mentionné que cela a vraisemblablement été inventé par Ken Thompson (un autre créateur du C) car cette possibilité était présente sur le PDP-7 [199], [Dennis M. Ritchie, *The development of the C language*, (1993)][200].

Ainsi, les compilateurs de langage C peuvent l'utiliser, si elle est présente sur le processeur cible.

C'est très pratique pour le traitement de tableau.

1.32.3 Charger une constante dans un registre

ARM 32-bit

Comme nous le savons déjà, toutes les instructions ont une taille de 4 octets en mode ARM et de 2 octets en mode Thumb.

Mais comment peut-on charger une valeur 32-bit dans un registre, s'il n'est pas possible de l'encoder dans une instruction?

Essayons:

```
unsigned int f()
{
        return 0x12345678;
};
```

Listing 1.404: GCC 4.6.3 -O3 Mode ARM

```
f :
        ldr     r0, .L2
        bx      lr
.L2 :
        .word   305419896 ; 0x12345678
```

Donc, la valeur `0x12345678` est simplement stockée à part en mémoire et chargée si besoin.

Mais il est possible de se débarrasser de l'accès supplémentaire en mémoire.

Listing 1.405: GCC 4.6.3 -O3 -march=armv7-a (Mode ARM)

```
movw    r0, #22136      ; 0x5678
movt    r0, #4660       ; 0x1234
bx      lr
```

Nous voyons que la valeur est chargée dans le registre par parties, la partie basse en premier (en utilisant `MOVW`), puis la partie haute (en utilisant `MOVT`).

Ceci implique que 2 instructions sont nécessaires en mode ARM pour charger une valeur 32-bit dans un registre.

Ce n'est pas un problème, car en fait il n'y pas beaucoup de constantes dans du code réel (excepté pour 0 et 1).

[199] http://yurichev.com/mirrors/C/c_dmr_postincrement.txt
[200] Aussi disponible en http://go.yurichev.com/17264

Est-ce que ça signifie que la version à deux instructions est plus lente que celle à une instruction?

C'est discutable. Le plus souvent, les processeurs ARM modernes sont capable de détecter de telle séquences et les exécutent rapidement.

D'un autre côté, IDA est capable de détecter ce genre de patterns dans le code et désassemble cette fonction comme:

```
MOV     R0, 0x12345678
BX      LR
```

ARM64

```
uint64_t f()
{
        return 0x12345678ABCDEF01;
};
```

Listing 1.406: GCC 4.9.1 -O3

```
mov     x0, 61185   ; 0xef01
movk    x0, 0xabcd, lsl 16
movk    x0, 0x5678, lsl 32
movk    x0, 0x1234, lsl 48
ret
```

MOVK signifie « MOV Keep » (déplacer garder), i.e., elle écrit une valeur 16-bit dans le registre, sans affecter le reste des bits. Le suffixe LSL signifie décaler la valeur à gauche de 16, 32 et 48 bits à chaque étape. Le décalage est fait avant le chargement.

Ceci implique que 4 instructions sont nécessaires pour charger une valeur de 64-bit dans un registre.

Charger un nombre à virgule flottante dans un registre

Il est possible de stocker un nombre à virgule flottante dans un D-registre en utilisant une seule instruction.

Par exemple:

```
double a()
{
        return 1.5;
};
```

Listing 1.407: GCC 4.9.1 -O3 + objdump

```
0000000000000000 <a> :
   0:   1e6f1000        fmov    d0, #1.5000000000000000e+000
   4:   d65f03c0        ret
```

Le nombre 1.5 a en effet été encodé dans une instruction 32-bit. Mais comment?

En ARM64, il y a 8 bits dans l'instruction FMOV pour encoder certains nombres à virgule flottante.

L'algorithme est appelé VFPExpandImm() en [*ARM Architecture Reference Manual, ARMv8, for ARMv8-A architecture profile*, (2013)][201]. Ceci est aussi appelé *minifloat*[202] (mini flottant).

Nous pouvons essayer différentes valeurs. Le compilateur est capable d'encoder 30.0 et 31.0, mais il ne peut pas encoder 32.0, car 8 octets doivent être alloués pour ce nombre au format IEEE 754:

```
double a()
{
        return 32;
};
```

[201] Aussi disponible en http://yurichev.com/mirrors/ARMv8-A_Architecture_Reference_Manual_(Issue_A.a).pdf
[202] Wikipédia

Listing 1.408: GCC 4.9.1 -O3

```
a :
        ldr     d0, .LC0
        ret
.LC0 :
        .word   0
        .word   1077936128
```

1.32.4 Relocations en ARM64

Comme nous le savons, il y a des instructions 4-octet en ARM64, donc il est impossible d'écrire un nombre large dans un registre en utilisant une seule instruction.

Cependant, une image exécutable peut être chargée à n'importe quelle adresse aléatoire en mémoire, c'est pourquoi les relocations existent.

L'adresse est formée en utilisant la paire d'instructions ADRP et ADD en ARM64.

La première charge l'adresse d'une page de 4KiB et la seconde ajoute le reste. Compilons l'exemple de « Hello, world! » (liste.1.8) avec GCC (Linaro) 4.9 sous win32:

Listing 1.409: GCC (Linaro) 4.9 et objdump du fichier objet

```
...>aarch64-linux-gnu-gcc.exe hw.c -c

...>aarch64-linux-gnu-objdump.exe -d hw.o

...

0000000000000000 <main> :
   0:   a9bf7bfd        stp     x29, x30, [sp,#-16]!
   4:   910003fd        mov     x29, sp
   8:   90000000        adrp    x0, 0 <main>
   c :   91000000        add     x0, x0, #0x0
  10:   94000000        bl      0 <printf>
  14:   52800000        mov     w0, #0x0                        // #0
  18:   a8c17bfd        ldp     x29, x30, [sp],#16
  1c :   d65f03c0        ret

...>aarch64-linux-gnu-objdump.exe -r hw.o

...

RELOCATION RECORDS FOR [.text]:
OFFSET           TYPE              VALUE
0000000000000008 R_AARCH64_ADR_PREL_PG_HI21  .rodata
000000000000000c R_AARCH64_ADD_ABS_LO12_NC  .rodata
0000000000000010 R_AARCH64_CALL26  printf
```

Donc, il y a 3 relocations dans ce fichier objet.

- La première prend l'adresse de la page, coupe les 12 bits du bas et écrit les 21 bits du haut restants dans le champs de bit de l'instruction ADRP. Ceci car nous n'avons pas besoin d'encoder les 12 bits bas, et l'instruction ADRP possède seulement de l'espace pour 21 bits.
- La seconde met les 12 bits de l'adresse relative au début de la page dans le champ de bits de l'instruction ADD.
- La dernière, celle de 26-bit, est appliquée à l'instruction à l'adresse 0x10 où le saut à la fonction printf() se trouve.

  Toutes les adresses des instructions ARM64 (et ARM en mode ARM) ont zéro dans les deux bits les plus bas (car toutes les instructions ont une taille de 4 octets), donc on doit seulement encoder les 26 bits du haut de l'espace d'adresse de 28-bit (± 128MB).

Il n'y a pas de telles relocations dans le fichier exécutable: car l'adresse où se trouve la chaîne « Hello! » est connue, la page, et l'adresse de puts() sont aussi connues.

Donc, il y a déjà des valeurs mises dans les instructions ADRP, ADD et BL (l'éditeur de liens à écrit des valeurs lors de l'édition de liens):

Listing 1.410: objdump du fichier exécutable

```
0000000000400590 <main> :
  400590:       a9bf7bfd        stp     x29, x30, [sp,#-16]!
  400594:       910003fd        mov     x29, sp
  400598:       90000000        adrp    x0, 400000 <_init-0x3b8>
  40059c :       91192000        add     x0, x0, #0x648
  4005a0 :       97ffffa0        bl      400420 <puts@plt>
  4005a4 :       52800000        mov     w0, #0x0                        // #0
  4005a8 :       a8c17bfd        ldp     x29, x30, [sp],#16
  4005ac :       d65f03c0        ret

...

Contents of section .rodata :
 400640 01000200 00000000 48656c6c 6f210000  ........Hello!..
```

À titre d'exemple, essayons de désassembler manuellement l'instruction BL.
0x97ffffa0 est $0b10010111111111111111111110100000$. D'après [*ARM Architecture Reference Manual, ARMv8, for ARMv8-A architecture profile*, (2013)C5.6.26], *imm26* correspond aux derniers 26 bits:
$imm26 = 0b11111111111111111111010000$0. Il s'agit de 0x3FFFFA0, mais le MSB est 1, donc le nombre est négatif, et nous pouvons le convertir manuellement en une forme pratique pour nous. D'après les règles de la négation (2.2 on page 457), il faut simplement inverser tous les bits (ça donne 0b1011111=0x5F), et ajouter 1 (0x5F+1=0x60). Donc le nombre sous sa forme signée est -0x60. Multiplions -0x60 par 4 (car l'adresse est stockée dans l'opcode est divisée par 4): ça fait -0x180. Maintenant calculons l'adresse de destination: 0x4005a0 + (-0x180) = 0x400420 (noter s'il vous plaît: nous considérons l'adresse de l'instruction BL, pas la valeur courante du PC, qui peut être différente!). Donc l'adresse de destination est 0x400420.

Plus d'informations relatives aux relocations en ARM64: [*ELF for the ARM 64-bit Architecture (AArch64)*, (2013)][203].

1.33 Détails spécifiques MIPS

1.33.1 Charger une constante 32-bit dans un registre

```
unsigned int f()
{
        return 0x12345678;
};
```

Toutes les instructions MIPS, tout comme en ARM, ont une taille de 32-bit, donc il n'est pas possible d'inclure une constante 32-bit dans une instruction.

Donc il faut utiliser au moins deux instructions: la première charge la partie haute du nombre de 32-bit et la seconde effectue une opération OR, qui met effectivement la partie 16-bit basse du registre de destination:

Listing 1.411: GCC 4.4.5 -O3 (résultat en sortie de l'assembleur)

```
        li      $2,305397760  # 0x12340000
        j       $31
        ori     $2,$2,0x5678 ; slot de délai de branchement
```

IDA reconnaît ce pattern de code, qui se rencontre fréquemment, donc, par commodité, il montre la dernière instruction ORI comme la pseudo-instruction LI qui charge soit disant un nombre entier de 32-bit dans le registre $V0.

Listing 1.412: GCC 4.4.5 -O3 (IDA)

```
         lui     $v0, 0x1234
         jr      $ra
         li      $v0, 0x12345678 ; slot de délai de branchement
```

[203] Aussi disponible en http://go.yurichev.com/17288

La sortie de l'assembleur GCC a la pseudo instruction LI, mais il s'agit en fait ici de LUI (« Load Upper Immediate » charger la valeur immédiate en partie haute), qui stocke une valeur 16-bit dans la partie haute du registre.

Regardons la sortie de *objdump* :

Listing 1.413: objdump

```
00000000 <f> :
   0:   3c021234        lui     v0,0x1234
   4:   03e00008        jr      ra
   8:   34425678        ori     v0,v0,0x5678
```

Charger une variable globale 32-bit dans un registre

```
unsigned int global_var=0x12345678;

unsigned int f2()
{
        return global_var;
};
```

Ceci est légèrement différent: LUI charge les 16-bit haut de *global_var* dans $2 (ou $V0) et ensuite LW charge les 16-bit bas en l'ajoutant au contenu de $2:

Listing 1.414: GCC 4.4.5 -O3 (résultat en sortie de l'assembleur)

```
f2 :
        lui     $2,%hi(global_var)
        lw      $2,%lo(global_var)($2)
        j       $31
        nop     ; slot de délai de branchement

        ...

global_var :
        .word   305419896
```

IDA reconnaît cette paire d'instructions fréquemment utilisée, donc il concatène en une seule instruction LW.

Listing 1.415: GCC 4.4.5 -O3 (IDA)

```
_f2 :
                lw      $v0, global_var
                jr      $ra
                or      $at, $zero      ; slot de délai de branchement

                ...

                .data
                .globl global_var
global_var :     .word 0x12345678         # DATA XREF : _f2
```

La sortie d'*objdump* est la même que la sortie assembleur de GCC: Affichons le code de relocation du fichier objet:

Listing 1.416: objdump

```
objdump -D filename.o

...

0000000c <f2> :
   c :   3c020000        lui     v0,0x0
  10:   8c420000        lw      v0,0(v0)
  14:   03e00008        jr      ra
  18:   00200825        move    at,at    ; slot de délai de branchement
  1c :   00200825        move    at,at
```

```
Désassemblage de la section .data :

00000000 <global_var> :
   0:   12345678        beq     s1,s4,159e4 <f2+0x159d8>

...

objdump -r filename.o

...

RELOCATION RECORDS FOR [.text]:
OFFSET   TYPE              VALUE
0000000c R_MIPS_HI16       global_var
00000010 R_MIPS_LO16       global_var

...
```

Nous voyons que l'adresse de *global_var* est écrite dans les instructions `LUI` et `LW` lors du chargement de l'exécutable: la partie haute de *global_var* se trouve dans la première (`LUI`), la partie basse dans la seconde (`LW`).

1.33.2 Autres lectures sur les MIPS

Dominic Sweetman, *See MIPS Run, Second Edition*, (2010).

Chapitre 2

Fondamentaux importants

2.1 Types intégraux

Un type intégral est un type de données dont les valeurs peuvent être converties en nombres. Les types intégraux comportent les nombres, les énumérations et les booléens.

2.1.1 Bit

Les valeurs booléennes sont une utilisation évidente des bits: 0 pour *faux* et 1 pour *vrai*.

Plusieurs valeurs booléennes peuvent être regroupées en un mot : Un mot de 32 bits contiendra 32 valeur booléennes, etc. On appelle *bitmap* ou *bitfield* un tel assemblage.

Cette approche engendre un surcoût de traitement: décalages, extraction, etc. A l'inverse l'utilisation d'un mot (ou d'un type *int*) pour chaque booléen gaspille de l'espace, au profit des performances.

Dans les environnements C/C++, la valeur 0 représente *faux* et toutes les autres valeurs *vrai*. Par exemple:

```
        if (1234)
                printf ("toujours exécuté\n");
        else
                printf ("jamais exécuté\n");
```

Une manière courante d'énumérer les caractères d'une chaîne en langage C:

```
        char *input=...;

        while(*input) // exécute le corps si le caractère *input est différent de zéro
        {
                // utiliser *input
                input++;
        };
```

2.1.2 Nibble

AKA demi-octet, tétrade. Représente 4 bits.

Toutes ces expressions sont toujours en usage.

Binary-Coded decimal (BCD[1])

Les demi-octets ont été utilisés par des CPU 4-bits tel que le Intel 4004 (utilisé dans les calculatrices).

On notera que la représentation *binary-coded decimal* (BCD) a été utilisée pour représenter les nombres sur 4 bits. L'entier 0 est représenté par la valeur 0b0000, l'entier 9 par 0b1001 tandis que les valeurs supérieurs ne sont pas utilisées. La valeur décimale 1234 est ainsi représentée par 0x1234. Il est évident que cette représentation n'est pas la plus efficace en matière d'espace.

Elle possède en revanche un avantage: la conversion des nombres depuis et vers le format BCD est extrêmement simple. Les nombres au format BCD peuvent être additionnés, soustraits, etc., au prix d'une opération supplémentaire de gestion des demi-retenues. Les CPUs x86 proposent pour cela quelques instructions assez rares: `AAA`/`DAA` (gestion de la demi-retenue après addition), `AAS`/`DAS` (gestion de la demi-retenue après soustraction), `AAM` (après multiplication), `AAD` (après division).

Le support par les CPUs des nombres au format BCD est la raison d'être des *half-carry flag* (sur 8080/Z80) et *auxiliary flag* (AF sur x86). Ils représentent la retenue générée après traitement des 4 bits de poids faible (d'un octet). Le drapeau est utilisé par les instructions de gestion de retenue ci-dessus.

Le livre [Peter Abel, *IBM PC assembly language and programming* (1987)] doit sa popularité à la facilité de ces conversions. Hormis ce livre, l'auteur de ces notes n'a jamais rencontré en pratique de nombres au format BCD, sauf dans certains *nombres magiques* (**??** on page ??), tels que lorsque la date de naissance d'un individu est encodé sous la forme 0x19791011—qui n'est autre qu'un nombre au format BCD.

Les instructions x86 destinées au traitement des nombres BCD ont parfois été utilisées à d'autres fins, le plus souvent non documentées, par exemple:

[1] Binary-Coded Decimal

```
        cmp al,10
        sbb al,69h
        das
```

Ce fragment de code abscons converties les nombres de 0 à 15 en caratères ASCII '0'..'9', 'A'..'F'.

Z80

Le processeur Z80 était un clone de la CPU 8 bits 8080 d'Intel. Par manque de place, il utilisait une UAL de 4 bits. Chaque opération impliquant deux nombres de 8 bits devait être traitée en deux étapes. Il en a découlé une utilisation naturelle des *half-carry flag*.

2.1.3 Caractère

A l'heure actuelle, l'utilisation de 8 bits par caractère est pratique courante. Il n'en a pas toujours été ainsi. Les cartes perforées utilisées pour les télétypes ne pouvaient comporter que 5 ou 6 perforations par caractères, et donc autant de bits.

Le terme *octet* met l'accent sur l'utilisation de 8 bits.: *fetchmail* est un de ceux qui utilise cette terminologie.

Sur les architectures à 36 bits, l'utilisation de 9 bits par caractère a été utilisée: un mot pouvait contenir 4 caractères. Ceci explique peut-être que le standard C/C++ indique que le type *char* doit supporter *au moins* 8 bits, mais que l'utilisation d'un nombre plus importants de bits est autorisé.

Par exemple, dans l'un des premiers ouvrage sur le langage C [2], nous trouvons :

```
char  one byte character (PDP-11, IBM360 : 8 bits; H6070 : 9 bits)
```

H6070 signifie probablement Honeywell 6070, qui comprenait des mots de 36 bits.

table ASCII standard

La représentation ASCII des caractères sur 7 bits constitue un standard, qui supporte donc 128 caractères différents. Les premiers logiciels de transport de mails fonctionnaient avec des codes ASCII sur 7 bits. Le standard MIME[3] nécessitait donc l'encodage des messages rédigés avec des alphabets non latins. Le code ASCII sur 7 bits a ensuite été augmenté d'un bit de parité qui a aboutit à la représentation sur 8 bits.

Les clefs de chiffrage utilisées par *Data Encryption Standard* (DES[4]) comportent 56 bits, soit 8 groupes de 7 bits ce qui laisse un espace pour un bit de parité dans chaque groupe.

La mémorisation de la table ASCII est inutile. Il suffit de se souvenir de certains intervalles. [0..0x1F] sont les caractères de contrôle (non imprimables). [0x20..0x7E] sont les caractères imprimables. Les codes à partir de la valeur 0x80 sont généralement utilisés pour les caractères non latins et pour certains caractères pseudo graphiques.

Quelques valeurs typiques à mémoriser sont : 0 (terminateur d'une chaîne de caractères en C, `'\0'` et C/C++); 0xA ou 10 (*fin de ligne*, `'\n'` en C/C++); 0xD ou 13 (*retour chariot*, `'\r'` en C/C++).

0x20 (espace).

CPUs 8 bits

Les processeurs x86 - descendants des CPUs 8080 8 bits - supportent la manipulation d'octet(s) au sein des registres. Les CPUs d'architecture RISC telles que les processeurs ARM et MIPS n'offrent pas cette possibilité.

[2] https://yurichev.com/mirrors/C/bwk-tutor.html
[3] Multipurpose Internet Mail Extensions
[4] Data Encryption Standard

2.1.4 Alphabet élargi

Il s'agit d'une tentative de supporter des langues non européennes en étendant le stockage d'un caractère à 16 bits. L'exemple le plus connu en est le noyau Windows NT et les fonctions win32 suffixées d'un *W*. Cet encodage est nommé UCS-2 ou UTF-16. Son utilisation explique la présence d'octets à zéro entre chaque caractère d'un texte en anglais ne comportant que des caractères latins.

En règle général, la notation *wchar_t* est un synonyme du type *short* qui utilise 16 bits.

2.1.5 Entier signé ou non signé

Certains s'étonneront qu'il existe un type de données entier non signé (positif ou nul) puisque chaque entier de ce type peut être représenté par un entier signé (positif ou négatif). Certes, mais le fait de ne pas avoir à utiliser un bit pour représenter le signe permet de doubler la taille de l'intervalle des valeurs qu'il est possible de représenter. Ainsi un octet signé permet de représenter les valeurs de -128 à +127, et l'octet non signé les valeurs de 0 à 255. Un autre avantage d'utiliser un type de données non signé est l'auto-documentation: vous définissez une variable qui ne peut pas recevoir de valeurs négatives.

L'absence de type de données non signées dans le langage Java a été critiqué. L'implémentation d'algorithmes cryptographiques à base d'opérations booléennes avec les seuls types de données signées est compliquée.

Une valeur telle que 0xFFFFFFFF (-1) est souvent utilisée, en particulier pour représenter un code d'erreur.

2.1.6 Mot

mot Le terme de 'mot' est quelque peu ambigu et dénote en général un type de données dont la taille correspond à celle d'un GPR. L'utilisation d'octets est pratique pour le stockage des caractères, mais souvent inadapté aux calculs arithmétiques.

C'est pourquoi, nombre de CPUs possèdent des GPRs dont la taille est de 16, 32 ou 64 bits. Les CPUs 8 bits tels que le 8080 et le Z80 proposent quant à eux de travailler sur des paires de registres 8 bits, dont chacune constitue un *pseudo-registre* de 16 bits. (*BC*, *DE*, *HL*, etc.). Les capacités des paires de registres du Z80 en font, en quelque sorte, un émulateur d'une CPU 16 bits.

En règle générale, un CPU présenté comme "CPU n-bits" possède des GPRs dont la taille est de n bits.

A une certaine époque, les disques durs et les barettes de RAM étaient caractérisés comme ayant *n* kilo-mots et non pas *b* kilooctets/megaoctets.

Par exemple, *Apollo Guidance Computer*[5] possède 2048 mots de RAM. S'agissant d'un ordinateur 16 bits, il y avait donc 4096 octets de RAM.

La mémoire magnétique du *TX-0*[6] était de 64K mots de 18 bits, i.e., 64 kilo-mots.

DECSYSTEM-2060[7] pouvait supporter jusqu'à 4096 kilo mots de *solid state memory* (i.e., hard disks, tapes, etc). S'agissant d'un ordinateur 36 bits, cela représentait 18432 kilo octets ou 18 mega octets.

---

int en C/C++ est presque systématiquement représenté par un mot. (L'architecture AMD64 fait exception car le type *int* possède une taille de 32 bits, peut-être pour une meilleure portabilité.)

Le type *int* est représenté sur 16 bits par le PDP-11 et les anciens compilateurs MS-DOS. Le type *int* est représenté sur 32 bits sur VAX, ainsi que sur l'architecture x86 à partir du 80386, etc.

De plus, dans les programmes C/C++, le type *int* est utilisé par défaut lorsque le type d'une variable n'est pas explicitement déclaré. Cette pratique peut apparaître comme un héritage du langage de programmation B[8].

---

[5] https://en.wikipedia.org/wiki/Apollo_Guidance_Computer
[6] https://en.wikipedia.org/wiki/TX-0
[7] https://en.wikipedia.org/wiki/DECSYSTEM-20
[8] http://yurichev.com/blog/typeless/

L'accès le plus rapide à une variable s'effectue lorsqu'elle est contenue dans un GPR, plus même qu'un ensemble de bits, et parfois même plus rapide qu'un octet (puisqu'il n'est pas besoin d'isoler un bit ou un octet au sein d'un GPR). Ceci reste vrai même lorsque le registre est utilisé comme compteur d'itération d'une boucle de 0 à 99.

En langage assembleur x86, un mot représente 16 bits, car il en était ainsi sur les processeurs 8086 16 bits. Un *Double word* représente 32 bits, et un *quad word* 64 bits. C'est pourquoi, les mots de 16 bits sont déclarés par `DW` en assembleur x86, ceux de 32 bits par `DD` et ceux de 64 bits par `DQ`.

Dans les architectures ARM, MIPS, etc... un mot représente 32 bits, on parlera alors de *demi-mot* pour les types sur 16 bits. En conséquence, un *double word* sur une architecture RISC 32 bits est un type de données qui représente 64 bits.

GDB utilise la terminologie suivante : *demi-mot* pour 16 bits, mot pour 32 bits et *mot géant* pour 64 bits.

Les environnements C/C++ 16 bits sur PDP-11 et MS-DOS définissent le type *long* comme ayant une taille de 32 bits, ce qui serait sans doute une abbréviation de *long word* ou de *long int*.

Les environnements C/C++ 32 bits définissent le type *long long* dont la taille est de 64 bits.

L'ambiguïté du terme *mot* est donc désormais évidente.

Dois-je utiliser le type *int* ?

Certains affirment que le type *int* ne doit jamais être utilisé, l'ambiguïté de sa définition pouvant être génératrice de bugs. A une certaine époque, la bibliothèque bien connue *lzhuf* utilisais le type *int* et fonctionnait parfaitement sur les architectures 16 bits. Portée sur une architecture pour laquelle le type *int* représentait 32 bits, elle pouvait alors crasher: `http://yurichev.com/blog/lzhuf/`.

Des types de données moins ambigus sont définis dans le fichier *stdint.h* : *uint8_t*, *uint16_t*, *uint32_t*, *uint64_t*, etc.

Donald E. Knuth fut l'un de ceux qui proposa[9] d'utiliser pour ces différents types des dénominations aux consonances distinctes: *octet/wyde/tetrabyte/octabyte*. Cette pratique est cependant moins courante que celle consistant à inclure directement dans le nom du type les termes *u* (*unsigned*) ainsi que le nombre de bits.

Ordinateurs à base de mots

En dépit de l'ambiguïté du terme mot, les ordinateurs modernes restent conçus sur ce concept: la RAM ainsi que tous les niveaux de mémoire cache demeurent organisés en mots et non pas en octets. La notion d'octet reste prépondérante en marketing.

Les accès aux adresses mémoire et cache alignées sur des frontières de mots est souvent plus performante que lorsque l'adresse n'est pas alignée.

Afin de rendre performante l'utilisation des structures de données, il convient toujours de de prendre en compte la longueur du mot du CPU sur lequel sera exécuté le programme lors de la définition des structures de données. Certains compilateurs - mais pas tous - prennent en charge cet alignement.

2.1.7 Registre d'adresse

Ceux qui ont fait leur premières armes sur les processeurs x86 32 et 64 bits, ou les processeurs RISC des années 90 tels que ARM, MIPS ou PowerPC prennent pour acquis que la taille du bus d'adresse est la même que celle d'un GPR ou d'un mot. Cependant, cette règle n'est pas toujours respectée sur d'autres architectures.

Le processeur 8 bits Z80 peut adresser 2^{16} octets, en utilisant une paire de registres 8 bits ou certains registres spécialisés (*IX*, *IY*). En outre sur ce processeur les registres *SP* et *PC* contiennent 16 bits.

Le super calculateur Cray-1 possèdent des registres généraux de 64-bit, et des registres d'adressage de 24 bits. Il peut donc adresser 2^{24} octets, soit (16 mega mots ou 128 mega octets). Dans les années 70,

[9] `http://www-cs-faculty.stanford.edu/~uno/news98.html`

la RAM était très coûteuse. Il paraissait alors inconcevable qu'un tel calculateur atteigne les 128 Mo. Dès lors pourquoi aurait-on utilisé des registres 64 bits pour l'adressage?

Les processeurs 8086/8088 utilisent un schéma d'adressage particulièrement bizarre: Les valeurs de deux registres de 16 bits sont additionnées de manière étrange afin d'obtenir une adresse sur 20 bits. S'agirait-il d'une sorte de virtualisation gadget (7.6 on page 644)? Les processeurs 8086 pouvaient en effet faire fonctionner plusieurs programmes côte à côte (mais pas simultanément bien sûr).

Les premiers processeurs ARM1 implémentent un artéfact intéressant:

> Un autre point intéressant est l'absence de quelques bits dans le registre PC. Le processeur ARM1 utilisant des adresses sur 26 bits, les 6 bits de poids fort ne sont pas utilisés. Comme toutes les adresses sont alignées sur une frontière de 32 bits, les deux bits les moins significatifs du registre PC sont toujours égaux à 0. Ces 8 bits sont non seulement inutilisés mais purement et simplement absents du processeur.

(http://www.righto.com/2015/12/reverse-engineering-arm1-ancestor-of.html)

En conséquence, il n'est pas possible d'affecter au registre PC une valeur dont l'un des deux bits de poids faible est différent de 0, pas plus qu'il n'est possible de positionner à 1 l'un des 6 bits de poids fort.

L'architecture x86-64 utilise des pointeurs et des adresses sur 64 bits, cependant en interne la largeur du bus d'adresse est de 48 bits, (ce qui est suffisant pour adresser 256 Tera octets de RAM).

2.1.8 Nombres

A quoi sont utilisés les nombres ?

Lorsque vous constatez que la valeur d'un registre de la CPU est modifié selon un certain motif, vous pouvez chercher à comprendre à quoi correspond ce motif. La capacité à déterminer le type de données qui découle de ce motif est une compétence précieuse pour le reverse engineer .

Booléen

Si le nombre alterne entre les valeurs 0 et 1, il y a des chances importantes pour qu'il s'agisse d'une valeur booléenne.

Compteur de boucle, index dans un tableau

Une variable dont la valeur augmente régulièrement en partant de 0, tel que 0, 1, 2, 3...— est probablement un compteur de boucle et/ou un index dans un tableau.

Nombres signés

Si vous constatez qu'une variable contient parfois des nombres très petits et d'autre fois des nombres très grands, tels que 0, 1, 2, 3, et 0xFFFFFFFF, 0xFFFFFFFE, 0xFFFFFFFD, il est probable qu'il s'agisse d'un entier signé sous forme de *two's complement* (2.2 on page 456) auquel cas les 3 dernières valeurs représentent en réalité -1, -2, -3.

Nombres sur 32 bits

Il existe des nombres tellement grands [10], qu'il existe une notation spéciale pour les représenter (Notation exponentielle de Knuth's [11]). De tels nombres sont tellement grands qu'ils s'avèrent peu pratiques pour l'ingénierie, les sciences ou les mathématiques.

La plupart des ingénieurs et des scientifiques sont donc ravis d'utiliser la notation IEEE 754 pour les nombres flottants à double précision, laquelle peut représenter des valeurs allant jusqu'à $1.8 \cdot 10^{308}$. (En comparaison, le nombre d'atomes dans l'univers observable est estimé être entre $4 \cdot 10^{79}$ et $4 \cdot 10^{81}$.)

[10] https://en.wikipedia.org/wiki/Large_numbers
[11] https://en.wikipedia.org/wiki/Knuth%27s_up-arrow_notation

De fait, la limite supérieure des nombres utilisés dans les opérations concrètes est très très inférieure.

Si vous examinez le codes source UNIX v6 pour PDP-11 [12], les *int* 16 bits y sont utilisés partout, tandis que le type *long* sur 32 bits ne l'est jamais.

Pareil à l'époque de MS-DOS: les *int* 16 bits étaient utilisés pratiquement pour tout (indice de tableau, compteur de boucle), tandis que le type *long* sur 32 bits ne l'était que rarement.

Durant l'avènement de l'architecture x86-64, il fut décidé que le type *int* conserverait une taille de 32 bits, probablement parce que l'utilisation d'un type *int* de 64 bits est encore plus rare.

Je dirais que les nombre sur 16 bits qui couvrent l'intervalle 0..65535 sont probablement les nombres les plus utilisés en informatique.

Ceci étant, si vous rencontrez des nombres sur 32 bits particulièrement élevé tels que 0x87654321, il existe une bonne chance qu'il s'agisse :

- Il peut toujours s'agir d'un entier sur 16 bits, mais signé lorsque la valeur est entre 0xFFFF8000 (-32768) et 0xFFFFFFFF (-1).
- une adresse mémoire (ce qui peut être vérifié en utilisant les fonctionnalités de gestion mémoire du débogueur).
- des octets compactés (ce qui peut être vérifié visuellement).
- un ensemble de drapeaux binaires.
- de la cryptographie (amateur).
- un nombre magique (**??** on page ??).
- un nombre flottant utilisant la représentation IEEE 754 (également vérifiable).

Il en va à peu près de même pour les valeurs sur 64 bits.

…donc un *int* sur 16 bits est suffisant pout à peu près n'importe quoi?

Il est intéressant de constater que dans [Michael Abrash, *Graphics Programming Black Book*, 1997 chapitre 13] nous pouvons lire qu'il existe pléthore de cas pour lesquels des variables sur 16 bits sont largement suffisantes. Dans le même temps, Michael Abrash se plaint que les CPUs 80386 et 80486 disposent de si peu de registres et propose donc de placer deux registres de 16 bits dans un registre de 32 bits et d'en effectuer des rotations en utilisant les instructions `ROR reg, 16` (sur 80386 et suivant) (`ROL reg, 16` fonctionne également) ou `BSWAP` (sur 80486 et suivant).

Cette approche rappelle celle du Z80 et de ses groupes de registres alternatifs (suffixés d'une apostrophe) vers lesquels le CPU pouvait être basculé (et inversement) au moyen de l'instruction EXX.

Taille des buffers

Lorssqu'un programmeur doit déclarer la taille d'un buffer, il utilise généralement une valeur de la forme 2^x (512 octets, 1024, etc.). Les valeurs de la forme 2^x sont faciles à reconnaître (1.22.5 on page 324) en décimal, en hexadécimal et en binaire.

Les programmeurs restent cependant des humains et conservent leur culture décimale. C'est pourquoi, dans le domaine des DBMS[13], la taille des champs textuels est souvent choisie sous la forme 10^x, 100, 200 par exemple. Ils pensent simplement « Okay, 100 suffira, attendez, 200 ira mieux ». Et bien sûr, ils ont raison.

La taille maximale du type *VARCHAR2* dans Oracle RDBMS est de 4000 caractères et non de 4096.

Il n'y a rien à redire à ceci, ce n'est qu'un exemple d'utilisation des nombres sous la forme d'un multiple d'une puissance de dix.

[12] http://minnie.tuhs.org/Archive/PDP-11/Distributions/research/Dennis_v6/
[13] Database Management Systems

Addresse

Garder à l'esprit une cartographie approximative de l'occupation mémoire du processus que vous déboguez est toujours une bonne idée. Ainsi, beaucoup d'exécutables win32 démarrent à l'adresse 0x00401000, donc une adresse telle que 0x00451230 se situe probablement dans sa section exécutable. Vous trouverez des adresses de cette sorte dans le registre EIP.

La pile est généralement située à une adresse inférieure à

Beaucoup de débogueurs sont capables d'afficher la cartographie d'occupation mémoire du processus débogué, par exemple: 1.9.3 on page 79.

Une adresse qui augmente par pas de 4 sur une architecture 32-bit, ou par pas de 8 sur une architecture 64-bit constitue probablement l'énumération des adresses des éléments d'un tableau.

Il convient de savoir que win32 n'utilise pas les adresses inférieures à 0x10000, donc si vous observez un nombre inférieur à cette valeur, ce ne peut être une adresse (voir aussi `https://msdn.microsoft.com/en-us/library/ms810627.aspx`).

De toute manière, beaucoup de débogueurs savent vous indiquer si la valeur contenue dans un registre peut représenter l'adresse d'un élément. OllyDbg peut également vous afficher le contenu d'une chaîne de caractères ASCII si la valeur du registre est l'adresse d'une telle chaîne.

Drapeaux

Si vous observez une valeur pour laquelle un ou plusieurs bits changent de valeur de temps en temps tel que 0xABCD1234 → 0xABCD1434 et retour, il s'agit probablement d'un ensemble de drapeaux ou bitmap.

Compactage de caractères

Quand *strcmp()* ou *memcmp()* copient un buffer, ils traitent 4 (ou 8) octets à la fois. Donc, si une chaîne de caractères « 4321 » est recopié à une autre adresse, il adviendra un moment où vous observerez la valeur 0x31323334 dans un registre. Il s'agit du bloc de 4 caractères traité comme un entier sur 32 bits.

2.2 Représentations des nombres signés

Il existe plusieurs méthodes pour représenter les nombres signées[14], mais le « complément à deux » est la plus populaire sur les ordinateurs.

Voici une table pour quelques valeurs d'octet:

[14] wikipedia

binaire	hexadécimal	non-signé	signé
01111111	0x7f	127	127
01111110	0x7e	126	126
...			
00000110	0x6	6	6
00000101	0x5	5	5
00000100	0x4	4	4
00000011	0x3	3	3
00000010	0x2	2	2
00000001	0x1	1	1
00000000	0x0	0	0
11111111	0xff	255	-1
11111110	0xfe	254	-2
11111101	0xfd	253	-3
11111100	0xfc	252	-4
11111011	0xfb	251	-5
11111010	0xfa	250	-6
...			
10000010	0x82	130	-126
10000001	0x81	129	-127
10000000	0x80	128	-128

La différence entre nombres signé et non-signé est que si l'on représente `0xFFFFFFFE` et `0x00000002` comme non signées, alors le premier nombre (4294967294) est plus grand que le second (2). Si nous les représentons comme signés, le premier devient -2, et il est plus petit que le second. C'est la raison pour laquelle les sauts conditionnels (1.14 on page 124) existent à la fois pour des opérations signées (p. ex. `JG`, `JL`) et non-signées (`JA`, `JB`).

Par souci de simplicité, voici ce qu'il faut retenir:

- Les nombres peuvent être signés ou non-signés.
- Types C/C++ signés:
  - `int64_t` (-9,223,372,036,854,775,808 .. 9,223,372,036,854,775,807) (- 9.2.. 9.2 quintillions) ou `0x8000000000000000..0x7FFFFFFFFFFFFFFF`),
  - *int* (-2,147,483,648..2,147,483,647 (- 2.15.. 2.15Gb) ou `0x80000000..0x7FFFFFFF`),
  - *char* (-128..127 ou `0x80..0x7F`),
  - `ssize_t`.

  Non-signés:
  - `uint64_t` (0..18,446,744,073,709,551,615 (18 quintillions) ou `0..0xFFFFFFFFFFFFFFFF`),
  - `unsigned int` (0..4,294,967,295 (4.3Gb) ou `0..0xFFFFFFFF`),
  - `unsigned char` (0..255 ou `0..0xFF`),
  - `size_t`.
- Les types signés ont le signe dans le MSB : 1 signifie « moins », 0 signifie « plus ».
- Étendre à un type de données plus large est facile: 1.28.5 on page 407.
- La négation est simple: il suffit d'inverser tous les bits et d'ajouter 1.

  Nous pouvons garder à l'esprit qu'un nombre de signe opposé se trouve de l'autre côté, à la même distance de zéro. L'addition d'un est nécessaire car zéro se trouve au milieu.
- Les opérations d'addition et de soustraction fonctionnent bien pour les valeurs signées et non-signées. Mais pour la multiplication et la division, le x86 possède des instructions différentes: `IDIV`/`IMUL` pour les signés et `DIV`/`MUL` pour les non-signés.
- Voici d'autres instructions qui fonctionnent avec des nombres signés: `CBW`/`CWD`/`CWDE`/`CDQ`/`CDQE` (**??** on page ??), `MOVSX` (1.17.1 on page 201), `SAR` (**??** on page ??).

Une table avec quelques valeurs négatives et positives (**??**) ressemble à un thermomètre avec une échelle Celsius. C'est pourquoi l'addition et la soustraction fonctionnent bien pour les nombres signés et non-signés: si le premier opérande est représenté par une marque sur un thermomètre, et que l'on doit ajouter un second opérande, et qu'il est positif, nous devons juste augmenter la marque sur le thermomètre de la

valeur du second opérande. Si le second opérande est négatif, alors nous baissons la marque de la valeur absolue du second opérande.

L'addition de deux nombres négatifs fonctionne comme suit. Par exemple, nous devons ajouter -2 et -3 en utilisant des registres 16-bit. -2 et -3 sont respectivement 0xfffe et 0xfffd. si nous les ajoutons comme nombres non-signés, nous obtenons 0xfffe+0xfffd=0x1fffb. Mais nous travaillons avec des registres 16-bit, le résultat est *tronqué*, le premier 1 est perdu, et il reste 0xfffb et c'est -5. Ceci fonctionne car -2 (ou 0xfffe) peut être représenté en utilisant des mots simples comme suit: "il manque 2 à la valeur maximale d'un registre 16-bit + 1". -3 peut être représenté comme "...il manque 3 à la valeur maximale jusqu'à ...". La valeur maximale d'un registre 16-bit + 1 est 0x10000. Pendant l'addition de deux nombres et en *tronquant* modulo 2^{16}, il manquera $2+3=5$.

2.2.1 Utiliser IMUL au lieu de MUL

Un exemple comme liste. **??** où deux valeurs non signées sont multipliées compile en liste. **??** où `IMUL` est utilisé à la place de `MUL`.

Ceci est une propriété importante des instructions `MUL` et `IMUL`. Tout d'abord, les deux produisent une valeur 64-bit si deux valeurs 32-bit sont multipliées, ou une valeur 128-bit si deux valeurs 64-bit sont multipliées (le plus grand produit dans un environnement 32-bit est `0xffffffff*0xffffffff=0xfffffffe00000001`). Mais les standards C/C++ n'ont pas de moyen d'accèder à la moitié supérieure du résultat, et un produit a toujours la même taille que ses multiplicandes. Et les deux instructions `MUL` et `IMUL` fonctionnent de la même manière si la moitié supérieure est ignorée, i.e, elles produisent le même résultat dans la partie inférieure. Ceci est une propriété importante de la façon de représenter les nombre en « complément à deux ».

Donc, le compilateur C/C++ peut utiliser indifféremment ces deux instructions.

Mais `IMUL` est plus flexible que `MUL`, car elle prend n'importe quel(s) registre(s) comme source, alors que `MUL` nécessite que l'un des multiplicandes soit stocké dans le registre AX/EAX/RAX Et même plus que ça: `MUL` stocke son résultat dans la paire `EDX:EAX` en environnement 32-bit, ou `RDX:RAX` dans un 64-bit, donc elle calcule toujours le résultat complet. Au contraire, il est possible de ne mettre qu'un seul registre de destination lorsque l'on utilise `IMUL`, au lieu d'une paire, et alors le CPU calculera seulement la partie basse, ce qui fonctionne plus rapidement [voir Torborn Granlund, *Instruction latencies and throughput for AMD and Intel x86 processors*[15]].

Cela étant considéré, les compilateurs C/C++ peuvent générer l'instruction `IMUL` plus souvent que `MUL`.

Néanmoins, en utilisant les fonctions intrinsèques du compilateur, il est toujours possible d'effectuer une multiplication non signée et d'obtenir le résultat *complet*. Ceci est parfois appelé *multiplication étendue*. MSVC a une fonction intrinsèque pour ceci, appelée *__emul*[16] et une autre: *_umul128*[17]. GCC offre le type de données *__int128*, et dans le cas de multiplicandes 64-bit, ils sont déjà promus en 128-bit, puis le produit est stocké dans une autre valeur *__int128*, puis le résultat est décalé de 64 bits à droite, et vous obtenez la moitié haute du résultat[18].

Fonction MulDiv() dans Windows

Windows possède la fonction MulDiv()[19], fonction qui fusionne une multiplication et une division, elle multiplie deux entiers 32-bit dans une valeur 64-bit intermédiaire et la divise par un troisième entier 32-bit. C'est plus facile que d'utiliser deux fonctions intrinsèques, donc les développeurs de Microsoft ont écrit une fonction spéciale pour cela. Et il semble que ça soit une fonction très utilisée, à en juger par son utilisation.

[15] http://yurichev.com/mirrors/x86-timing.pdf
[16] https://msdn.microsoft.com/en-us/library/d2s81xt0(v=vs.80).aspx
[17] https://msdn.microsoft.com/library/3dayytw9%28v=vs.100%29.aspx
[18] Exemple: http://stackoverflow.com/a/13187798
[19] https://msdn.microsoft.com/en-us/library/windows/desktop/aa383718(v=vs.85).aspx

2.2.2 Quelques ajouts à propos du complément à deux

> Exercice 2-1. Écrire un programme pour déterminer les intervalles des variables `char`, `short`, `int`, et `long`, signées et non signées, en affichant les valeurs appropriées depuis les headers standards et par calcul direct.
>
> Brian W. Kernighan, Dennis M. Ritchie, *The C Programming Language*, 2ed, (1988)

Obtenir le nombre maximum de quelques mots

Le maximum d'un nombre non signé est simplement un nombre où tous les bits sont mis: *0xFF....FF* (ceci est -1 si le mot est traité comme un entier signé). Donc, vous prenez un mot, vous mettez tous les bits et vous obtenez la valeur:

```
#include <stdio.h>

int main()
{
        unsigned int val=~0; // changer à "unsigned char" pour obtenir la valeur maximale pour ↙
    ↳ un octet 8-bit non-signé
        // 0-1 fonctionnera aussi, ou juste -1
        printf ("%u\n", val); // %u pour unsigned
};
```

C'est 4294967295 pour un entier 32-bit.

Obtenir le nombre maximum de quelques mots signés

Le nombre signé minimum est encodé en *0x80....00*, i.e., le bit le plus significatif est mis, tandis que tous les autres sont à zéro. Le nombre maximum signé est encodé de la même manière, mais tous les bits sont inversés: *0x7F....FF*.

Déplaçons un seul bit jusqu'à ce qu'il disparaisse:

```
#include <stdio.h>

int main()
{
        signed int val=1; // changer à "signed char" pour trouver les valeurs pour un octet ↙
    ↳ signé
        while (val!=0)
        {
                printf ("%d %d\n", val, ~val);
                val=val<<1;
        };
};
```

La sortie est:

```
...

536870912 -536870913
1073741824 -1073741825
-2147483648 2147483647
```

Les deux dernier nombres sont respectivement le minimum et le maximum d'un entier signé 32-bit *int*.

2.3 Dépassement d'entier

J'ai intentionnellement mis cette section après celle sur la représentation des nombres signés.

Tout d'abord, regardons l'implémentation de la fonction *itoa()* dans [Brian W. Kernighan, Dennis M. Ritchie, *The C Programming Language*, 2ed, (1988)] :

```
void itoa(int n, char s[])
{
        int i, sign;
        if ((sign = n) < 0) /* record sign */
                n = -n; /* make n positive */
        i = 0;
        do { /* generate digits in reverse order */
                s[i++] = n % 10 + '0'; /* get next digit */
        } while ((n /= 10) > 0); /* delete it */
        if (sign < 0)
                s[i++] = '-';
        s[i] = '\0';
        strrev(s);
}
```

(Le code source complet: https://github.com/DennisYurichev/RE-for-beginners/blob/master/fundament itoa_KR.c)

Elle a un bogue subtil. Essayez de le trouver. Vous pouvez téléchargez le code source, le compiler, etc. La réponse se trouve à la page suivante.

De [Brian W. Kernighan, Dennis M. Ritchie, *The C Programming Language*, 2ed, (1988)] :

> Exercice 3-4. Dans un système de représentation des nombres par complément à deux notre version de *itoa* ne peut pas traiter le plus grand nombre négatif. c'est-à- dire la valeur de *n* égale à $-(2^{wordsize-1})$. Pourquoi ? Modifiez *itoa* de façon à ce qu'elle traite ce cas correctement, quelle que soit la machine utilisée.

La réponse est: la fonction ne peut pas traiter le plus grand nombre négatif (INT_MIN ou 0x80000000 ou -2147483648) correctement.

Comment changer le signe? Inverser tous les bits et ajouter 1. Si vous inversez tous les bits de la valeur INT_MIN (0x80000000), ça donne 0x7fffffff. Ajouter 1 et vous obtenez à nouveau 0x80000000. C'est un artefact important du système de complément à deux.

Lectures complémentaires:

- blexim – Basic Integer Overflows[20]
- Yannick Moy, Nikolaj Bjørner, et David Sielaff – Modular Bug-finding for Integer Overflows in the Large: Sound, Efficient, Bit-precise Static Analysis[21]

2.4 AND

2.4.1 Tester si une valeur est alignée sur une limite de 2^n

Si vous devez vérifier si votre valeur est divisible par un nombre 2^n (comme 1024, 4096, etc.) sans reste, vous pouvez utiliser l'opérateur % en C/C++, mais il y a un moyen plus simple. 4096 est 0x1000, donc il a toujours les $4 * 3 = 12$ bits inférieurs à zéro.

Ce dont vous avez besoin est simplement:

```
if (value&0xFFF)
{
        printf ("la valeur n'est pas divisible par 0x1000 (ou 4096)\n");
        printf ("a propos, le reste est %d\n", value&0xFFF);
}
else
        printf ("la valeur est divisible par 0x1000 (ou 4096)\n");
```

Autrement dit, ce code vérifie si il y a un bit mis parmi les 12 bits inférieurs. Un effet de bord, les 12 bits inférieurs sont toujours le reste de la division d'une valeur par 4096 (car une division par 2^n est simplement un décalage à droite, et les bits décalés (et perdus) sont les bits du reste).

Même principe si vous voulez tester si un nombre est pair ou impair:

```
if (value&1)
        // odd
else
        // even
```

Ceci est la même chose que de diviser par 2 et de prendre le reste de 1-bit.

2.4.2 Encodage cyrillique KOI-8R

Il fût un temps où la table ASCII 8-bit n'était pas supportée par certains services Internet, incluant l'email. Certains supportaient, d'autres—non.

Il fût aussi un temps, où les systèmes d'écriture non-latin utilisaient la seconde moitié de la table ASCII pour stocker les caractères non-latin. Il y avait plusieurs encodages cyrillique populaires, mais KOI-8R (conçu par Andrey "ache" Chernov) est plutôt unique en comparaison avec les autres.

[20] http://phrack.org/issues/60/10.html
[21] https://yurichev.com/mirrors/SMT/z3prefix.pdf

Fig. 2.1: KOI8-R table

On peut remarquer que les caractères cyrilliques sont alloués presque dans la même séquence que les caractères Latin. Ceci conduit à une propriété importante: si tout les 8ème bits d'un texte encodé en Cyrillique sont mis à zéro, le texte est transformé en un texte translittéré avec des caractères latin à la place de cyrillique. Par exemple, une phrase en Russe:

> Мой дядя самых честных правил, Когда не в шутку занемог, Он уважать себя заставил, И лучше выдумать не мог.

...s'il est encodé en KOI-8R et que le 8ème bit est supprimé, il est transformé en:

> mOJ DQDQ SAMYH ^ESTNYH PRAWIL, kOGDA NE W [UTKU ZANEMOG, oN UWAVATX SEBQ ZASTAWIL, i LU^[E WYDUMATX NE MOG.

...ceci n'est peut-être très esthétiquement attrayant, mais ce texte est toujours lisible pour les gens de langue maternelle russe.

De ce fait, un texte cyrillique encodé en KOI-8R, passé à travers un vieux service 7-bit survivra à la translittération, et sera toujours un texte lisible.

Supprimer le 8ème bit transpose automatiquement un caractère de la seconde moitié de n'importe quelle table ASCII 8-bit dans la première, à la même place (regardez la flèche rouge à droite de la table). Si le caractère était déjà dans la première moitié (i.e., il était déjà dans la table ASCII 7-bit standard), il n'est pas transposé.

Peut-être qu'un texte translittéré est toujours récupérable, si vous ajoutez le 8ème bit aux caractères qui ont l'air d'avoir été translittérés.

L'inconvénient est évident: les caractères cyrilliques alloués dans la table KOI-8R ne sont pas dans le même ordre dans l'alphabet Russe/Bulgare/Ukrainien/etc., et ce n'est pas utilisable pour le tri, par exemple.

2.5 AND et OR comme soustraction et addition

2.5.1 Chaînes de texte de la ROM du ZX Spectrum

Ceux qui ont étudié une fois le contenu de la ROM du ZX Spectrum, ont probablement remarqués que le dernier caractère de chaque chaîne de texte est apparemment absent.

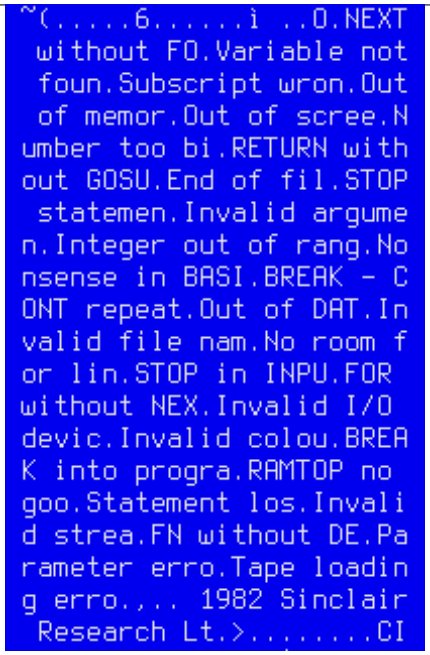


Fig. 2.2: Partie de la ROM du ZX Spectrum

Ils sont présents, en fait.

Voici un extrait de la ROM du ZX Spectrum 128K désassemblée:

```
L048C :  DEFM "MERGE erro"           ; Report 'a'.
         DEFB 'r'+$80
L0497 :  DEFM "Wrong file typ"       ; Report 'b'.
         DEFB 'e'+$80
L04A6 :  DEFM "CODE erro"            ; Report 'c'.
         DEFB 'r'+$80
L04B0 :  DEFM "Too many bracket"     ; Report 'd'.
         DEFB 's'+$80
L04C1 :  DEFM "File already exist"   ; Report 'e'.
         DEFB 's'+$80
```

(http://www.matthew-wilson.net/spectrum/rom/128_ROM0.html)

Le dernier caractère a le bit le plus significatif mis, ce qui marque la fin de la chaîne. Vraisemblablement que ça a été fait pour économiser de l'espace. Les vieux ordinateurs 8-bit avaient une mémoire très restreinte.

Les caractères de tous les messages sont toujours dans la table ASCII 7-bit standard, donc il est garanti que le 8ème bit n'est jamais utilisé pour les caractères.

Pour afficher une telle chaîne, nous devons tester le MSB de chaque octet, et s'il est mis, nous devons l'effacer, puis afficher le caractère et arrêter. Voici un exemple en C:

```
unsigned char hw[]=
{
        'H',
        'e',
        'l',
        'l',
        'o'|0x80
};

void print_string()
{
        for (int i=0; ;i++)
        {
                if (hw[i]&0x80) // check MSB
                {
                        // clear MSB
                        // (en d'autres mots, les effacer tous, mais laisser les 7 bits ↙
    ↳ inférieurs intacts)
```

```
                        printf ("%c", hw[i] & 0x7F);
                        // stop
                        break;
                };
                printf ("%c", hw[i]);
        };
};
```

Maintenant ce qui est intéressant, puisque le 8ème bit est le bit le plus significatif (dans un octet), c'est que nous pouvons le tester, le mettre et le supprimer en utilisant des opérations arithmétiques au lieu de logiques:

Je peux récrire mon exemple en C:

```
unsigned char hw[]=
{
        'H',
        'e',
        'l',
        'l',
        'o'+0x80
};

void print()
{
        for (int i=0; ;i++)
        {
                // hw[] doit avoir le type 'unsigned char'
                if (hw[i] >= 0x80) // tester le MSB
                {
                        printf ("%c", hw[i]-0x80); // clear MSB
                        // stop
                        break;
                };
                printf ("%c", hw[i]);
        };
};
```

Par défaut, *char* est un type signé en C/C++, donc pour le comparer avec une variable comme 0x80 (qui est négative (-128) si elle est traitée comme signée), nous devons traiter chaque caractère dans le texte du message comme non signé.

Maintenant si le 8ème bit est mis, le nombre est toujours supérieur ou égal à 0x80. Si le 8ème est à zéro, le nombre est toujours plus petit que 0x80.

Et même plus que ça: si le 8ème bit est mis, il peut être effacé en soustrayant 0x80, rien d'autre. Si il n'est pas mis avant, toutefois, la soustraction va détruire d'autres bits.

De même, si le 8ème est à zéro, il est possible de le mettre en ajoutant 0x80. Mais s'il est déjà mis, l'opération d'addition va détruire d'autres bits.

En fait, ceci est valide pour n'importe quel bit. Si le 4ème bit est à zéro, vous pouvez le mettre juste en ajoutant 0x10: 0x100+0x10 = 0x110. Si le 4ème bit est mis, vous pouvez l'effacer en soustrayant 0x10: 0x1234-0x10 = 0x1224.

Ça fonctionne, car il n'y a pas de retenue générée pendant l'addition/soustraction. Elle le serait, toutefois, si le bit est déjà à 1 avant l'addition, ou à 0 avant la soustraction.

De même, addition/soustraction peuvent être remplacées en utilisant une opération OR/AND si deux conditions sont réunies: 1) vous voulez ajouter/soustraire un nombre de la forme 2^n ; 2) la valeur du bit d'indice n dans la valeur source est 0/1.

Par exemple, l'addition de 0x20 est la même chose que OR-er la valeur avec 0x20 sous la condition que ce bit est à zéro avant: 0x1204|0x20 = 0x1204+0x20 = 0x1224.

La soustraction de 0x20 est la même chose que AND-er la valeur avec 0x20 (0x....FFDF), mais si ce bit est mis avant: 0x1234&(~0x20) = 0x1234&0xFFDF = 0x1234-0x20 = 0x1214.

À nouveau, ceci fonctionne parce qu'il n'y a pas de retenue générée lorsque vous ajoutez le nombre 2^n et que ce bit n'est pas à 1 avant.

Cette propriété de l'algèbre booléenne est importante, elle vaut la peine d'être comprise et gardée à l'esprit.

Un autre exemple dans ce livre: 3.16.3 on page 542.

2.6 XOR (OU exclusif)

X0R est très utilisé lorsque l'on doit inverser un ou plusieurs bits spécifiques. En effet, l'opération X0R appliquée avec 1 inverse effectivement un bit:

entrée A	entrée B	sortie
0	0	0
0	1	1
1	0	1
1	1	0

Et vice-versa, l'opération X0R appliquée avec 0 ne fait rien, i.e, c'est une opération sans effet. C'est une propriété très importante de l'opération X0R et il est fortement recommandé de s'en souvenir.

2.6.1 Langage courant

L'opération XOR est présente dans le langage courant. Lorsque quelqu'un demande "s'il te plaît, achète des pommes ou des bananes", ceci signifie généralement "achète le premier item ou le second, mais pas les deux"—ceci est exactement un OU exclusif, car le OU logique signifierait "les deux objets sont bien aussi".

Certaines personnes suggèrent que "et/ou" devraient être utilisés dans le langage courant pour mettre l'accent sur le fait que le OU logique est utilisé à la place du OU exclusif: https://en.wikipedia.org/wiki/And/or.

2.6.2 Chiffrement

XOR est beaucoup utilisé à la fois par le chiffrement amateur (6.1) et *réel* (au moins dans le *réseau de Feistel*).

XOR est trés pratique ici car: $cipher\_text = plain\_text \oplus key$ et alors: $(plain\_text \oplus key) \oplus key = plain\_text$.

2.6.3 RAID4

RAID4 offre une méthode très simple pour protéger les disques dur. Par exemple, il y a quelques disques (D_1, D_2, D_3, etc.) et un disque de parité (P). Chaque bit/octet écrit sur le disque de parité est calculé et écrit au vol:

$$P = D_1 \oplus D_2 \oplus D_3 \tag{2.1}$$

Si n'importe lequel des disques est défaillant, par exemple, D_2, il est restauré en utilisant la même méthode:

$$D_2 = D_1 \oplus P \oplus D_3 \tag{2.2}$$

Si le disque de parité est défaillant, il est restauré en utilisant la méthode 2.1. Si deux disques sont défaillants, alors il n'est pas possible de les restaurer les deux.

RAID5 est plus avancé, mais cet propriété de XOR y est encore utilisé.

C'est pourquoi les contrôleurs RAID ont des "accélérateurs XOR" matériel pour aider les opérations XOR sur de larges morceaux de données écrites au vol. Depuis que les ordinateurs deviennent de plus en plus rapide, cela peut maintenant être effectué au niveau logiciel, en utilisant SIMD.

2.6.4 Algorithme d'échange XOR

C'est difficile à croire, mais ce code échangent les valeurs dans EAX et EBX sans l'aide d'aucun autre registre ni d'espace mémoire.

```
xor eax, ebx
xor ebx, eax
xor eax, ebx
```

Cherchons comment ça fonctionne. D'abord, récrivons le afin de retirer le langage d'assemblage x86:

```
X = X XOR Y
Y = Y XOR X
X = X XOR Y
```

Qu'est ce que X et Y valent à chaque étape? Gardez à l'esprit cette règle simple: $(X \oplus Y) \oplus Y = X$ pour toutes valeurs de X et Y.

Regardons, après la 1ère étape X vaut $X \oplus Y$; après la 2ème étape Y vaut $Y \oplus (X \oplus Y) = X$; après la 3ème étape X vaut $(X \oplus Y) \oplus X = Y$.

Difficile de dire si on doit utiliser cette astuce, mais elle est un bon exemple de démonstration des propriétés de XOR.

L'article de Wikipédia (https://en.wikipedia.org/wiki/XOR_swap_algorithm) donne d'autres explication: l'addition et la soustraction peuvent être utilisées à la place de XOR:

```
X = X + Y
Y = X - Y
X = X - Y
```

Regardons: après la 1ère étape X vaut $X + Y$; après la 2ème étape Y vaut $X + Y - Y = X$; après la 3ème étape X vaut $X + Y - X = Y$.

2.6.5 liste chaînée XOR

Une liste doublement chaînée est une liste dans laquelle chaque élément a un lien sur l'élément précédent et sur le suivant. Ainsi, il est très facile de traverser la liste dans un sens ou dans l'autre. `std::list`, qui implémente les listes doublement chaînées en C++, est également examiné dans ce livre: **??**.

Donc chaque élément possède deux pointeurs. Est-il possible, peut-être dans un environnement avec peu de mémoire, de garder toutes ces fonctionnalités, avec un seul pointeur au lieu de deux? Oui, si la valeur de $prev \oplus next$ est stockée dans cette cellule mémoire, qui est habituellement appelé "lien".

Peut-être que nous pouvons dire que l'adresse de l'élément précédent est "chiffrée" en utilisant l'adresse de l'élément suivant et réciproquement: l'adresse de l'élément suivant est "chiffrée" en utilisant l'adresse de l'élément précédent.

Lorsque nous traversons cette liste en avant, nous connaissons toujours l'adresse de l'élément précédent, donc nous pouvons "déchiffrer" ce champ et obtenir l'adresse de l'élément suivant. De même, il est possible de traverser cette liste en arrière, "déchiffrer" ce champ en utilisant l'adresse de l'élément suivant.

Mais il n'est pas possible de trouver l'adresse de l'élément précédent ou suivant d'un élément spécifique sans connaître l'adresse du premier.

Deux éléments pour compléter cette solution: le premier élément aura toujours l'adresse de l'élément suivant sans aucun XOR, le dernier élément aura l'adresse du premier élément sans aucun XOR.

Maintenant, résumons. Ceci est un exemple d'une liste doublement chaînée de 5 éléments. A_x est l'adresse de l'élément.

adresse	contenu du champ *link*
A_0	A_1
A_1	$A_0 \oplus A_2$
A_2	$A_1 \oplus A_3$
A_3	$A_2 \oplus A_4$
A_4	A_3

À nouveau, il est difficile de dire si quelqu'un doit utiliser ce truc rusé, mais c'est une bonne démonstration des propriétés de XOR. Avec l'algorithme d'échange avec XOR, l'article de Wikipédia montre des méthodes pour utiliser l'addition et la soustraction au lieu de XOR: https://en.wikipedia.org/wiki/XOR_linked_list.

2.6.6 hachage Zobrist / hachage de tabulation

Si vous travaillez sur un moteur de jeu d'échec, vous traversez l'arbre de jeu de très nombreuses fois par seconde, et souvent, vous pouvez rencontrer une même position, qui a déjà été étudiée.

Donc, vous devez utiliser un méthode pour stocker quelque part les positions déjà calculées. Mais les positions d'un jeu d'échec demandent beaucoup de mémoire, et une fonction de hachage peut être utilisée à la place.

Voici un moyen de compresser une position d'échecs dans une valeur 64-bit, appelée le hachage de Zobrist:

```
// nous avons un échiquier de 8*8 et 12 pièces (6 pour le côté blanc et 6 pour le noir)

uint64_t table[12][8][8]; // remplir avec des valeurs aléatoires

int position[8][8]; // pour chaque case de l'échiquier. 0 - no piece. 1..12 - piece

uint64_t hash;

for (int row=0; row<8; row++)
        for (int col=0; col<8; col++)
        {
                int piece=position[row][col];

                if (piece!=0)
                        hash=hash^table[piece][row][col];
        };

return hash;
```

Maintenant la partie la plus intéressante: si la position suivante (modifiée) diffère seulement d'une pièce (déplacée), vous ne devez pas recalculer le hachage pour la position complète, tout ce que vous devez faire est:

```
hash=...; // (déjà calculé)

// soustraire l'information à propos de l'ancienne pièce :
hash=hash^table[old_piece][old_row][old_col];

// ajouter l'information à propos de la nouvelle pièce :
hash=hash^table[new_piece][new_row][new_col];
```

2.6.7 À propos

Le *OR* usuel est parfois appelé *OU inclusif* (ou même *IOR*), par opposition au *OU exclusif*. C'est ainsi dans la bibliothèque Python *operator* : il y est appelé *operator.ior*.

2.6.8 AND/OR/XOR au lieu de MOV

`OR reg, 0xFFFFFFFF` mets tous les bits à 1, en conséquence, peu importe ce qui se trouvait avant dans le registre, il sera mis à -1. `OR reg, -1` est plus court que `MOV reg, -1`, donc MSVC utilise OR au lieu de ce dernier suivant, par exemple: 3.15.1 on page 532.

De même, `AND reg, 0` efface tous les bits, par conséquent, elle se comporte comme `MOV reg, 0`.

`XOR reg, reg`, peu importe ce qui se trouvait précédemment dans le registre, efface tous les bits et se comporte donc comme `MOV reg, 0`.

2.7 Comptage de population

L'instruction POPCNT est « population count » (comptage de population) (AKA poids de Hamming). Elle compte simplement les nombres de bits mis dans une valeur d'entrée.

Par effet de bord, l'instruction POPCNT (ou opération) peut-être utilisée pour déterminer si la valeur est de la forme 2^n. Puisqu'un nombre de la forme 2^n a un seul bit à 1, le résultat de POPCNT (ou opération) sera toujours 1.

Par exemple, j'ai écrit une fois un scanner de chaînes en base64 pour chercher des choses intéressantes dans les fichiers binaires[22]. Et il y beaucoup de déchets et de faux-positifs, donc j'ai ajouté une option pour filtrer les blocs de données ayant une taille de 2^n octets (i.e., 256 octets, 512, 1024, etc.). La taille du bloc est testée simplement comme ceci:

```
if (popcnt(size)==1)
        // OK
...
```

Cette instruction est aussi connue en tant qu'« instruction NSA[23] » à cause de rumeurs:

> Cette branche de la cryptographie croît très rapidement et est très influencée politiquement. La plupart des conceptions sont secrètes ; la majorité des systèmes de chiffrement militaire utilisés aujourd'hui est basée sur les RDRL. De fait, la plupart des ordinateurs CRAY (Cray 1, Cray X-MP, Cray Y-MP) possèdent une instruction curieuse du nom de « comptage de la population ». Elle compte les 1 dans un registre et peut être utilisée à la fois pour calculer efficacement la distance de Hamming entre deux mots binaires et pour réaliser une version vectorielle d'un RDRL. Certains la nomment l'instruction canonique de la NSA, elle est demandée sur presque tous les contrats d'ordinateur.

[Bruce Schneier, *Applied Cryptography*, (John Wiley & Sons, 1994)][24]
Traduction française: [Cryptographie appliquée : protocoles, algorithmes et codes source en C / Bruce Schneier ; traduction de Laurent Viennot][25]

2.8 Endianness

L'endianness (boutisme) est la façon de représenter les valeurs en mémoire.

2.8.1 Big-endian

La valeur 0x12345678 est représentée en mémoire comme:

adresse en mémoire	valeur de l'octet
+0	0x12
+1	0x34
+2	0x56
+3	0x78

Les CPUs big-endian comprennent les Motorola 68k, IBM POWER.

2.8.2 Little-endian

La valeur 0x12345678 est représentée en mémoire comme:

[22] https://github.com/DennisYurichev/base64scanner
[23] National Security Agency (Agence Nationale de la Sécurité)
[24] NDT: traduit en français par Laurent Viennot
[25] La traduction de la citation est extraite de ce livre.

adresse en mémoire	valeur de l'octet
+0	0x78
+1	0x56
+2	0x34
+3	0x12

Les CPUS little-endian comprennent les Intel x86.

2.8.3 Exemple

Prenons une système Linux MIPS big-endian déjà installé et prêt dans QEMU [26].

Et compilons cet exemple simple:

```
#include <stdio.h>

int main()
{
        int v;

        v=123;

        printf ("%02X %02X %02X %02X\n",
                *(char*)&v,
                *(((char*)&v)+1),
                *(((char*)&v)+2),
                *(((char*)&v)+3));
};
```

Après l'avoir lancé nous obtenons:

```
root@debian-mips :~# ./a.out
00 00 00 7B
```

C'est ça. 0x7B est 123 en décimal. En architecture little-endian, 7B est le premier octet (vous pouvez vérifier en x86 ou en x86-64, mais ici c'est le dernier, car l'octet le plus significatif vient en premier.

C'est pourquoi il y a des distributions Liux séparées pour MIPS (« mips » (big-endian) et « mipsel » (little-endian)). Il est impossible pour un binaire compilé pour une architecture de fonctionner sur un OS avec une architecture différente.

Il y a un exemple de MIPS big-endian dans ce livre: 1.24.4 on page 367.

2.8.4 Bi-endian

Les CPUs qui peuvent changer d'endianness sont les ARM, PowerPC, SPARC, MIPS, IA64[27], etc.

2.8.5 Convertir des données

L'instruction BSWAP peut être utilisée pour la conversion.

Les paquets de données des réseaux TCP/IP utilisent la convention bit-endian, c'est donc pourquoi un programme travaillant en architecture little-endian doit convertir les valeurs. Les fonctions `htonl()` et `htons()` sont utilisées en général.

En TCP/IP, big-endian est aussi appelé « network byte order », tandis que l'ordre des octets sur l'ordinateur « host byte order ». Le « host byte order » est little-endian sur les x86 Intel et les autres architectures little-endian, mais est big-endian sur les IBM POWER, donc `htonl()` et `htons()` ne modifient aucun octet sur cette dernière.

[26] Disponible au téléchargement ici: http://go.yurichev.com/17008
[27] Intel Architecture 64 (Itanium)

2.9 Mémoire

Il y a 3 grands types de mémoire:

- Mémoire globale AKA « allocation statique de mémoire ». Pas besoin de l'allouer explicitement, l'allocation est effectuée juste en déclarant des variables/tableaux globalement. Ce sont des variables globales, se trouvant dans le segment de données ou de constantes. Elles sont accessibles globalement (ce qui est considéré comme un anti-pattern). Ce n'est pas pratique pour les buffers/tableaux, car ils doivent avoir une taille fixée. Les débordements de tampons se produisant ici le sont en général en récrivant les variables ou les buffers se trouvant à côté d'eux en mémoire. Il y a un exemple dans ce livre: 1.9.3 on page 76.
- Stack (pile) AKA « allocation sur la pile ». L'allocation est effectuée simplement en déclarant des variables/ tableaux localement dans la fonction. Ce sont en général des variables locales de la fonction. Parfois ces variables locales sont aussi visibles depuis les fonctions appelées, si l'appelant passe un pointeur sur une variable à la fonction appelée qui va être exécutée). L'allocation et la dé-allocation sont très rapide, il suffit de décaler SP.

  Mais elles ne conviennent pas non plus pour les tampons/tableaux, car la taille du tampon doit être fixée, à moins qu'`alloca()` (1.7.2 on page 35) (ou un tableau de longueur variable) ne soit utilisé. Les débordements de tampons écrasent en général les structures de pile importantes: 1.20.2 on page 276.
- Heap (tas) AKA « allocation dynamique de mémoire ». L'allocation/dé-allocation est effectuée en appelant `malloc()/free()` ou `new/delete` en C++. Ceci est la méthode la plus pratique: la taille du bloc peut être définie lors de l'exécution.

  Il est possible de redimensionner (en utilisant `realloc()`), mais ça peut être long. Ceci est le moyen le plus lent d'allouer de la mémoire: L'allocation de mémoire doit gérer et mettre à jour toutes les structures de contrôle pendant l'allocation et la dé-allocation. Les débordements de tampons écrasent en général ces structures. L'allocation sur le tas est aussi la source des problème de fuite de mémoire: chaque bloc de mémoire doit être dé-alloué explicitement, mais on peut oublier de le faire, ou le faire de manière incorrecte.

  Un autre problème est l'« utilisation après la libération »—-utiliser un bloc de mémoire après que `free()` ait été appelé, ce qui est très dangereux.

  Exemple dans ce livre: 1.24.2 on page 350.

2.10 CPU

2.10.1 Prédicteurs de branchement

Certains des derniers compilateurs essayent d'éliminer les instructions de saut. Il y a des exemples dans ce livre: 1.14.1 on page 135, 1.14.3 on page 143, 1.22.5 on page 332.

C'est parce que le prédicteur de branchement n'est pas toujours parfait, donc les compilateurs essayent de faire sans les sauts conditionnels, si possible.

Les instructions conditionnelles en ARM (comme ADRcc) sont une manière, une autre est l'instruction x86 CMOVcc.

2.10.2 Dépendances des données

Les CPUs modernes sont capables d'exécuter des instructions simultanément (OOE[28]), mais pour ce faire, le résultat d'une instruction dans un groupe ne doit pas influencer l'exécution des autres. Par conséquent, le compilateur s'efforce d'utiliser des instructions avec le minimum d'influence sur l'état du CPU.

C'est pourquoi l'instruction LEA est si populaire, car elle ne modifie pas les flags du CPU, tandis que d'autres instructions arithmétiques le font.

[28] Out-of-Order Execution

2.11 Fonctions de hachage

Un exemple très simple est CRC32, un algorithme qui fournit des checksum plus « fort » à des fins de vérifications d'intégrité. Il est impossible de restaurer le texte d'origine depuis la valeur du hash, il a beaucoup moins d'informations: Mais CRC32 n'est pas cryptographiquement sûr: on sait comment modifier un texte afin que son hash CRC32 résultant soit celui que l'on veut. Les fonctions cryptographiques sont protégées contre cela.

MD5, SHA1, etc. sont de telles fonctions et elles sont largement utilisées pour hacher les mots de passe des utilisateurs afin de les stocker dans une base de données. En effet: la base de données d'un forum Internet ne doit pas contenir les mots de passe des utilisateurs (une base de données volée compromettrait tous les mots de passe des utilisateurs) mais seulement les hachages (donc un cracker ne pourrait pas révéler les mots de passe). En outre, un forum Internet n'a pas besoin de connaître votre mot de passe exactement, il a seulement besoin de vérifier si son hachage est le même que celui dans la base de données, et vous donne accès s'ils correspondent. Une des méthodes de cracking la plus simple est simplement d'essayer de hacher tous les mots de passe possible pour voir celui qui correspond à la valeur recherchée. D'autres méthodes sont beaucoup plus complexes.

2.11.1 Comment fonctionnent les fonctions à sens unique?

Une fonction à sens unique est une fonction qui est capable de transformer une valeur en une autre, tandis qu'il est impossible (ou très difficile) de l'inverser. Certaines personnes éprouvent des difficultés à comprendre comment ceci est possible. Voici une démonstration simple.

Nous avons un vecteur de 10 nombres dans l'intervalle 0..9, chacun est présent une seule fois, par exemple:

```
4 6 0 1 3 5 7 8 9 2
```

L'algorithme pour une fonction à sens unique la plus simple possible est:

- prendre le nombre à l'indice zéro (4 dans notre cas);
- prendre le nombre à l'indice 1 (6 dans notre cas);
- échanger les nombres aux positions 4 et 6.

Marquons les nombres aux positions 4 et 6:

```
4 6 0 1 3 5 7 8 9 2
        ^   ^
```

Échangeons-les et nous obtenons ce résultat:

```
4 6 0 1 7 5 3 8 9 2
```

En regardant le résultat, et même si nous connaissons l'algorithme, nous ne pouvons pas connaître l'état initial de façon certaine, car les deux premiers nombres pourraient être 0 et/ou 1, et pourraient donc participer à la procédure d'échange.

Ceci est un exemple extrêmement simplifié pour la démonstration. Les fonctions à sens unique réelles sont bien plus complexes.

Chapitre 3

Exemples un peu plus avancés

3.1 Double négation

Une façon répandue[1] de convertir des valeurs différentes de zéro en 1 (ou le booléen *true*) et la valeur zéro en 0 (ou le booléen *false*) est la déclaration *!!variable* :

```
int convert_to_bool(int a)
{
        return !!a;
};
```

GCC 5.4 x86 avec optimisation:

```
convert_to_bool :
        mov     edx, DWORD PTR [esp+4]
        xor     eax, eax
        test    edx, edx
        setne   al
        ret
```

XOR efface toujours la valeur de retour dans EAX, même si SETNE n'est pas déclenché. I.e., XOR met la valeur de retour par défaut à zéro.

Si la valeur en entrée n'est pas égale à zéro (le suffixe -NE dans l'instruction SET), 1 est mis dans AL, autrement AL n'est pas modifié.

Pourquoi est-ce que SETNE opère sur la partie 8-bit basse du registre EAX? Parce que ce qui compte c'est juste le dernier bit (0 or 1), puisque les autres bits sont mis à zéro par XOR.

Ainsi, ce code C/C++ peut être récrit comme ceci:

```
int convert_to_bool(int a)
{
        if (a!=0)
                return 1;
        else
                return 0;
};
```

...ou même:

```
int convert_to_bool(int a)
{
        if (a)
                return 1;
        else
                return 0;
};
```

Les compilateurs visant des CPUs n'ayant pas d'instructions similaires à SET, génèrent dans ce cas des instructions de branchement, etc.

[1] C'est sujet à controverse, car ça conduit à du code difficile à lire

3.2 Exemple strstr()

Revenons au fait que GCC peut parfois utiliser une partie d'une chaîne de caractères: 1.5.3 on page 18.

La fonction *strstr()* de la bibliothèque standard C/C++ est utilisée pour trouver une occurrence dans une chaîne. C'est ce que nous voulons faire:

```
#include <string.h>
#include <stdio.h>

int main()
{
        char *s="Hello, world!";
        char *w=strstr(s, "world");

        printf ("%p, [%s]\n", s, s);
        printf ("%p, [%s]\n", w, w);
};
```

La sortie est:

```
0x8048530, [Hello, world!]
0x8048537, [world!]
```

La différence entre l'adresse de la chaîne originale et l'adresse de la sous-chaîne que *strstr()* a renvoyé est 7. En effet, la chaîne « Hello, » a une longueur de 7 caractères.

La fonction `printf()` lors du second appel n'a aucune idée qu'il y a des autres caractères avant la chaîne passée et elle affiche des caractères depuis le milieu de la chaîne originale jusqu'à la fin (marquée par un octet à zéro).

3.3 Conversion de température

Un autre exemple très populaire dans les livres de programmation est un petit programme qui convertit une température de Fahrenheit vers Celsius ou inversement.

$$C = \frac{5 \cdot (F - 32)}{9}$$

Nous pouvons aussi ajouter une gestion des erreurs simples: 1) nous devons vérifier si l'utilisateur a entré un nombre correct; 2) nous devons tester si la température en Celsius n'est pas en dessous de -273 (qui est en dessous du zéro absolu, comme vu pendant les cours de physique à l'école)

La fonction `exit()` termine le programme instantanément, sans retourner à la fonction appelante.

3.3.1 Valeurs entières

```
#include <stdio.h>
#include <stdlib.h>

int main()
{
        int celsius, fahr;
        printf ("Enter temperature in Fahrenheit :\n");
        if (scanf ("%d", &fahr)!=1)
        {
                printf ("Error while parsing your input\n");
                exit(0);
        };

        celsius = 5 * (fahr-32) / 9;

        if (celsius<-273)
        {
```

```
                printf ("Error : incorrect temperature!\n");
                exit(0);
        };
        printf ("Celsius : %d\n", celsius);
};
```

MSVC 2012 x86 avec optimisation

Listing 3.1: MSVC 2012 x86 avec optimisation

```
$SG4228 DB      'Enter temperature in Fahrenheit :', 0aH, 00H
$SG4230 DB      '%d', 00H
$SG4231 DB      'Error while parsing your input', 0aH, 00H
$SG4233 DB      'Error : incorrect temperature!', 0aH, 00H
$SG4234 DB      'Celsius : %d', 0aH, 00H

_fahr$ = -4     ; taille = 4
_main   PROC
        push    ecx
        push    esi
        mov     esi, DWORD PTR __imp__printf
        push    OFFSET $SG4228          ; 'Enter temperature in Fahrenheit :'
        call    esi                     ; appeler printf()
        lea     eax, DWORD PTR _fahr$[esp+12]
        push    eax
        push    OFFSET $SG4230          ; '%d'
        call    DWORD PTR __imp__scanf
        add     esp, 12
        cmp     eax, 1
        je      SHORT $LN2@main
        push    OFFSET $SG4231          ; 'Error while parsing your input'
        call    esi                     ; appeler printf()
        add     esp, 4
        push    0
        call    DWORD PTR __imp__exit
$LN9@main :
$LN2@main :
        mov     eax, DWORD PTR _fahr$[esp+8]
        add     eax, -32        ; ffffffe0H
        lea     ecx, DWORD PTR [eax+eax*4]
        mov     eax, 954437177  ; 38e38e39H
        imul    ecx
        sar     edx, 1
        mov     eax, edx
        shr     eax, 31         ; 0000001fH
        add     eax, edx
        cmp     eax, -273       ; fffffeefH
        jge     SHORT $LN1@main
        push    OFFSET $SG4233          ; 'Error : incorrect temperature!'
        call    esi     ; appeler printf()
        add     esp, 4
        push    0
        call    DWORD PTR __imp__exit
$LN10@main :
$LN1@main :
        push    eax
        push    OFFSET $SG4234          ; 'Celsius : %d'
        call    esi     ; appeler printf()
        add     esp, 8
        ; renvoyer 0 - d'après le standard C99
        xor     eax, eax
        pop     esi
        pop     ecx
        ret     0
$LN8@main :
_main   ENDP
```

Ce que l'on peut en dire:

- L'adresse de `printf()` est d'abord chargée dans le registre ESI, donc les futurs appels à `printf()` seront faits juste par l'instruction `CALL ESI`. C'est une technique très populaire des compilateurs, possible si plusieurs appels consécutifs vers la même fonction sont présents dans le code, et/ou s'il y a un registre disponible qui peut être utilisé pour ça.
- Nous voyons l'instruction `ADD EAX, -32` à l'endroit où 32 doit être soustrait de la valeur. $EAX = EAX + (-32)$ est équivalent à $EAX = EAX - 32$ et curieusement, le compilateur a décidé d'utiliser ADD au lieu de SUB. Peut-être que ça en vaut la peine, difficile d'en être sûr.
- L'instruction LEA est utilisée quand la valeur est multipliée par 5: `lea ecx, DWORD PTR [eax+eax*4]`. Oui, $i + i * 4$ équivaut à $i * 5$ et LEA s'exécute plus rapidement que IMUL.
  D'ailleurs, la paire d'instructions `SHL EAX, 2 / ADD EAX, EAX` peut aussi être utilisée ici— certains compilateurs le font.
- La division par l'astuce de la multiplication (3.9 on page 501) est aussi utilisée ici.
- `main()` retourne 0 si nous n'avons pas `return 0` à la fin. Le standard C99 nous dit [*ISO/IEC 9899:TC3 (C C99 standard)*, (2007)5.1.2.2.3] que `main()` va retourner 0 dans le cas où la déclaration `return` est manquante. Cette règle fonctionne uniquement pour la fonction `main()`.

  Cependant, MSVC ne supporte pas officiellement C99, mais peut-être qu'il le supporte partiellement ?

MSVC 2012 x64 avec optimisation

Le code est quasiment le même, mais nous trouvons une instruction `INT 3` après chaque appel à `exit()`.

```
        xor     ecx, ecx
        call    QWORD PTR __imp_exit
        int     3
```

`INT 3` est un point d'arrêt du debugger.

C'est connu que `exit()` est l'une des fonctions qui ne retourne jamais [2], donc si elle le fait, quelque chose de vraiment étrange est arrivé et il est temps de lancer le debugger.

3.3.2 Valeurs à virgule flottante

```
#include <stdio.h>
#include <stdlib.h>

int main()
{
        double celsius, fahr;
        printf ("Enter temperature in Fahrenheit :\n");
        if (scanf ("%lf", &fahr)!=1)
        {
                printf ("Error while parsing your input\n");
                exit(0);
        };

        celsius = 5 * (fahr-32) / 9;

        if (celsius<-273)
        {
                printf ("Error : incorrect temperature!\n");
                exit(0);
        };
        printf ("Celsius : %lf\n", celsius);
};
```

MSVC 2010 x86 utilise des instructions FPU...

Listing 3.2: MSVC 2010 x86 avec optimisation

```
$SG4038 DB      'Enter temperature in Fahrenheit :', 0aH, 00H
$SG4040 DB      '%lf', 00H
```

[2] une autre connue est `longjmp()`

```
$SG4041 DB      'Error while parsing your input', 0aH, 00H
$SG4043 DB      'Error : incorrect temperature!', 0aH, 00H
$SG4044 DB      'Celsius : %lf', 0aH, 00H

__real@c071100000000000 DQ 0c071100000000000r   ; -273
__real@4022000000000000 DQ 04022000000000000r   ; 9
__real@4014000000000000 DQ 04014000000000000r   ; 5
__real@4040000000000000 DQ 04040000000000000r   ; 32

_fahr$ = -8     ; taille = 8
_main   PROC
        sub     esp, 8
        push    esi
        mov     esi, DWORD PTR __imp__printf
        push    OFFSET $SG4038          ; 'Enter temperature in Fahrenheit :'
        call    esi                     ; appeler printf()
        lea     eax, DWORD PTR _fahr$[esp+16]
        push    eax
        push    OFFSET $SG4040          ; '%lf'
        call    DWORD PTR __imp__scanf
        add     esp, 12
        cmp     eax, 1
        je      SHORT $LN2@main
        push    OFFSET $SG4041          ; 'Error while parsing your input'
        call    esi                     ; appeler printf()
        add     esp, 4
        push    0
        call    DWORD PTR __imp__exit
$LN2@main :
        fld     QWORD PTR _fahr$[esp+12]
        fsub    QWORD PTR __real@4040000000000000 ; 32
        fmul    QWORD PTR __real@4014000000000000 ; 5
        fdiv    QWORD PTR __real@4022000000000000 ; 9
        fld     QWORD PTR __real@c071100000000000 ; -273
        fcomp   ST(1)
        fnstsw  ax
        test    ah, 65  ; 00000041H
        jne     SHORT $LN1@main
        push    OFFSET $SG4043          ; 'Error : incorrect temperature!'
        fstp    ST(0)
        call    esi             ; appeler printf()
        add     esp, 4
        push    0
        call    DWORD PTR __imp__exit
$LN1@main :
        sub     esp, 8
        fstp    QWORD PTR [esp]
        push    OFFSET $SG4044          ; 'Celsius : %lf'
        call    esi
        add     esp, 12
        ; renvoyer 0 - d'après le standard C99
        xor     eax, eax
        pop     esi
        add     esp, 8
        ret     0
$LN10@main :
_main   ENDP
```

...mais MSVC 2012 utilise à la place des instructions SIMD :

Listing 3.3: MSVC 2010 x86 avec optimisation

```
$SG4228 DB      'Enter temperature in Fahrenheit :', 0aH, 00H
$SG4230 DB      '%lf', 00H
$SG4231 DB      'Error while parsing your input', 0aH, 00H
$SG4233 DB      'Error : incorrect temperature!', 0aH, 00H
$SG4234 DB      'Celsius : %lf', 0aH, 00H
__real@c071100000000000 DQ 0c071100000000000r   ; -273
__real@4040000000000000 DQ 04040000000000000r   ; 32
__real@4022000000000000 DQ 04022000000000000r   ; 9
__real@4014000000000000 DQ 04014000000000000r   ; 5
```

```
_fahr$ = -8     ; taile = 8
_main   PROC
        sub     esp, 8
        push    esi
        mov     esi, DWORD PTR __imp__printf
        push    OFFSET $SG4228          ; 'Enter temperature in Fahrenheit :'
        call    esi                     ; appeler printf()
        lea     eax, DWORD PTR _fahr$[esp+16]
        push    eax
        push    OFFSET $SG4230          ; '%lf'
        call    DWORD PTR __imp__scanf
        add     esp, 12
        cmp     eax, 1
        je      SHORT $LN2@main
        push    OFFSET $SG4231          ; 'Error while parsing your input'
        call    esi     ; appeler printf()
        add     esp, 4
        push    0
        call    DWORD PTR __imp__exit
$LN9@main :
$LN2@main :
        movsd   xmm1, QWORD PTR _fahr$[esp+12]
        subsd   xmm1, QWORD PTR __real@4040000000000000 ; 32
        movsd   xmm0, QWORD PTR __real@c071100000000000 ; -273
        mulsd   xmm1, QWORD PTR __real@4014000000000000 ; 5
        divsd   xmm1, QWORD PTR __real@4022000000000000 ; 9
        comisd  xmm0, xmm1
        jbe     SHORT $LN1@main
        push    OFFSET $SG4233          ; 'Error : incorrect temperature!'
        call    esi             ; appeler printf()
        add     esp, 4
        push    0
        call    DWORD PTR __imp__exit
$LN10@main :
$LN1@main :
        sub     esp, 8
        movsd   QWORD PTR [esp], xmm1
        push    OFFSET $SG4234          ; 'Celsius : %lf'
        call    esi                     ; appeler printf()
        add     esp, 12
        ; renvoyer 0 - d'après le standard C99
        xor     eax, eax
        pop     esi
        add     esp, 8
        ret     0
$LN8@main :
_main   ENDP
```

Bien sûr, les instructions SIMD sont disponibles dans le mode x86, incluant celles qui fonctionnent avec les nombres à virgule flottante.

C'est un peu plus simple de les utiliser pour les calculs, donc le nouveau compilateur de Microsoft les utilise.

Nous pouvons aussi voir que la valeur -273 est chargée dans le registre `XMM0` trop tôt. Et c'est OK, parce que le compilateur peut mettre des instructions dans un ordre différent de celui du code source.

3.4 Suite de Fibonacci

Un autre exemple très utilisé dans les livres de programmation est la fonction récursive qui génère les termes de la suite de Fibonacci[3]. Cette suite est très simple: chaque nombre consécutif est la somme des deux précédents. Les deux premiers termes sont 0 et 1 ou 1 et 1.

La suite commence comme ceci:

[3] http://go.yurichev.com/17332

$0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, 233, 377, 610, 987, 1597, 2584, 4181...$

3.4.1 Exemple #1

L'implémentation est simple. Ce programme génère la suite jusqu'à 21.

```
#include <stdio.h>

void fib (int a, int b, int limit)
{
        printf ("%d\n", a+b);
        if (a+b > limit)
                return;
        fib (b, a+b, limit);
};

int main()
{
        printf ("0\n1\n1\n");
        fib (1, 1, 20);
};
```

Listing 3.4: MSVC 2010 x86

```
_a$ = 8                 ; size = 4
_b$ = 12                ; size = 4
_limit$ = 16            ; size = 4
_fib    PROC
        push    ebp
        mov     ebp, esp
        mov     eax, DWORD PTR _a$[ebp]
        add     eax, DWORD PTR _b$[ebp]
        push    eax
        push    OFFSET $SG2643
        call    DWORD PTR __imp__printf
        add     esp, 8
        mov     ecx, DWORD PTR _a$[ebp]
        add     ecx, DWORD PTR _b$[ebp]
        cmp     ecx, DWORD PTR _limit$[ebp]
        jle     SHORT $LN1@fib
        jmp     SHORT $LN2@fib
$LN1@fib :
        mov     edx, DWORD PTR _limit$[ebp]
        push    edx
        mov     eax, DWORD PTR _a$[ebp]
        add     eax, DWORD PTR _b$[ebp]
        push    eax
        mov     ecx, DWORD PTR _b$[ebp]
        push    ecx
        call    _fib
        add     esp, 12
$LN2@fib :
        pop     ebp
        ret     0
_fib    ENDP

_main   PROC
        push    ebp
        mov     ebp, esp
        push    OFFSET $SG2647 ; "0\n1\n1\n"
        call    DWORD PTR __imp__printf
        add     esp, 4
        push    20
        push    1
        push    1
        call    _fib
        add     esp, 12
        xor     eax, eax
        pop     ebp
```

```
        ret     0
_main   ENDP
```

Nous allons illustrer les frames de pile avec ceci.

Chargeons cet exemple dans OllyDbg et traçons jusqu'au dernier appel de f() :

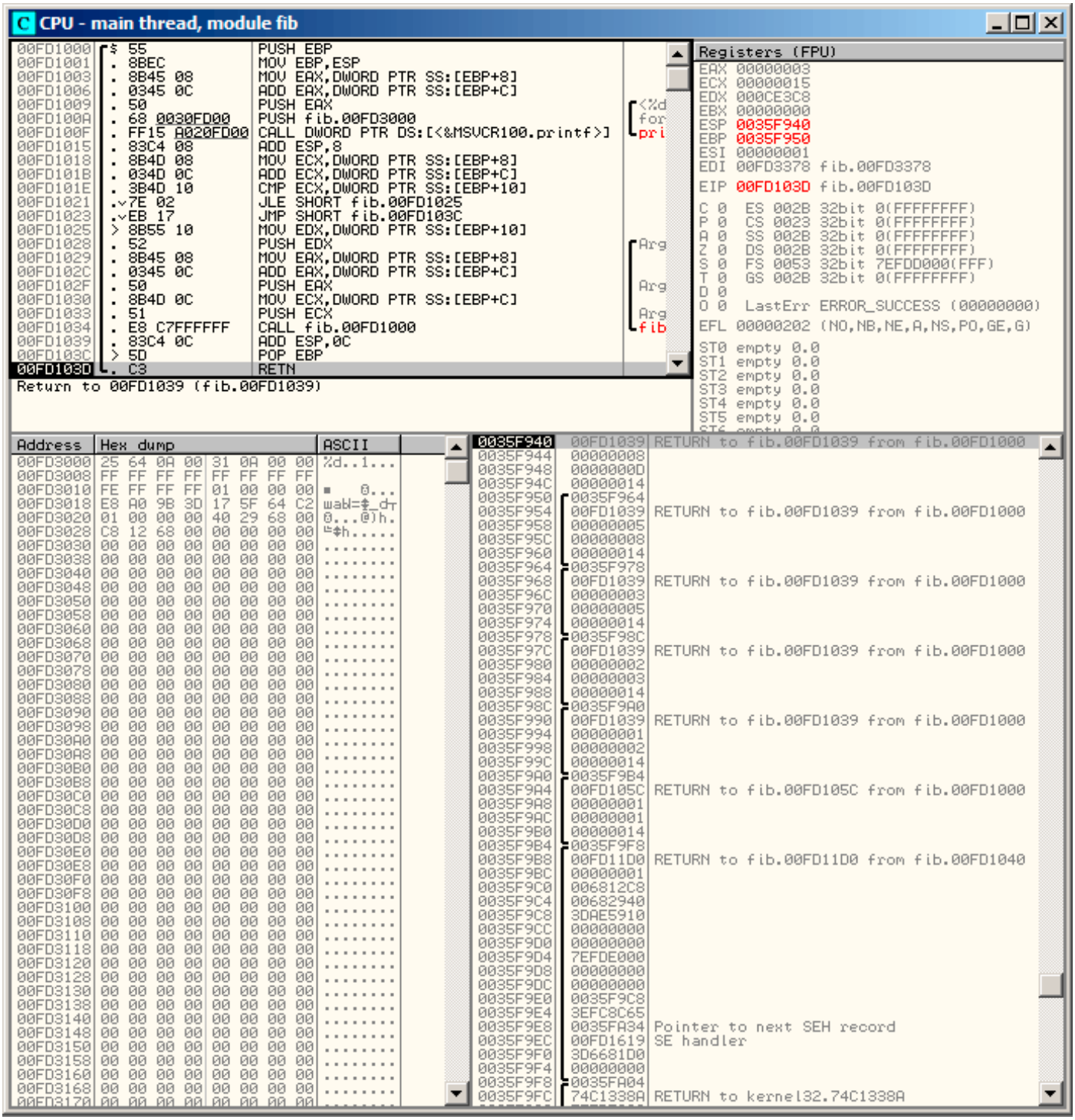


Fig. 3.1: OllyDbg : dernier appel de f()

Examinons plus attentivement la pile. Les commentaires ont été ajoutés par l'auteur de ce livre [4] :

```
0035F940   00FD1039  RETURN to fib.00FD1039 from fib.00FD1000
0035F944   00000008  1er  argument : a
0035F948   0000000D  2nd  argument b
0035F94C   00000014  3ème argument : limit
0035F950  /0035F964  registre EBP sauvé
0035F954  |00FD1039  RETURN to fib.00FD1039 from fib.00FD1000
0035F958  |00000005  1er  argument : a
0035F95C  |00000008  2nd  argument : b
0035F960  |00000014  3ème argument : limit
0035F964  ]0035F978  registre EBP sauvé
0035F968  |00FD1039  RETURN to fib.00FD1039 from fib.00FD1000
0035F96C  |00000003  1er  argument : a
0035F970  |00000005  2nd  argument : b
0035F974  |00000014  3ème argument : limit
0035F978  ]0035F98C  registre EBP sauvé
0035F97C  |00FD1039  RETURN to fib.00FD1039 from fib.00FD1000
0035F980  |00000002  1er  argument : a
0035F984  |00000003  2nd  argument : b
0035F988  |00000014  3ème argument : limit
0035F98C  ]0035F9A0  registre EBP sauvé
0035F990  |00FD1039  RETURN to fib.00FD1039 from fib.00FD1000
0035F994  |00000001  1er  argument : a
0035F998  |00000002  2nd  argument : b
0035F99C  |00000014  3ème argument : limit
0035F9A0  ]0035F9B4  registre EBP sauvé
0035F9A4  |00FD105C  RETURN to fib.00FD105C from fib.00FD1000
0035F9A8  |00000001  1er  argument : a            \
0035F9AC  |00000001  2nd  argument : b            | préparé dans main() pour f1()
0035F9B0  |00000014  3ème argument : limit        /
0035F9B4  ]0035F9F8  registre EBP sauvé
0035F9B8  |00FD11D0  RETURN to fib.00FD11D0 from fib.00FD1040
0035F9BC  |00000001  main() 1er  argument : argc  \
0035F9C0  |006812C8  main() 2nd  argument : argv  | préparé dans CRT pour main()
0035F9C4  |00682940  main() 3ème argument : envp  /
```

La fonction est récursive [5], donc la pile ressemble à un « sandwich ».

Nous voyons que l'argument *limit* est toujours le même (0x14 ou 20), mais que les arguments a et b sont différents pour chaque appel.

Il y a aussi ici le RA-s et la valeur sauvée de EBP. OllyDbg est capable de déterminer les frames basés sur EBP, donc il les dessine ces accolades. Les valeurs dans chaque accolade constituent la frame de pile, autrement dit, la zone de la pile qu'un appel de fonction utilise comme espace dédié.

Nous pouvons aussi dire que chaque appel de fonction ne doit pas accéder les éléments de la pile au delà des limites de son bloc (en excluant les arguments de la fonction), bien que cela soit techniquement possible.

C'est généralement vrai, à moins que la fonction n'ait des bugs.

Chaque valeur sauvée de EBP est l'adresse de la structure de pile locale précédente: c'est la raison pour laquelle certains débogueurs peuvent facilement diviser la pile en blocs et afficher chaque argument de la fonction.

Comme nous le voyons ici, chaque exécution de fonction prépare les arguments pour l'appel de fonction suivant.

À la fin, nous voyons les 3 arguments de `main()`. `argc` vaut 1 (oui, en effet, nous avons lancé le programme sans argument sur la ligne de commande).

Ceci peut conduire facilement à un débordement de pile: il suffit de supprimer (ou commenter) le test de la limite et ça va planter avec l'exception 0xC00000FD (stack overflow).

[4] À propos, il est possible de sélectionner plusieurs entrées dans OllyDbg et de les copier dans le presse-papier (Crtl-C). C'est ce qui a été fait par l'auteur pour cet exemple.

[5] i.e., elle s'appelle elle-même

3.4.2 Exemple #2

Ma fonction a quelques redondances, donc ajoutons une nouvelle variable locale *next* et remplaçons tout les « a+b » avec elle:

```
#include <stdio.h>

void fib (int a, int b, int limit)
{
        int next=a+b;
        printf ("%d\n", next);
        if (next > limit)
                return;
        fib (b, next, limit);
};

int main()
{
        printf ("0\n1\n1\n");
        fib (1, 1, 20);
};
```

C'est la sortie de MSVC sans optimisation, donc la variable *next* est allouée sur la pile locale:

Listing 3.5: MSVC 2010 x86

```
_next$ = -4     ; size = 4
_a$ = 8         ; size = 4
_b$ = 12        ; size = 4
_limit$ = 16    ; size = 4
_fib    PROC
        push    ebp
        mov     ebp, esp
        push    ecx
        mov     eax, DWORD PTR _a$[ebp]
        add     eax, DWORD PTR _b$[ebp]
        mov     DWORD PTR _next$[ebp], eax
        mov     ecx, DWORD PTR _next$[ebp]
        push    ecx
        push    OFFSET $SG2751 ; '%d'
        call    DWORD PTR __imp__printf
        add     esp, 8
        mov     edx, DWORD PTR _next$[ebp]
        cmp     edx, DWORD PTR _limit$[ebp]
        jle     SHORT $LN1@fib
        jmp     SHORT $LN2@fib
$LN1@fib :
        mov     eax, DWORD PTR _limit$[ebp]
        push    eax
        mov     ecx, DWORD PTR _next$[ebp]
        push    ecx
        mov     edx, DWORD PTR _b$[ebp]
        push    edx
        call    _fib
        add     esp, 12
$LN2@fib :
        mov     esp, ebp
        pop     ebp
        ret     0
_fib    ENDP

_main   PROC
        push    ebp
        mov     ebp, esp
        push    OFFSET $SG2753 ; "0\n1\n1\n"
        call    DWORD PTR __imp__printf
        add     esp, 4
        push    20
        push    1
        push    1
```

```
        call    _fib
        add     esp, 12
        xor     eax, eax
        pop     ebp
        ret     0
_main   ENDP
```

Chargeons-le à nouveau dans OllyDbg :

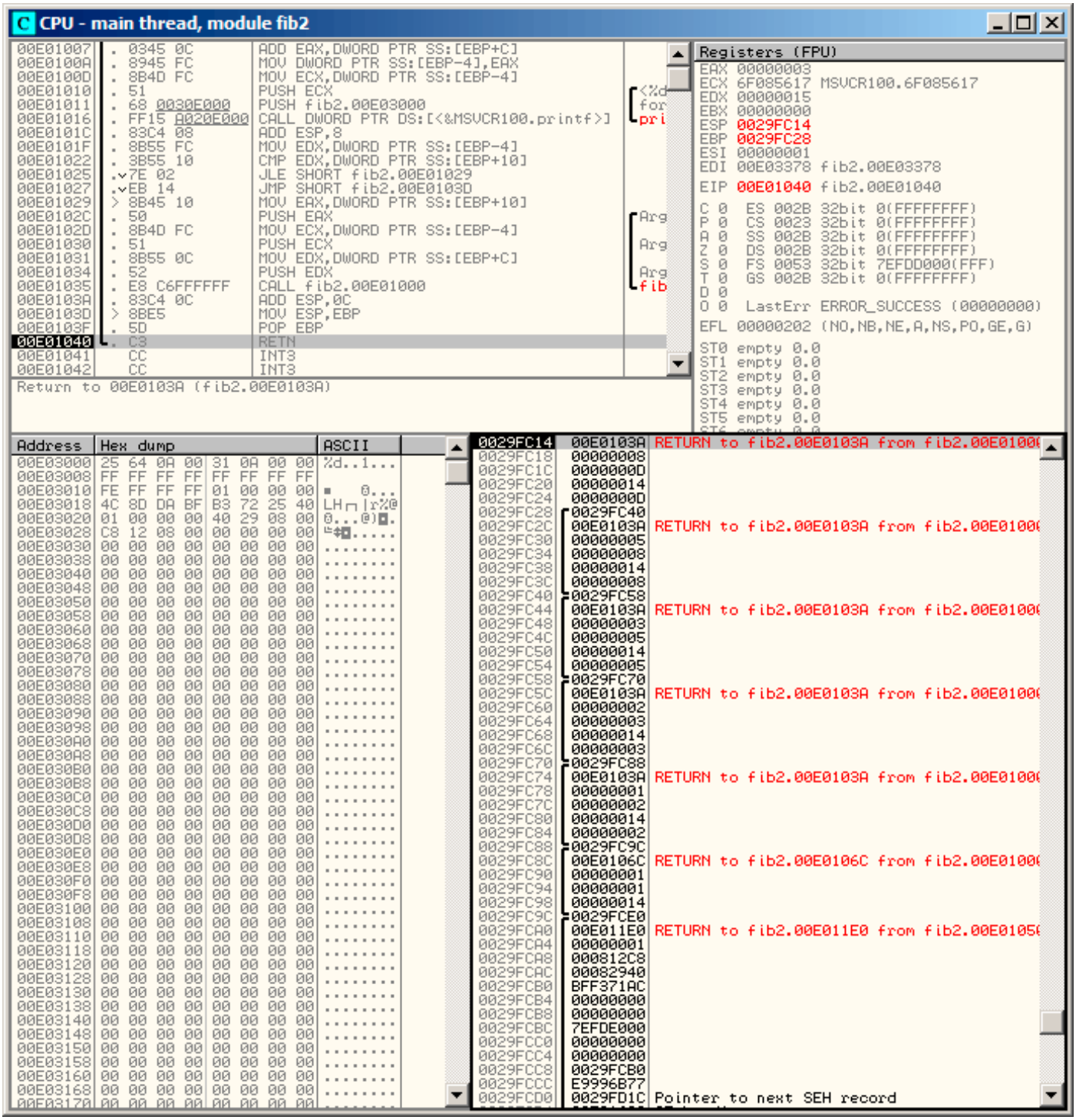


Fig. 3.2: OllyDbg : dernier appel de f()

Maintenant, la variable *next* est présente dans chaque frame.

Examinons plus attentivement la pile. L'auteur a de nouveau ajouté ses commentaires:

```
0029FC14   00E0103A  RETURN to fib2.00E0103A from fib2.00E01000
0029FC18   00000008  1er  argument : a
0029FC1C   0000000D  2nd  argument : b
0029FC20   00000014  3ème argument : limit
0029FC24   0000000D  variable "next"
0029FC28  /0029FC40  registre EBP sauvé
0029FC2C  |00E0103A  RETURN to fib2.00E0103A from fib2.00E01000
0029FC30  |00000005  1er  argument : a
0029FC34  |00000008  2nd  argument : b
0029FC38  |00000014  3ème argument : limit
0029FC3C  |00000008  "next" variable
0029FC40  ]0029FC58  registre EBP sauvé
0029FC44  |00E0103A  RETURN to fib2.00E0103A from fib2.00E01000
0029FC48  |00000003  1er  argument : a
0029FC4C  |00000005  2nd  argument : b
0029FC50  |00000014  3ème argument : limit
0029FC54  |00000005  variable "next"
0029FC58  ]0029FC70  registre EBP sauvé
0029FC5C  |00E0103A  RETURN to fib2.00E0103A from fib2.00E01000
0029FC60  |00000002  1er  argument : a
0029FC64  |00000003  2nd  argument : b
0029FC68  |00000014  3ème argument : limit
0029FC6C  |00000003  variable "next"
0029FC70  ]0029FC88  registre EBP sauvé
0029FC74  |00E0103A  RETURN to fib2.00E0103A from fib2.00E01000
0029FC78  |00000001  1er  argument : a            \
0029FC7C  |00000002  2nd  argument : b            | préparé dans f1() pour le prochain appel à ↙
    ↳ f1()
0029FC80  |00000014  3ème argument : limit        /
0029FC84  |00000002  variable "next"
0029FC88  ]0029FC9C  registre EBP sauvé
0029FC8C  |00E0106C  RETURN to fib2.00E0106C from fib2.00E01000
0029FC90  |00000001  1er  argument : a            \
0029FC94  |00000001  2nd  argument : b            | préparé dans main() pour f1()
0029FC98  |00000014  3ème argument : limit        /
0029FC9C  ]0029FCE0  registre EBP sauvé
0029FCA0  |00E011E0  RETURN to fib2.00E011E0 from fib2.00E01050
0029FCA4  |00000001  main() 1er  argument : argc  \
0029FCA8  |000812C8  main() 2nd  argument : argv  | préparé dans CRT pour main()
0029FCAC  |00082940  main() 3ème argument : envp  /
```

Voici ce que l'on voit: la valeur *next* est calculée dans chaque appel de la fonction, puis passée comme argument b au prochain appel.

3.4.3 Résumé

Les fonctions récursives sont esthétiquement jolies, mais techniquement elles peuvent dégrader les performances à cause de leur usage intensif de la pile. Quiconque qui écrit du code dont la perfomance est critique devrait probablement éviter la récursion.

Par exemple, j'ai écrit une fois une fonction pour chercher un nœud particulier dans un arbre binaire. Bien que la fonction récursive avait l'air élégante, il y avait du temps passé à chaque appel de fonction pour le prologue et l'épilogue, elle fonctionnait deux ou trois fois plus lentement que l'implémentation itérative (sans récursion).

À propos, c'est la raison pour laquelle certains compilateurs fonctionnels LP[6] (où la récursion est très utilisée) utilisent les appels terminaux. Nous parlons d'appel terminal lorsqu'une fonction a un seul appel à elle-même, situé à sa fin, comme:

Listing 3.6: Scheme, exemple copié/collé depuis Wikipédia

```
;; factorial  : nombre -> nombre
;; pour calculer le produit de tous les entiers
;; positifs inférieurs ou égaux à n.
(define (factorial n)
```

[6] LISP, Python, Lua, etc.

```
 (if (= n 1)
     1
         (* n (factorial (- n 1)))))
```

Les appels terminaux sont importants car le compilateur peut retravailler facilement ce code en un code itératif, pour supprimer la récursion.

3.5 Exemple de calcul de CRC32

C'est une technique très répandue de calcul de hachage basée sur une table CRC32[7].

```
/* By Bob Jenkins, (c) 2006, Public Domain */

#include <stdio.h>
#include <stddef.h>
#include <string.h>

typedef  unsigned long  ub4;
typedef  unsigned char  ub1;

static const ub4 crctab[256] = {
  0x00000000, 0x77073096, 0xee0e612c, 0x990951ba, 0x076dc419,
  0x706af48f, 0xe963a535, 0x9e6495a3, 0x0edb8832, 0x79dcb8a4,
  0xe0d5e91e, 0x97d2d988, 0x09b64c2b, 0x7eb17cbd, 0xe7b82d07,
  0x90bf1d91, 0x1db71064, 0x6ab020f2, 0xf3b97148, 0x84be41de,
  0x1adad47d, 0x6ddde4eb, 0xf4d4b551, 0x83d385c7, 0x136c9856,
  0x646ba8c0, 0xfd62f97a, 0x8a65c9ec, 0x14015c4f, 0x63066cd9,
  0xfa0f3d63, 0x8d080df5, 0x3b6e20c8, 0x4c69105e, 0xd56041e4,
  0xa2677172, 0x3c03e4d1, 0x4b04d447, 0xd20d85fd, 0xa50ab56b,
  0x35b5a8fa, 0x42b2986c, 0xdbbbc9d6, 0xacbcf940, 0x32d86ce3,
  0x45df5c75, 0xdcd60dcf, 0xabd13d59, 0x26d930ac, 0x51de003a,
  0xc8d75180, 0xbfd06116, 0x21b4f4b5, 0x56b3c423, 0xcfba9599,
  0xb8bda50f, 0x2802b89e, 0x5f058808, 0xc60cd9b2, 0xb10be924,
  0x2f6f7c87, 0x58684c11, 0xc1611dab, 0xb6662d3d, 0x76dc4190,
  0x01db7106, 0x98d220bc, 0xefd5102a, 0x71b18589, 0x06b6b51f,
  0x9fbfe4a5, 0xe8b8d433, 0x7807c9a2, 0x0f00f934, 0x9609a88e,
  0xe10e9818, 0x7f6a0dbb, 0x086d3d2d, 0x91646c97, 0xe6635c01,
  0x6b6b51f4, 0x1c6c6162, 0x856530d8, 0xf262004e, 0x6c0695ed,
  0x1b01a57b, 0x8208f4c1, 0xf50fc457, 0x65b0d9c6, 0x12b7e950,
  0x8bbeb8ea, 0xfcb9887c, 0x62dd1ddf, 0x15da2d49, 0x8cd37cf3,
  0xfbd44c65, 0x4db26158, 0x3ab551ce, 0xa3bc0074, 0xd4bb30e2,
  0x4adfa541, 0x3dd895d7, 0xa4d1c46d, 0xd3d6f4fb, 0x4369e96a,
  0x346ed9fc, 0xad678846, 0xda60b8d0, 0x44042d73, 0x33031de5,
  0xaa0a4c5f, 0xdd0d7cc9, 0x5005713c, 0x270241aa, 0xbe0b1010,
  0xc90c2086, 0x5768b525, 0x206f85b3, 0xb966d409, 0xce61e49f,
  0x5edef90e, 0x29d9c998, 0xb0d09822, 0xc7d7a8b4, 0x59b33d17,
  0x2eb40d81, 0xb7bd5c3b, 0xc0ba6cad, 0xedb88320, 0x9abfb3b6,
  0x03b6e20c, 0x74b1d29a, 0xead54739, 0x9dd277af, 0x04db2615,
  0x73dc1683, 0xe3630b12, 0x94643b84, 0x0d6d6a3e, 0x7a6a5aa8,
  0xe40ecf0b, 0x9309ff9d, 0x0a00ae27, 0x7d079eb1, 0xf00f9344,
  0x8708a3d2, 0x1e01f268, 0x6906c2fe, 0xf762575d, 0x806567cb,
  0x196c3671, 0x6e6b06e7, 0xfed41b76, 0x89d32be0, 0x10da7a5a,
  0x67dd4acc, 0xf9b9df6f, 0x8ebeeff9, 0x17b7be43, 0x60b08ed5,
  0xd6d6a3e8, 0xa1d1937e, 0x38d8c2c4, 0x4fdff252, 0xd1bb67f1,
  0xa6bc5767, 0x3fb506dd, 0x48b2364b, 0xd80d2bda, 0xaf0a1b4c,
  0x36034af6, 0x41047a60, 0xdf60efc3, 0xa867df55, 0x316e8eef,
  0x4669be79, 0xcb61b38c, 0xbc66831a, 0x256fd2a0, 0x5268e236,
  0xcc0c7795, 0xbb0b4703, 0x220216b9, 0x5505262f, 0xc5ba3bbe,
  0xb2bd0b28, 0x2bb45a92, 0x5cb36a04, 0xc2d7ffa7, 0xb5d0cf31,
  0x2cd99e8b, 0x5bdeae1d, 0x9b64c2b0, 0xec63f226, 0x756aa39c,
  0x026d930a, 0x9c0906a9, 0xeb0e363f, 0x72076785, 0x05005713,
  0x95bf4a82, 0xe2b87a14, 0x7bb12bae, 0x0cb61b38, 0x92d28e9b,
  0xe5d5be0d, 0x7cdcefb7, 0x0bdbdf21, 0x86d3d2d4, 0xf1d4e242,
  0x68ddb3f8, 0x1fda836e, 0x81be16cd, 0xf6b9265b, 0x6fb077e1,
  0x18b74777, 0x88085ae6, 0xff0f6a70, 0x66063bca, 0x11010b5c,
```

[7] Le code source provient d'ici: http://go.yurichev.com/17327

```
  0x8f659eff, 0xf862ae69, 0x616bffd3, 0x166ccf45, 0xa00ae278,
  0xd70dd2ee, 0x4e048354, 0x3903b3c2, 0xa7672661, 0xd06016f7,
  0x4969474d, 0x3e6e77db, 0xaed16a4a, 0xd9d65adc, 0x40df0b66,
  0x37d83bf0, 0xa9bcae53, 0xdebb9ec5, 0x47b2cf7f, 0x30b5ffe9,
  0xbdbdf21c, 0xcabac28a, 0x53b39330, 0x24b4a3a6, 0xbad03605,
  0xcdd70693, 0x54de5729, 0x23d967bf, 0xb3667a2e, 0xc4614ab8,
  0x5d681b02, 0x2a6f2b94, 0xb40bbe37, 0xc30c8ea1, 0x5a05df1b,
  0x2d02ef8d
};

/* how to derive the values in crctab[] from polynomial 0xedb88320 */
void build_table()
{
  ub4 i, j;
  for (i=0; i<256; ++i) {
    j = i;
    j = (j>>1) ^ ((j&1) ? 0xedb88320  : 0);
    j = (j>>1) ^ ((j&1) ? 0xedb88320  : 0);
    j = (j>>1) ^ ((j&1) ? 0xedb88320  : 0);
    j = (j>>1) ^ ((j&1) ? 0xedb88320  : 0);
    j = (j>>1) ^ ((j&1) ? 0xedb88320  : 0);
    j = (j>>1) ^ ((j&1) ? 0xedb88320  : 0);
    j = (j>>1) ^ ((j&1) ? 0xedb88320  : 0);
    j = (j>>1) ^ ((j&1) ? 0xedb88320  : 0);
    printf("0x%.8lx, ", j);
    if (i%6 == 5) printf("\n");
  }
}


/* the hash function */
ub4 crc(const void *key, ub4 len, ub4 hash)
{
  ub4  i;
  const ub1 *k = key;
  for (hash=len, i=0; i<len; ++i)
    hash = (hash >> 8) ^ crctab[(hash & 0xff) ^ k[i]];
  return hash;
}

/* To use, try "gcc -O crc.c -o crc; crc < crc.c" */
int main()
{
  char s[1000];
  while (gets(s)) printf("%.8lx\n", crc(s, strlen(s), 0));
  return 0;
}
```

Nous sommes seulement intéressés par la fonction `crc()`. À propos, faîtes attention aux deux déclarations d'initialisation dans la boucle `for()` : `hash=len, i=0`. Le standard C/C++ permet ceci, bien sûr. Le code généré contiendra deux opérations dans la partie d'initialisation de la boucle, au lieu d'une.

Compilons-le dans MSVC avec l'optimisation (/0x). Dans un soucis de concision, seule la fonction `crc()` est listée ici, avec mes commentaires.

```
_key$ = 8                ; size = 4
_len$ = 12               ; size = 4
_hash$ = 16              ; size = 4
_crc    PROC
    mov    edx, DWORD PTR _len$[esp-4]
    xor    ecx, ecx  ; i est stocké dans ECX
    mov    eax, edx
    test   edx, edx
    jbe    SHORT $LN1@crc
    push   ebx
    push   esi
    mov    esi, DWORD PTR _key$[esp+4]  ; ESI = key
    push   edi
$LL3@crc :
```

```
; fonctionne avec des octets en utilisant seulement des registres 32-bit.
; l'octet à l'adresse key+i est stocké dans EDI

    movzx  edi, BYTE PTR [ecx+esi]
    mov    ebx, eax ; EBX = (hash = len)
    and    ebx, 255 ; EBX = hash & 0xff

; XOR EDI, EBX (EDI=EDI^EBX) - cette opération utilise tous les 32 bits de chaque registre
; mais les autres bits (8-31) sont toujours mis à 0, donc c'est OK'
; ils sont mis à 0 car, comme pour EDI, cela a été fait avec l'instruction MOVZX ci-dessus
; les bits hauts de EBX sont mis à 0 par l'instruction AND EBX, 255 ci-dessus (255 = 0xff)

    xor    edi, ebx

; EAX=EAX>>8; bits 24-31 pris de nul part seront mis à 0
    shr    eax, 8

; EAX=EAX^crctab[EDI*4] - choisir le EDI-ème élément de la table crctab[]
    xor    eax, DWORD PTR _crctab[edi*4]
    inc    ecx            ; i++
    cmp    ecx, edx       ; i<len ?
    jb     SHORT $LL3@crc ; oui
    pop    edi
    pop    esi
    pop    ebx
$LN1@crc :
    ret    0
_crc    ENDP
```

Essayons la même chose dans GCC 4.4.1 avec l'option -03 :

```
                public crc
crc             proc near

key             = dword ptr  8
hash            = dword ptr  0Ch

                push    ebp
                xor     edx, edx
                mov     ebp, esp
                push    esi
                mov     esi, [ebp+key]
                push    ebx
                mov     ebx, [ebp+hash]
                test    ebx, ebx
                mov     eax, ebx
                jz      short loc_80484D3
                nop                     ; remplissage
                lea     esi, [esi+0]    ; remplissage; fonctionne comme NOP
                                        ; (ESI ne change pas ici)

loc_80484B8 :
                mov     ecx, eax        ; sauve l'état pérécédent du hash dans ECX
                xor     al, [esi+edx]   ; AL=*(key+i)
                add     edx, 1          ; i++
                shr     ecx, 8          ; ECX=hash>>8
                movzx   eax, al         ; EAX=*(key+i)
                mov     eax, dword ptr ds :crctab[eax*4] ; EAX=crctab[EAX]
                xor     eax, ecx        ; hash=EAX^ECX
                cmp     ebx, edx
                ja      short loc_80484B8

loc_80484D3 :
                pop     ebx
                pop     esi
                pop     ebp
                retn
crc             endp
\
```

GCC a aligné le début de la boucle sur une limite de 8-octet en ajoutant `NOP` et `lea esi, [esi+0]` (qui est aussi une opération sans effet). Vous pouvez en lire plus à ce sujet dans la section npad (**??** on page ??).

3.6 Exemple de calcul d'adresse réseau

Comme nous le savons, une adresse (IPv4) consiste en quatre nombres dans l'intervalle $0 \ldots 255$, i.e., quatre octets.

Quatre octets peuvent être stockés facilement dans une variable 32-bit, donc une adresse IPv4 d'hôte, de masque réseau ou d'adresse de réseau peuvent toutes être un entier 32-bit.

Du point de vue de l'utilisateur, le masque réseau est défini par quatre nombres et est formaté comme 255.255.255.0, mais les ingénieurs réseaux (sysadmins) utilisent une notation plus compacte comme « /8 », « /16 », etc.

Cette notation défini simplement le nombre de bits qu'a le masque, en commençant par le MSB.

Masque	Hôte	Utilisable	Masque de réseau	Masque hexadécimal	
/30	4	2	255.255.255.252	0xfffffffc	
/29	8	6	255.255.255.248	0xfffffff8	
/28	16	14	255.255.255.240	0xfffffff0	
/27	32	30	255.255.255.224	0xffffffe0	
/26	64	62	255.255.255.192	0xffffffc0	
/24	256	254	255.255.255.0	0xffffff00	réseau de classe C
/23	512	510	255.255.254.0	0xfffffe00	
/22	1024	1022	255.255.252.0	0xfffffc00	
/21	2048	2046	255.255.248.0	0xfffff800	
/20	4096	4094	255.255.240.0	0xfffff000	
/19	8192	8190	255.255.224.0	0xffffe000	
/18	16384	16382	255.255.192.0	0xffffc000	
/17	32768	32766	255.255.128.0	0xffff8000	
/16	65536	65534	255.255.0.0	0xffff0000	réseau de classe B
/8	16777216	16777214	255.0.0.0	0xff000000	réseau de classe A

Voici un petit exemple, qui calcule l'adresse du réseau en appliquant le masque réseau à l'adresse de l'hôte.

```
#include <stdio.h>
#include <stdint.h>

uint32_t form_IP (uint8_t ip1, uint8_t ip2, uint8_t ip3, uint8_t ip4)
{
        return (ip1<<24) | (ip2<<16) | (ip3<<8) | ip4;
};

void print_as_IP (uint32_t a)
{
        printf ("%d.%d.%d.%d\n",
                (a>>24)&0xFF,
                (a>>16)&0xFF,
                (a>>8)&0xFF,
                (a)&0xFF);
};

// bit=31..0
uint32_t set_bit (uint32_t input, int bit)
{
        return input=input|(1<<bit);
};

uint32_t form_netmask (uint8_t netmask_bits)
{
        uint32_t netmask=0;
        uint8_t i;

        for (i=0; i<netmask_bits; i++)
                netmask=set_bit(netmask, 31-i);
```

```
        return netmask;
};

void calc_network_address (uint8_t ip1, uint8_t ip2, uint8_t ip3, uint8_t ip4, uint8_t ↙
    ↳ netmask_bits)
{
        uint32_t netmask=form_netmask(netmask_bits);
        uint32_t ip=form_IP(ip1, ip2, ip3, ip4);
        uint32_t netw_adr;

        printf ("netmask=");
        print_as_IP (netmask);

        netw_adr=ip&netmask;

        printf ("network address=");
        print_as_IP (netw_adr);
};

int main()
{
        calc_network_address (10, 1, 2, 4, 24);    // 10.1.2.4, /24
        calc_network_address (10, 1, 2, 4, 8);     // 10.1.2.4, /8
        calc_network_address (10, 1, 2, 4, 25);    // 10.1.2.4, /25
        calc_network_address (10, 1, 2, 64, 26);   // 10.1.2.4, /26
};
```

3.6.1 calc_network_address()

La fonction `calc_network_address()` est la plus simple: elle effectue simplement un AND entre l'adresse de l'hôte et le masque de réseau, dont le résultat est l'adresse du réseau.

Listing 3.7: MSVC 2012 avec optimisation /Ob0

```
1  _ip1$ = 8               ; size = 1
2  _ip2$ = 12              ; size = 1
3  _ip3$ = 16              ; size = 1
4  _ip4$ = 20              ; size = 1
5  _netmask_bits$ = 24     ; size = 1
6  _calc_network_address PROC
7          push    edi
8          push    DWORD PTR _netmask_bits$[esp]
9          call    _form_netmask
10         push    OFFSET $SG3045 ; 'netmask='
11         mov     edi, eax
12         call    DWORD PTR __imp__printf
13         push    edi
14         call    _print_as_IP
15         push    OFFSET $SG3046 ; 'network address='
16         call    DWORD PTR __imp__printf
17         push    DWORD PTR _ip4$[esp+16]
18         push    DWORD PTR _ip3$[esp+20]
19         push    DWORD PTR _ip2$[esp+24]
20         push    DWORD PTR _ip1$[esp+28]
21         call    _form_IP
22         and     eax, edi        ; network address = host address & netmask
23         push    eax
24         call    _print_as_IP
25         add     esp, 36
26         pop     edi
27         ret     0
28 _calc_network_address ENDP
```

À la ligne 22, nous voyons le plus important `AND`—ici l'adresse du réseau est calculée.

3.6.2 form_IP()

La fonction `form_IP()` met juste les 4 octets dans une valeur 32-bit.

Voici comment cela est fait habituellement:

- Allouer une variable pour la valeur de retour. La mettre à 0.
- Prendre le 4ème octet (de poids le plus faible), appliquer l'opération OR à cet octet et renvoyer la valeur.
- Prendre le troisième octet, le décaler à gauche de 8 bits. Vous obtenez une valeur comme `0x0000bb00` où bb est votre troisième octet. Appliquer l'opération OR à la valeur résultante. La valeur de retour contenait `0x000000aa` jusqu'à présent, donc effectuer un OU logique des valeurs produira une valeur comme `0x0000bbaa`.
- Prendre le second octet, le décaler à gauche de 16 bits. Vous obtenez une valeur comme `0x00cc0000` où cc est votre deuxième octet. Appliquer l'opération OR à la valeur résultante. La valeur de retour contenait `0x0000bbaa` jusqu'à présent, donc effectuer un OU logique des valeurs produira une valeur comme `0x00ccbbaa`.
- Prendre le premier octet, le décaler à gauche de 24 bits. Vous obtenez une valeur comme `0xdd000000` où dd est votre premier octet. Appliquer l'opération OR à la valeur résultante. La valeur de retour contenait `0x00ccbbaa` jusqu'à présent, donc effectuer un OU logique des valeurs produira une valeur comme `0xddccbbaa`.

Voici comment c'est fait par MSVC 2012 sans optimisation:

Listing 3.8: MSVC 2012 sans optimisation

```
; désigner ip1 comme "dd", ip2 comme "cc", ip3 comme "bb", ip4 comme "aa".
_ip1$ = 8       ; taille = 1
_ip2$ = 12      ; taille = 1
_ip3$ = 16      ; taille = 1
_ip4$ = 20      ; taille = 1
_form_IP PROC
        push    ebp
        mov     ebp, esp
        movzx   eax, BYTE PTR _ip1$[ebp]
        ; EAX=000000dd
        shl     eax, 24
        ; EAX=dd000000
        movzx   ecx, BYTE PTR _ip2$[ebp]
        ; ECX=000000cc
        shl     ecx, 16
        ; ECX=00cc0000
        or      eax, ecx
        ; EAX=ddcc0000
        movzx   edx, BYTE PTR _ip3$[ebp]
        ; EDX=000000bb
        shl     edx, 8
        ; EDX=0000bb00
        or      eax, edx
        ; EAX=ddccbb00
        movzx   ecx, BYTE PTR _ip4$[ebp]
        ; ECX=000000aa
        or      eax, ecx
        ; EAX=ddccbbaa
        pop     ebp
        ret     0
_form_IP ENDP
```

Certes, l'ordre est différent, mais, bien sûr, l'ordre des opérations n'a pas d'importance.

MSVC 2012 avec optimisation produit en fait la même chose, mais d'une façon différente:

Listing 3.9: MSVC 2012 avec optimisation /Ob0

```
; désigner ip1 comme "dd", ip2 comme "cc", ip3 comme "bb", ip4 comme "aa".
_ip1$ = 8       ; taille = 1
_ip2$ = 12      ; taille = 1
_ip3$ = 16      ; taille = 1
_ip4$ = 20      ; taille = 1
```

```
_form_IP PROC
        movzx   eax, BYTE PTR _ip1$[esp-4]
        ; EAX=000000dd
        movzx   ecx, BYTE PTR _ip2$[esp-4]
        ; ECX=000000cc
        shl     eax, 8
        ; EAX=0000dd00
        or      eax, ecx
        ; EAX=0000ddcc
        movzx   ecx, BYTE PTR _ip3$[esp-4]
        ; ECX=000000bb
        shl     eax, 8
        ; EAX=00ddcc00
        or      eax, ecx
        ; EAX=00ddccbb
        movzx   ecx, BYTE PTR _ip4$[esp-4]
        ; ECX=000000aa
        shl     eax, 8
        ; EAX=ddccbb00
        or      eax, ecx
        ; EAX=ddccbbaa
        ret     0
_form_IP ENDP
```

Nous pourrions dire que chaque octet est écrit dans les 8 bits inférieurs de la valeur de retour, et qu'elle est ensuite décalée à gauche d'un octet à chaque étape.

Répéter 4 fois pour chaque octet en entrée.

C'est tout! Malheureusement, il n'y sans doute pas d'autre moyen de le faite.

Il n'y a pas de CPUs ou d'ISAs répandues qui possède une instruction pour composer une valeur à partir de bits ou d'octets.

C'est d'habitude fait par décalage de bit et OU logique.

3.6.3 print_as_IP()

La fonction `print_as_IP()` effectue l'inverse: séparer une valeur 32-bit en 4 octets.

Le découpage fonctionne un peu plus simplement: il suffit de décaler la valeur en entrée de 24, 16, 8 ou 0 bits, prendre les 8 bits d'indice 0 à 7 (octet de poids faible), et c'est fait:

Listing 3.10: MSVC 2012 sans optimisation

```
_a$ = 8                 ; size = 4
_print_as_IP PROC
        push    ebp
        mov     ebp, esp
        mov     eax, DWORD PTR _a$[ebp]
        ; EAX=ddccbbaa
        and     eax, 255
        ; EAX=000000aa
        push    eax
        mov     ecx, DWORD PTR _a$[ebp]
        ; ECX=ddccbbaa
        shr     ecx, 8
        ; ECX=00ddccbb
        and     ecx, 255
        ; ECX=000000bb
        push    ecx
        mov     edx, DWORD PTR _a$[ebp]
        ; EDX=ddccbbaa
        shr     edx, 16
        ; EDX=0000ddcc
        and     edx, 255
        ; EDX=000000cc
        push    edx
        mov     eax, DWORD PTR _a$[ebp]
        ; EAX=ddccbbaa
```

```
        shr     eax, 24
        ; EAX=000000dd
        and     eax, 255 ; sans doute une instruction redondante
        ; EAX=000000dd
        push    eax
        push    OFFSET $SG2973 ; '%d.%d.%d.%d'
        call    DWORD PTR __imp__printf
        add     esp, 20
        pop     ebp
        ret     0
_print_as_IP ENDP
```

MSVC 2012 avec optimisation fait presque la même chose, mais sans recharger inutilement la valeur en entrée:

Listing 3.11: MSVC 2012 avec optimisation /Ob0

```
_a$ = 8         ; size = 4
_print_as_IP PROC
        mov     ecx, DWORD PTR _a$[esp-4]
        ; ECX=ddccbbaa
        movzx   eax, cl
        ; EAX=000000aa
        push    eax
        mov     eax, ecx
        ; EAX=ddccbbaa
        shr     eax, 8
        ; EAX=00ddccbb
        and     eax, 255
        ; EAX=000000bb
        push    eax
        mov     eax, ecx
        ; EAX=ddccbbaa
        shr     eax, 16
        ; EAX=0000ddcc
        and     eax, 255
        ; EAX=000000cc
        push    eax
        ; ECX=ddccbbaa
        shr     ecx, 24
        ; ECX=000000dd
        push    ecx
        push    OFFSET $SG3020 ; '%d.%d.%d.%d'
        call    DWORD PTR __imp__printf
        add     esp, 20
        ret     0
_print_as_IP ENDP
```

3.6.4 form_netmask() et set_bit()

La fonction `form_netmask()` produit un masque de réseau à partir de la notation CIDR[8]. Bien sûr, il serait plus efficace d'utiliser une sorte de table pré-calculée, mais nous utilisons cette façon de faire intentionnellement, afin d'illustrer les décalages de bit.

Nous allons aussi écrire une fonction séparées `set_bit()`. Ce n'est pas une très bonne idée de créer un fonction pour une telle opération primitive, mais cela facilite la compréhension du fonctionnement.

Listing 3.12: MSVC 2012 avec optimisation /Ob0

```
_input$ = 8             ; size = 4
_bit$ = 12              ; size = 4
_set_bit PROC
        mov     ecx, DWORD PTR _bit$[esp-4]
        mov     eax, 1
        shl     eax, cl
        or      eax, DWORD PTR _input$[esp-4]
        ret     0
```

[8]Classless Inter-Domain Routing

```
_set_bit ENDP

_netmask_bits$ = 8      ; size = 1
_form_netmask PROC
        push    ebx
        push    esi
        movzx   esi, BYTE PTR _netmask_bits$[esp+4]
        xor     ecx, ecx
        xor     bl, bl
        test    esi, esi
        jle     SHORT $LN9@form_netma
        xor     edx, edx
$LL3@form_netma :
        mov     eax, 31
        sub     eax, edx
        push    eax
        push    ecx
        call    _set_bit
        inc     bl
        movzx   edx, bl
        add     esp, 8
        mov     ecx, eax
        cmp     edx, esi
        jl      SHORT $LL3@form_netma
$LN9@form_netma :
        pop     esi
        mov     eax, ecx
        pop     ebx
        ret     0
_form_netmask ENDP
```

`set_bit()` est primitive: elle décale juste 1 à gauche du nombre de bits dont nous avons besoin et puis effectue un OU logique avec la valeur « input ». `form_netmask()` a une boucle: elle met autant de bits (en partant du MSB) que demandé dans l'argument `netmask_bits`.

3.6.5 Résumé

C'est tout! Nous le lançons et obtenons:

```
netmask=255.255.255.0
network address=10.1.2.0
netmask=255.0.0.0
network address=10.0.0.0
netmask=255.255.255.128
network address=10.1.2.0
netmask=255.255.255.192
network address=10.1.2.64
```

3.7 Boucles: quelques itérateurs

Dans la plupart des cas, les boucles ont un seul itérateur, mais elles peuvent en avoir plusieurs dans le code résultant.

Voici un exemple très simple:

```
#include <stdio.h>

void f(int *a1, int *a2, size_t cnt)
{
        size_t i;

        // copier d'un tableau a l'autre selon un schema bizarre
        for (i=0; i<cnt; i++)
                a1[i*3]=a2[i*7];
};
```

Il y a deux multiplications à chaque itération et ce sont des opérations coûteuses. Est-ce que ça peut être optimisé d'une certaine façon?

Oui, si l'on remarque que les deux indices de tableau prennent des valeurs qui peuvent être facilement calculées sans multiplication.

3.7.1 Trois itérateurs

Listing 3.13: MSVC 2013 x64 avec optimisation

```
f       PROC
; RCX=a1
; RDX=a2
; R8=cnt
        test    r8, r8          ; cnt==0? sortir si oui
        je      SHORT $LN1@f
        npad    11
$LL3@f :
        mov     eax, DWORD PTR [rdx]
        lea     rcx, QWORD PTR [rcx+12]
        lea     rdx, QWORD PTR [rdx+28]
        mov     DWORD PTR [rcx-12], eax
        dec     r8
        jne     SHORT $LL3@f
$LN1@f :
        ret     0
f       ENDP
```

Maintenant il y a 3 itérateurs: la variable *cnt* et deux indices, qui sont incrémentés par 12 et 28 à chaque itération. Nous pouvons récrire ce code en C/C++ :

```
#include <stdio.h>

void f(int *a1, int *a2, size_t cnt)
{
        size_t i;
        size_t idx1=0; idx2=0;

        // copier d'un tableau a l'autre selon un schema bizarre
        for (i=0; i<cnt; i++)
        {
                a1[idx1]=a2[idx2];
                idx1+=3;
                idx2+=7;
        };
};
```

Donc, au prix de la mise à jour de 3 itérateurs à chaque itération au lieu d'un, nous pouvons supprimer deux opérations de multiplication.

3.7.2 Deux itérateurs

GCC 4.9 fait encore mieux, en ne laissant que 2 itérateurs:

Listing 3.14: GCC 4.9 x64 avec optimisation

```
; RDI=a1
; RSI=a2
; RDX=cnt
f :
        test    rdx, rdx  ; cnt==0? sortir si oui
        je      .L1
; calculer l'adresse du dernier élément dans "a2" et la laisser dans RDX
        lea     rax, [0+rdx*4]
; RAX=RDX*4=cnt*4
        sal     rdx, 5
; RDX=RDX<<5=cnt*32
```

```
        sub     rdx, rax
; RDX=RDX-RAX=cnt*32-cnt*4=cnt*28
        add     rdx, rsi
; RDX=RDX+RSI=a2+cnt*28
.L3 :
        mov     eax, DWORD PTR [rsi]
        add     rsi, 28
        add     rdi, 12
        mov     DWORD PTR [rdi-12], eax
        cmp     rsi, rdx
        jne     .L3
.L1 :
        rep ret
```

Il n'y a plus de variable *counter* : GCC en a conclu qu'elle n'étais pas nécessaire.

Le dernier élément du tableau *a2* est calculé avant le début de la boucle (ce qui est facile: $cnt * 7$) et c'est ainsi que la boucle est arrêtée: itérer jusqu'à ce que le second index atteignent cette valeur pré-calculée.

Vous trouverez plus d'informations sur la multiplication en utilisant des décalages/additions/soustractions ici: 1.18.1 on page 214.

Ce code peut être récrit en C/C++ comme ceci:

```
#include <stdio.h>

void f(int *a1, int *a2, size_t cnt)
{
        size_t idx1=0; idx2=0;
        size_t last_idx2=cnt*7;

        // copier d'un tableau a l'autre selon un schema bizarre
        for (;;)
        {
                a1[idx1]=a2[idx2];
                idx1+=3;
                idx2+=7;
                if (idx2==last_idx2)
                        break;
        };
};
```

GCC (Linaro) 4.9 pour ARM64 fait la même chose, mais il pré-calcule le dernier index de *a1* au lieu de *a2*, ce qui a bien sûr le même effet:

Listing 3.15: GCC (Linaro) 4.9 ARM64 avec optimisation

```
; X0=a1
; X1=a2
; X2=cnt
f :
        cbz     x2, .L1             ; cnt==0? sortir si oui
; calculer le dernier élément du tableau "a1"
        add     x2, x2, x2, lsl 1
; X2=X2+X2<<1=X2+X2*2=X2*3
        mov     x3, 0
        lsl     x2, x2, 2
; X2=X2<<2=X2*4=X2*3*4=X2*12
.L3 :
        ldr     w4, [x1],28         ; charger en X1, ajouter 28 à X1 (post-incrémentation)
        str     w4, [x0,x3]         ; stocker en X0+X3=a1+X3
        add     x3, x3, 12          ; décaler X3
        cmp     x3, x2              ; fini?
        bne     .L3
.L1 :
        ret
```

GCC 4.4.5 pour MIPS fait la même chose:

Listing 3.16: GCC 4.4.5 for MIPS avec optimisation (IDA)

```
; $a0=a1
; $a1=a2
; $a2=cnt
f :
; sauter au code de vérification de la boucle :
                beqz    $a2, locret_24
; initialiser le compteur (i) à 0:
                move    $v0, $zero ; slot de délai de branchement, NOP

loc_8 :
; charger le mot 32-bit en $a1
                lw      $a3, 0($a1)
; incrémenter le compteur (i) :
                addiu   $v0, 1
; vérifier si terminé (comparer "i" dans $v0 et "cnt" dans $a2) :
                sltu    $v1, $v0, $a2
; stocker le mot 32-bit en $a0 :
                sw      $a3, 0($a0)
; ajouter 0x1C (28) à $a1 à chaque itération :
                addiu   $a1, 0x1C
; sauter au corps de la boulce si i<cnt :
                bnez    $v1, loc_8
; ajouter 0xC (12) à $a0 à chaque itération :
                addiu   $a0, 0xC ; slot de délai de branchement

locret_24 :
                jr      $ra
                or      $at, $zero ; slot de délai de branchement, NOP
```

3.7.3 Cas Intel C++ 2011

Les optimisations du compilateur peuvent être bizarre, mais néanmoins toujours correctes. Voici ce qu le compilateur Intel C++ 2011 effectue:

Listing 3.17: Intel C++ 2011 x64 avec optimisation

```
f       PROC
; parameter 1: rcx = a1
; parameter 2: rdx = a2
; parameter 3: r8  = cnt
.B1.1::
        test      r8, r8
        jbe       exit

.B1.2::
        cmp       r8, 6
        jbe       just_copy

.B1.3::
        cmp       rcx, rdx
        jbe       .B1.5

.B1.4::
        mov       r10, r8
        mov       r9, rcx
        shl       r10, 5
        lea       rax, QWORD PTR [r8*4]
        sub       r9, rdx
        sub       r10, rax
        cmp       r9, r10
        jge       just_copy2

.B1.5::
        cmp       rdx, rcx
        jbe       just_copy

.B1.6::
        mov       r9, rdx
```

```
        lea       rax, QWORD PTR [r8*8]
        sub       r9, rcx
        lea       r10, QWORD PTR [rax+r8*4]
        cmp       r9, r10
        jl        just_copy

just_copy2 ::
; R8 = cnt
; RDX = a2
; RCX = a1
        xor       r10d, r10d
        xor       r9d, r9d
        xor       eax, eax

.B1.8::
        mov       r11d, DWORD PTR [rax+rdx]
        inc       r10
        mov       DWORD PTR [r9+rcx], r11d
        add       r9, 12
        add       rax, 28
        cmp       r10, r8
        jb        .B1.8
        jmp       exit

just_copy ::
; R8 = cnt
; RDX = a2
; RCX = a1
        xor       r10d, r10d
        xor       r9d, r9d
        xor       eax, eax

.B1.11::
        mov       r11d, DWORD PTR [rax+rdx]
        inc       r10
        mov       DWORD PTR [r9+rcx], r11d
        add       r9, 12
        add       rax, 28
        cmp       r10, r8
        jb        .B1.11

exit ::
        ret
```

Tout d'abord, quelques décisions sont prises, puis une des routines est exécutée.

Il semble qu'il teste si les tableaux se recoupent.

C'est une façons très connue d'optimiser les routines de copie de blocs de mémoire. Mais les routines de copie sont les même!

ça doit être une erreur de l'optimiseur Intel C++, qui produit néanmoins un code fonctionnel.

Nous prenons volontairement en compte de tels exemples dans ce livre, afin que lecteur comprenne que le ce que génère un compilateur est parfois bizarre mais toujours correct, car lorsque le compilateur a été testé, il a réussi les tests.

3.8 Duff's device

Le dispositif de Duff[9] est une boucle déroulée avec la possibilité d'y sauter au milieu. La boucle déroulée est implémentée en utilisant une déclaration switch() sans arrêt. Nous allons utiliser ici une version légèrement simplifiée du code original de Tom Duff. Disons que nous voulons écrire une fonction qui efface une zone en mémoire. On pourrait le faire avec une simple boucle, effaçant octet par octet. C'est étonnement lent, puisque tous les ordinateurs modernes ont des bus mémoire bien plus large. Donc, la meilleure façon de faire est d'effacer des zones de mémoire en utilisant des blocs de 4 ou 8 octets. Comme nous allons travailler ici avec un exemple 64-bit, nous allons effacer la mémoire par bloc de 8 octets. Jusqu'ici, tout

[9] Wikipédia

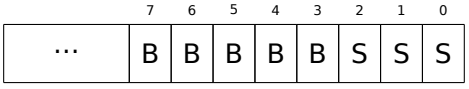
va bien. Mais qu'en est-il du reste? La routine de mise à zéro de la mémoire peut aussi être appelée pour des zones de taille non multiple de 8. Voici l'algorithme:

- calculer le nombre de bloc de 8 octets, les effacer en utilisant des accès mémoire 8-octets (64-bit);
- calculer la taille du reste, l'effacer en utilisant ces accès mémoire d'un octet.

La seconde étape peut être implémentée en utilisant une simple boucle. Mais implémentons-là avec une boucle déroulée:

```
#include <stdint.h>
#include <stdio.h>

void bzero(uint8_t* dst, size_t count)
{
        int i;

        if (count&(~7))
                // traiter les blocs de 8 octets
                for (i=0; i<count>>3; i++)
                {
                        *(uint64_t*)dst=0;
                        dst=dst+8;
                };

        // traiter le rset
        switch(count & 7)
        {
        case 7: *dst++ = 0;
        case 6: *dst++ = 0;
        case 5: *dst++ = 0;
        case 4: *dst++ = 0;
        case 3: *dst++ = 0;
        case 2: *dst++ = 0;
        case 1: *dst++ = 0;
        case 0: // ne rien faire
                break;
        }
}
```

Tout d'abord, comprenons comment le calcul est effectué. La taille de la zone mémoire est passée comme une valeur 64-bit. Et cette valeur peut être divisée en deux parties:



(« B » est le nombre de blocs de 8-bit et « S » est la longueur du reste en octets).

Lorsque nous divisons la taille de la zone de mémoire entrée, la valeur est juste décalée de 3 bits vers la droite. Mais pour calculer le reste nous pouvons simplement isoler les 3 bits les plus bas! Donc, le nombre de bloc de 8-octets est calculé comme $count >> 3$ et le reste comme $count\&7$. Nous devons aussi savoir si nous allons exécuter la procédure 8-octets, donc nous devons vérifier si la valeur de $count$ est plus grande que 7. Nous le faisons en mettant à zéro les 3 bits les plus faible et en comparant le résultat avec zéro, car tout ce dont nous avons besoin pour répondre à la question est la partie haute de $count$ non nulle, Bien sûr, ceci fonctionne car 8 est 2^3 et que diviser par des nombres de la forme 2^n est facile. Ce n'est pas possible pour d'autres nombres. Il est difficile de dire si ces astuces valent la peine d'être utilisées, car elles conduisent à du code difficile à lire. Toutefois, ces astuces sont très populaires et un programmeur pratiquant, même s'il/si elle ne va pas les utiliser, doit néanmoins les comprendre. Donc la première partie est simple: obtenir le nombre de blocs de 8-octets et écrire des valeurs 64-bits zéro en mémoire La seconde partie est une boucle déroulée implémentée avec une déclaration switch() sans arrêt.

Premièrement, exprimons en français ce que nous faisons ici.

Nous devons « écrire autant d'octets à zéro en mémoire, que la valeur $count\&7$ nous l'indique ». Si c'est 0, sauter à la fin, et il n'y a rien à faire. Si c'est 1, sauter à l'endroit à l'intérieur de la déclaration switch() où une seule opération de stockage sera exécutée. Si c'est 2, sauter à un autre endroit, où deux opérations de stockage seront exécutées, etc. Une valeur d'entrée de 7 conduit à l'exécution de toutes les 7 opérations. Il n'y a pas de 8, car une zone mémoire de 8 octets serait traitée par la première partie de notre fonction. Donc, nous avons écrit une boucle déroulée. C'était assurément plus rapide sur les anciens ordinateurs

que les boucles normales (et au contraire, les CPUs récents travaillent mieux avec des boucles courtes qu'avec des boucles déroulées). Peut-être est-ce encore utile sur les MCU[10]s embarqués moderne à bas coût.

Voyons ce que MSVC 2012 avec optimisation fait:

```
dst$ = 8
count$ = 16
bzero   PROC
        test    rdx, -8
        je      SHORT $LN11@bzero
; traiter les blocs de 8 octets
        xor     r10d, r10d
        mov     r9, rdx
        shr     r9, 3
        mov     r8d, r10d
        test    r9, r9
        je      SHORT $LN11@bzero
        npad    5
$LL19@bzero :
        inc     r8d
        mov     QWORD PTR [rcx], r10
        add     rcx, 8
        movsxd  rax, r8d
        cmp     rax, r9
        jb      SHORT $LL19@bzero
$LN11@bzero :
; traiter le reste
        and     edx, 7
        dec     rdx
        cmp     rdx, 6
        ja      SHORT $LN9@bzero
        lea     r8, OFFSET FLAT :__ImageBase
        mov     eax, DWORD PTR $LN22@bzero[r8+rdx*4]
        add     rax, r8
        jmp     rax
$LN8@bzero :
        mov     BYTE PTR [rcx], 0
        inc     rcx
$LN7@bzero :
        mov     BYTE PTR [rcx], 0
        inc     rcx
$LN6@bzero :
        mov     BYTE PTR [rcx], 0
        inc     rcx
$LN5@bzero :
        mov     BYTE PTR [rcx], 0
        inc     rcx
$LN4@bzero :
        mov     BYTE PTR [rcx], 0
        inc     rcx
$LN3@bzero :
        mov     BYTE PTR [rcx], 0
        inc     rcx
$LN2@bzero :
        mov     BYTE PTR [rcx], 0
$LN9@bzero :
        fatret  0
        npad    1
$LN22@bzero :
        DD      $LN2@bzero
        DD      $LN3@bzero
        DD      $LN4@bzero
        DD      $LN5@bzero
        DD      $LN6@bzero
        DD      $LN7@bzero
        DD      $LN8@bzero
bzero   ENDP
```

[10] Microcontroller Unit

La premières partie de la fonction est prévisible. La seconde partie est juste une boucle déroulée et un saut y passant le contrôle du flux à la bonne instruction. Il n'y a pas d'autre code entre la paire d'instructions MOV/INC, donc l'exécution va continuer jusqu'à la fin, exécutant autant de paires d'instructions que nécessaire. Á propos, nous pouvons observer que la paire d'instructions MOV/INC utilise un nombre fixe d'octets (3+3). Donc la paire utilise 6 octets. Sachant cela, mous pouvons nous passer de la table des sauts de switch(), nous pouvons simplement multiplier la valeur en entrée par 6 et sauter en $current\_RIP + input\_value * 6$.

Ceci peut aussi être plus rapide car nous ne devons pas aller chercher une valeur dans la table des sauts.

Il est possible que 6 ne soit pas une très bonne constante pour une multiplication rapide et peut-être que ça n'en vaut pas la peine, mais vous voyez l'idée[11].

C'est ce que les démomakers old-school faisaient dans le passé avec les boucles déroulées.

3.8.1 Faut-il utiliser des boucles déroulées?

Les boucles déroulées peuvent être bénéfiques si il n'y a pas de cache mémoire rapide entre la RAM et le CPU, et que le CPU, afin d'avoir le code de l'instruction suivante, doit le charger depuis la mémoire à chaque fois. C'est le cas des MCU low-cost moderne et des anciens CPUs.

Les boucles déroulées sont plus lentes que les boucles courtes si il y a un cache rapide entre la RAM et le CPU, et que le corps de la boucle tient dans le cache, et que le CPU va charger le code depuis ce dernier sans toucher à la RAM. Les boucles rapides sont les boucles dont le corps tient dans le cache L1, mais des boucles encore plus rapide sont ces petites qui tiennent dans le cache des micro-opérations.

3.9 Division par la multiplication

Une fonction très simple:

```
int f(int a)
{
        return a/9;
};
```

3.9.1 x86

...est compilée de manière très prédictive:

Listing 3.18: MSVC

```
_a$ = 8             ; taille = 4
_f    PROC
    push   ebp
    mov    ebp, esp
    mov    eax, DWORD PTR _a$[ebp]
    cdq            ; extension du signe de EAX dans EDX :EAX
    mov    ecx, 9
    idiv   ecx
    pop    ebp
    ret    0
_f  ENDP
```

IDIV divise le nombre 64-bit stocké dans la paire de registres EDX:EAX par la valeur dans ECX. Comme résultat, EAX contiendra le quotient, et EDX— le reste. Le résultat de la fonction f() est renvoyé dans le registre EAX, donc la valeur n'est pas déplacée après la division, elle est déjà à la bonne place.

Puisque IDIV utilise la valeur dans la paire de registres EDX:EAX, l'instruction CDQ (avant IDIV) étend la valeur dans EAX en une valeur 64-bit, en tenant compte du signe, tout comme MOVSX le fait.

Si nous mettons l'optimisation (/Ox), nous obtenons:

[11] Comme exercice, vous pouvez essayer de retravailler le code pour se passer de la table des sauts. La paire d'instructions peut être récrite de façon à ce qu'elle utilise 4 octets ou peut-être 8. 1 octet est aussi possible (en utilisant l'instruction STOSB).

Listing 3.19: MSVC avec optimisation

```
_a$ = 8                     ; size = 4
_f    PROC

    mov    ecx, DWORD PTR _a$[esp-4]
    mov    eax, 954437177   ; 38e38e39H
    imul   ecx
    sar    edx, 1
    mov    eax, edx
    shr    eax, 31          ; 0000001fH
    add    eax, edx
    ret    0
_f    ENDP
```

Ceci est la division par la multiplication. L'opération de multiplication est bien plus rapide. Et il possible d'utiliser cette astuce [12] pour produire du code effectivement équivalent et plus rapide.

Ceci est aussi appelé « strength reduction » dans les optimisations du compilateur.

GCC 4.4.1 génère presque le même code, même sans flag d'optimisation, tout comme MSVC avec l'optimisation:

Listing 3.20: GCC 4.4.1 sans optimisation

```
        public f
f       proc near

arg_0   = dword ptr  8

        push    ebp
        mov     ebp, esp
        mov     ecx, [ebp+arg_0]
        mov     edx, 954437177 ; 38E38E39h
        mov     eax, ecx
        imul    edx
        sar     edx, 1
        mov     eax, ecx
        sar     eax, 1Fh
        mov     ecx, edx
        sub     ecx, eax
        mov     eax, ecx
        pop     ebp
        retn
f       endp
```

3.9.2 Comment ça marche

Des mathématiques du niveau de l'école, nous pouvons nous souvenir que la division par 9 peut être remplacée par la multiplication par $\frac{1}{9}$. En fait, parfois les compilateurs font cela pour l'arithmétique en virgule flottante, par exemple, l'instruction FDIV en code x86 peut être remplacée par FMUL. Au moins MSVC 6.0 va remplacer la division par 9 par un multiplication par $0.111111...$ et parfois il est difficile d'être sûr de quelle opération il s'agissait dans le code source.

Mais lorsque nous opérons avec des valeurs entières et des registres CPU entier, nous ne pouvons pas utiliser de fractions. Toutefois, nous pouvons retravailler la fraction comme ceci:

$$result = \frac{x}{9} = x \cdot \frac{1}{9} = x \cdot \frac{1 \cdot MagicNumber}{9 \cdot MagicNumber}$$

Avec le fait que la division par 2^n est très rapide (en utilisant des décalages), nous devons maintenant trouver quels $MagicNumber$, pour lesquels l'équation suivante sera vraie: $2^n = 9 \cdot MagicNumber$.

La division par 2^{32} est quelque peu cachée: la partie basse 32-bit du produit dans EAX n'est pas utilisée (ignorée), seule la partie haute 32-bit du produit (dans EDX) est utilisée et ensuite décalée de 1 bit additionnel.

[12] En savoir plus sur la division par la multiplication dans [Henry S. Warren, *Hacker's Delight*, (2002)10-3]

Autrement dit, le code assembleur que nous venons de voir multiplie par $\frac{954437177}{2^{32+1}}$, ou divise par $\frac{2^{32+1}}{954437177}$. Pour trouver le diviseur, nous avons juste à diviser le numérateur par le dénominateur. En utilisant Wolfram Alpha, nous obtenons 8.99999999.... comme résultat (qui est proche de 9).

En lire plus à ce sujet dans [Henry S. Warren, *Hacker's Delight*, (2002)10-3].

Quelques mots pour une meilleure compréhension. Beaucoup de gens manquent la division "cachée" par 2^{32} or 2^{64}, lorsque la partie basse 32-bit (ou la partie 64-bit) du produit n'est pas utilisée. Il y a aussi l'idée erronée que le modulo inverse est utilisé ici. Ceci est proche, mais pas la même chose. L'algorithme d'Euclide étendu est utilisé d'habitude pour trouver le *coéfficient magique*, mais en fait, cet algorithme est plutôt utilisé pour résoudre l'équation. Vous pouvez la résoudre en utilisant n'importe quelle autre méthode. Quoi qu'il en soit, l'algorithme d'Euclide est sans doute le moyen le plus efficient de la résoudre. De plus, inutile de le mentionner, l'équation est insoluble pour certains diviseurs, car ceci est une équation diophantienne (i.e., une équation qui n'admet que des solutions entières), puisque nous travaillons sur des registres CPU entiers, après tout.

3.9.3 ARM

Le processeur ARM, tout comme un autre processeur « pur » RISC n'a pas d'instruction pour la division. Il manque aussi une simple instruction pour la multiplication avec une constante 32-bit (rappelez-vous qu'une constante 32-bit ne tient pas dans un opcode 32-bit).

En utilisant de cette astuce intelligente (ou *hack*), il est possible d'effectuer la division en utilisant seulement trois instructions: addition, soustraction et décalages de bit (1.22 on page 306).

Voici un exemple qui divise un nombre 32-bit par 10, tiré de [Advanced RISC Machines Ltd, *The ARM Cookbook*, (1994)3.3 Division by a Constant]. La sortie est constituée du quotient et du reste.

```
; prend l'argument dans a1
; renvoie le quotient dans a1, le reste dans a2
; on peut utiliser moins de cycles si seul le quotient ou le reste est requis
    SUB    a2, a1, #10             ; garde (x-10) pour plus tard
    SUB    a1, a1, a1, lsr #2
    ADD    a1, a1, a1, lsr #4
    ADD    a1, a1, a1, lsr #8
    ADD    a1, a1, a1, lsr #16
    MOV    a1, a1, lsr #3
    ADD    a3, a1, a1, asl #2
    SUBS   a2, a2, a3, asl #1      ; calcule (x-10) - (x/10)*10
    ADDPL  a1, a1, #1              ; fix-up quotient
    ADDMI  a2, a2, #10             ; fix-up reste
    MOV    pc, lr
```

avec optimisation Xcode 4.6.3 (LLVM) (Mode ARM)

```
__text :00002C58 39 1E 08 E3 E3 18 43 E3  MOV    R1, 0x38E38E39
__text :00002C60 10 F1 50 E7              SMMUL  R0, R0, R1
__text :00002C64 C0 10 A0 E1              MOV    R1, R0,ASR#1
__text :00002C68 A0 0F 81 E0              ADD    R0, R1, R0,LSR#31
__text :00002C6C 1E FF 2F E1              BX     LR
```

Ce code est presque le même que celui généré par MSVC avec optimisation et GCC.

Il semble que LLVM utilise le même algorithme pour générer des constantes.

Le lecteur attentif pourrait se demander comment `MOV` écrit une valeur 32-bit dans un registre, alors que ceci n'est pas possible en mode ARM.

C'est impossible, en effet, mais on voit qu'il y a 8 octets par instruction, au lieu des 4 standards, en fait, ce sont deux instructions.

La première instruction charge `0x8E39` dans les 16 bits bas du registre et la seconde instruction est `MOVT`, qui charge `0x383E` dans les 16 bits hauts du registre. IDA reconnaît de telles séquences, et par concision, il les réduit a une seule « pseudo-instruction ».

L'instruction SMMUL (*Signed Most Significant Word Multiply* mot le plus significatif d'une multiplication signée), multiplie deux nombres, les traitant comme des nombres signés et laisse la partie 32-bit haute dans le registre R0, en ignorant la partie 32-bit basse du résultat.

L'instruction « MOV R1, R0,ASR#1 » est le décalage arithmétique à droite d'un bit.

« ADD R0, R1, R0,LSR#31 » est $R0 = R1 + R0 >> 31$

Il n'y a pas d'instruction de décalage séparée en mode ARM. A la place, des instructions comme (MOV, ADD, SUB, RSB)[13] peuvent avoir un suffixe, indiquant si le second argument doit être décalé, et si oui, de quelle valeur et comment. ASR signifie *Arithmetic Shift Right*, LSR—*Logical Shift Right*.

avec optimisation Xcode 4.6.3 (LLVM) (Mode Thumb-2)

```
MOV             R1, 0x38E38E39
SMMUL.W         R0, R0, R1
ASRS            R1, R0, #1
ADD.W           R0, R1, R0,LSR#31
BX              LR
```

Il y a dex instructions de décalage séparées en mode Thumb, et l'une d'elle est utilisée ici—ASRS (arithmetic shift right).

sans optimisation Xcode 4.6.3 (LLVM) and Keil 6/2013

LLVM sans optimisation ne génère pas le code que nous avons vu avant dans cette section, mais insère à la place un appel à la fonction de bibliothèque *__divsi3*.

À propos de Keil: il insère un appel à la fonction de bibliothèque *__aeabi_idivmod* dans tous les cas.

3.9.4 MIPS

Pour une raison quelconque, GCC 4.4.5 avec optimisation génère seulement une instruction de division:

Listing 3.21: avec optimisation GCC 4.4.5 (IDA)

```
f :
                li      $v0, 9
                bnez    $v0, loc_10
                div     $a0, $v0 ; slot de délai de branchement
                break   0x1C00   ; "break 7" en assembleur sortir et objdump

loc_10 :
                mflo    $v0
                jr      $ra
                or      $at, $zero ; slot de délai de branchement, NOP
```

Ici, nous voyons une nouvelle instruction: BREAK. Elle lève simplement une exception.

Dans ce cas, une exception est levée si le diviseur est zéro (il n'est pas possible de diviser par zéro dans les mathématiques conventionnelles).

Mais GCC n'a probablement pas fait correctement le travail d'optimisation et n'a pas vu que $V0 ne vaut jamais zéro.

Donc le test est laissé ici. Donc, si $V0 est zéro, BREAK est exécuté, signalant l'exception à l'OS.

Autrement, MFLO s'exécute, qui prend le résultat de la division depuis le registre LO et le copie dans $V0.

À propos, comme on devrait le savoir, l'instruction MUL laisse les 32bits hauts du résultat dans le registre HI et les 32 bits bas dans le registre LO.

DIV laisse le résultat dans le registre LO, et le reste dans le registre HI.

Si nous modifions la déclaration en « a % 9 », l'instruction MFHI est utilisée au lieu de MFLO.

[13] Ces instructions sont également appelées « instructions de traitement de données »

3.9.5 Exercice

- http://challenges.re/27

3.10 Conversion de chaîne en nombre (atoi())

Essayons de ré-implémenter la fonction C standard atoi().

3.10.1 Exemple simple

Voici la manière la plus simple possible de lire un nombre encodé en ASCII.

C'est sujet aux erreurs: un caractère autre qu'un nombre conduit à un résultat incorrect.

```
#include <stdio.h>

int my_atoi (char *s)
{
        int rt=0;

        while (*s)
        {
                rt=rt*10 + (*s-'0');
                s++;
        };

        return rt;
};

int main()
{
        printf ("%d\n", my_atoi ("1234"));
        printf ("%d\n", my_atoi ("1234567890"));
};
```

Donc, tout ce que fait l'algorithme, c'est de lire les chiffres de gauche à droite.

Le caractère ASCII zéro est soustrait de chaque chiffre.

Les chiffres de « 0 » à « 9 » sont consécutifs dans la table ASCII, donc nous n'avons même pas besoin de connaître la valeur exacte du caractère « 0 ».

Tout ce que nous avons besoin de savoir, c'est que « 0 » moins « 0 » vaut 0, « 9 » moins « 0 » vaut 9, et ainsi de suite.

Soustraire « 0 » de chaque caractère résulte en un nombre de 0 à 9 inclus.

Tout autre caractère conduit à un résultat incorrect, bien sûr!

Chaque chiffre doit être ajouté au résultat final (dans la variable « rt »), mais le résultat final est aussi multiplié par 10 à chaque chiffre.

Autrement dit, le résultat est décalé à gauche d'une position au format décimal à chaque itération.

Le dernier chiffre est ajouté, mais il n'y a pas de décalage.

MSVC 2013 x64 avec optimisation

Listing 3.22: MSVC 2013 x64 avec optimisation

```
s$ = 8
my_atoi PROC
; charger le premier caractère
        movzx   r8d, BYTE PTR [rcx]
; EAX est alloué pour la variable "rt"
; qui vaut 0 au début
        xor     eax, eax
```

```
; est-ce que le premier caractère est un octet à zéro, i.e., fin de chaine?
; si oui, sortir
        test    r8b, r8b
        je      SHORT $LN9@my_atoi
$LL2@my_atoi :
        lea     edx, DWORD PTR [rax+rax*4]
; EDX=RAX+RAX*4=rt+rt*4=rt*5
        movsx   eax, r8b
; EAX=caractère en entrée
; charger le caractère suivant dans _R8D
        movzx   r8d, BYTE PTR [rcx+1]
; décaler le pointeur dans RCX sur le caractère suivant :
        lea     rcx, QWORD PTR [rcx+1]
        lea     eax, DWORD PTR [rax+rdx*2]
; EAX=RAX+RDX*2=caractère en entrée + rt*5*2=caractère en entrée + rt*10
 : chiffre correct en soustrayant 48 (0x30 ou '0')
        add     eax, -48                                ; ffffffffffffffd0H
; est-ce que le dernier caractère était zéro?
        test    r8b, r8b
; sauter au début de la boucle si non
        jne     SHORT $LL2@my_atoi
$LN9@my_atoi :
        ret     0
my_atoi ENDP
```

Un caractère peut-être chargé à deux endroits: le premier caractère et tous les caractères subséquents. Ceci est arrangé de cette manière afin de regrouper les boucles.

Il n'y a pas d'instructions pour multiplier par 10, à la place, deux instructions LEA le font.

Parfois, MSVC utilise l'instruction ADD avec une constante négative à la place d'un SUB. C'est le cas.

C'est très difficile de dire pourquoi c'est meilleur que SUB. Mais MSVC fait souvent ceci.

GCC 4.9.1 x64 avec optimisation

GCC 4.9.1 avec optimisation est plus concis, mais il y a une instruction RET redondante à la fin. Une suffit.

Listing 3.23: GCC 4.9.1 x64 avec optimisation

```
my_atoi :
 : charger le caractère en entrée dans EDX
        movsx   edx, BYTE PTR [rdi]
; EAX est alloué pour la variable "rt"
        xor     eax, eax
 : sortir, si le caractère chargé est l'octet nul
        test    dl, dl
        je      .L4
.L3 :
        lea     eax, [rax+rax*4]
; EAX=RAX*5=rt*5
; décaler le pointeur sur le caractère suivant :
        add     rdi, 1
        lea     eax, [rdx-48+rax*2]
; EAX=caractère en entrée - 48 + RAX*2 = caractère en entrée - '0' + rt*10
; charger le caractère suivant :
        movsx   edx, BYTE PTR [rdi]
; sauter au début de la boucle, si le caractère chargé n'est pas l'octet nul
        test    dl, dl
        jne     .L3
        rep ret
.L4 :
        rep ret
```

avec optimisation Keil 6/2013 (Mode ARM)

Listing 3.24: avec optimisation Keil 6/2013 (Mode ARM)

```
my_atoi PROC
; R1 contiendra le pointeur sur le caractère
        MOV      r1,r0
; R0 contiendra la variable "rt"
        MOV      r0,#0
        B        |L0.28|
|L0.12|
        ADD      r0,r0,r0,LSL #2
; R0=R0+R0<<2=rt*5
        ADD      r0,r2,r0,LSL #1
; R0=caractère entré +rt*5<<1 = caractère entré + rt*10
; corriger le tout en soustrayant '0' de rt :
        SUB      r0,r0,#0x30
; décaler le pointeur sur le caractère suivant :
        ADD      r1,r1,#1
|L0.28|
; charger le caractère entré dans R2
        LDRB     r2,[r1,#0]
; est-ce que c'est l'octet nul? si non, sauter au corps de la boucle.
        CMP      r2,#0
        BNE      |L0.12|
; sortir si octet nul.
; la variable "rt" est encore dans le registre R0, prête à être utilisée dans
; la fonction appelante
        BX       lr
        ENDP
```

avec optimisation Keil 6/2013 (Mode Thumb)

Listing 3.25: avec optimisation Keil 6/2013 (Mode Thumb)

```
my_atoi PROC
; R1 est le pointeur sur le caractère en entrée
        MOVS     r1,r0
; R0 est alloué pour la variable "rt"
        MOVS     r0,#0
        B        |L0.16|
|L0.6|
        MOVS     r3,#0xa
; R3=10
        MULS     r0,r3,r0
; R0=R3*R0=rt*10
; décaler le pointeur sur le caractère suivant :
        ADDS     r1,r1,#1
 : corriger le tout en lui soustrayant le 'caractère 0' :
        SUBS     r0,r0,#0x30
        ADDS     r0,r2,r0
; rt=R2+R0=caractère entré + (rt*10 - '0')
|L0.16|
; charger le caractère entré dans R2
        LDRB     r2,[r1,#0]
; est-ce zéro?
        CMP      r2,#0
; sauter au corps de la boucle si non
        BNE      |L0.6|
; la variable rt est maintenant dans R0, prête a être utilisé dans la fonction appelante
        BX       lr
        ENDP
```

De façon intéressante, nous pouvons nous rappeler des cours de mathématiques que l'ordre des opérations d'addition et de soustraction n'a pas d'importance.

C'est notre cas: d'abord, l'expression $rt * 10 - '0'$ est calculée, puis la valeur du caractère en entrée lui est ajoutée.

En effet, le résultat est le même, mais le compilateur a fait quelques regroupements.

GCC 4.9.1 ARM64 avec optimisation

Le compilateur ARM64 peut utiliser le suffixe de pré-incrémentation pour les instructions:

Listing 3.26: GCC 4.9.1 ARM64 avec optimisation

```
my_atoi :
; charger le caractère en entrée dans W1
        ldrb    w1, [x0]
        mov     x2, x0
; X2=adresse de la chaîne en entrée
; est-ce que le caractère chargé est zéro ?
 : sauter à la sortie si oui
; W1 contiendra 0 dans ce cas
; il sera rechargé dans W0 en L4.
        cbz     w1, .L4
; W0 contiendra la variable "rt"
; initialisons-la à zéro
        mov     w0, 0
.L3 :
; soustraire 48 ou '0' de la variable en entrée et mettre le résultat dans W3 :
        sub     w3, w1, #48
; charger le caractère suivant à l'adresse X2+1 dans W1 avec pré-incrémentation :
        ldrb    w1, [x2,1]!
        add     w0, w0, w0, lsl 2
; W0=W0+W0<<2=W0+W0*4=rt*5
        add     w0, w3, w0, lsl 1
; W0=chiffre entrée + W0<<1 = chiffre entrée + rt*5*2 = chiffre entrée + rt*10
; si le caractère que nous venons de charger n'est pas l'octet nul,
; sauter au début de la boucle
        cbnz    w1, .L3
; la variable qui doit être retournée (rt) est dans W0, prête à être utilisée
; dans la fonction appelante
        ret
.L4 :
        mov     w0, w1
        ret
```

3.10.2 Un exemple légèrement avancé

Mon nouvel extrait de code est plus avancé, maintenant il teste le signe « moins » au premier caractère et renvoie une erreur si un caractère autre qu'un chiffre est trouvé dans la chaîne en entrée:

```
#include <stdio.h>

int my_atoi (char *s)
{
        int negative=0;
        int rt=0;

        if (*s=='-')
        {
                negative=1;
                s++;
        };

        while (*s)
        {
                if (*s<'0' || *s>'9')
                {
                        printf ("Error! Unexpected char : '%c'\n", *s);
                        exit(0);
                };
                rt=rt*10 + (*s-'0');
                s++;
        };

        if (negative)
```

```
                return -rt;
        return rt;
};

int main()
{
        printf ("%d\n", my_atoi ("1234"));
        printf ("%d\n", my_atoi ("1234567890"));
        printf ("%d\n", my_atoi ("-1234"));
        printf ("%d\n", my_atoi ("-1234567890"));
        printf ("%d\n", my_atoi ("-a1234567890")); // error
};
```

GCC 4.9.1 x64 avec optimisation

Listing 3.27: GCC 4.9.1 x64 avec optimisation

```
.LC0 :
        .string "Error! Unexpected char : '%c'\n"

my_atoi :
        sub     rsp, 8
        movsx   edx, BYTE PTR [rdi]
; tester si c'est le signe moins
        cmp     dl, 45 ; '-'
        je      .L22
        xor     esi, esi
        test    dl, dl
        je      .L20
.L10 :
; ESI=0 ici si il n'y avait pas de signe moins et 1 si il en avait un
        lea     eax, [rdx-48]
; tout caractère autre qu'un chiffre résultera en un nombre non signé plus grand que 9 après
; soustraction donc s'il ne s'agit pas d'un chiffre, sauter en L4, oú l'erreur doit être
    rapportée
        cmp     al, 9
        ja      .L4
        xor     eax, eax
        jmp     .L6
.L7 :
        lea     ecx, [rdx-48]
        cmp     cl, 9
        ja      .L4
.L6 :
        lea     eax, [rax+rax*4]
        add     rdi, 1
        lea     eax, [rdx-48+rax*2]
        movsx   edx, BYTE PTR [rdi]
        test    dl, dl
        jne     .L7
; s'il n'y avait pas de signe moins, sauter l'instruction NEG
; s'il y en avait un, l'exécuter.
        test    esi, esi
        je      .L18
        neg     eax
.L18 :
        add     rsp, 8
        ret
.L22 :
        movsx   edx, BYTE PTR [rdi+1]
        lea     rax, [rdi+1]
        test    dl, dl
        je      .L20
        mov     rdi, rax
        mov     esi, 1
        jmp     .L10
.L20 :
        xor     eax, eax
```

```
        jmp     .L18
.L4 :
; signale une erreur. le caractère est dans EDX
        mov     edi, 1
        mov     esi, OFFSET FLAT :.LC0 ; "Error! Unexpected char : '%c'\n"
        xor     eax, eax
        call    __printf_chk
        xor     edi, edi
        call    exit
```

Si le signe « moins » a été rencontré au début de la chaîne, l'instruction NEG est exécutée à la fin. Elle rend le nombre négatif.

Il y a encore une chose à mentionner.

Comment ferait un programmeur moyen pour tester si le caractère n'est pas un chiffre? Tout comme nous l'avons dans le code source:

```
if (*s<'0' || *s>'9')
    ...
```

Il y a deux opérations de comparaison.

Ce qui est intéressant, c'est que l'on peut remplacer les deux opérations par une seule: simplement soustraire « 0 » de la valeur du caractère,

traiter le résultat comme une valeur non-signée (ceci est important) et tester s'il est plus grand que 9.

Par exemple, disons que l'entrée utilisateur contient le caractère point (« . ») qui a pour code ASCII 46. $46 - 48 = -2$ si nous traitons le résultat comme un nombre signé.

En effet, le caractère point est situé deux places avant le caractère « 0 » dans la table ASCII. Mais il correspond à 0xFFFFFFFE (4294967294) si nous traitons le résultat comme une valeur non signée, c'est définitivement plus grand que 9!

Le compilateur fait cela souvent, donc il est important de connaître ces astuces.

Un autre exemple dans ce livre: 3.16.1 on page 540.

MSVC 2013 x64 avec optimisation utilise les même astuces.

avec optimisation Keil 6/2013 (Mode ARM)

Listing 3.28: avec optimisation Keil 6/2013 (Mode ARM)

```
1  my_atoi PROC
2          PUSH     {r4-r6,lr}
3          MOV      r4,r0
4          LDRB     r0,[r0,#0]
5          MOV      r6,#0
6          MOV      r5,r6
7          CMP      r0,#0x2d '-'
8  ; R6 contiendra 1 si le signe moins a été rencontré, 0 sinon
9          MOVEQ    r6,#1
10         ADDEQ    r4,r4,#1
11         B        |L0.80|
12 |L0.36|
13         SUB      r0,r1,#0x30
14         CMP      r0,#0xa
15         BCC      |L0.64|
16         ADR      r0,|L0.220|
17         BL       __2printf
18         MOV      r0,#0
19         BL       exit
20 |L0.64|
21         LDRB     r0,[r4],#1
22         ADD      r1,r5,r5,LSL #2
23         ADD      r0,r0,r1,LSL #1
24         SUB      r5,r0,#0x30
25 |L0.80|
26         LDRB     r1,[r4,#0]
```

```
27          CMP      r1,#0
28          BNE      |L0.36|
29          CMP      r6,#0
30  ; negate result
31          RSBNE    r0,r5,#0
32          MOVEQ    r0,r5
33          POP      {r4-r6,pc}
34          ENDP
35
36  |L0.220|
37          DCB      "Error! Unexpected char : '%c'\n",0
```

Il n'y a pas d'instruction NEG en ARM 32-bit, donc l'opération « Reverse Subtraction » (ligne 31) est utilisée ici.

Elle est déclenchée si le résultat de l'instruction CMP (à la ligne 29) était « Not Equal » (non égal) (d'où le suffixe -NE).

Donc ce que fait RSBNE, c'est soustraire la valeur résultante de 0.

Cela fonctionne comme l'opération de soustraction normale, mais échange les opérandes

Soustraire n'importe quel nombre de 0 donne sa négation: $0 - x = -x$.

Le code en mode Thumb est en gros le même.

GCC 4.9 pour ARM64 peut utiliser l'instruction NEG, qui est disponible en ARM64.

3.10.3 Exercice

Oh, à propos, les chercheurs en sécurité sont souvent confrontés à un comportement imprévisible de programme lorsqu'il traite des données incorrectes.

Par exemple, lors du fuzzing. À titre d'exercice, vous pouvez essayer d'entrer des caractères qui ne soient pas des chiffres et de voir ce qui se passe.

Essayez d'expliquer ce qui s'est passé, et pourquoi.

3.11 Fonctions inline

Le code inline, c'est lorsque le compilateur, au lieu de mettre une instruction d'appel à une petite ou à une minuscule fonction, copie son corps à la place.

Listing 3.29: Un exemple simple

```
#include <stdio.h>

int celsius_to_fahrenheit (int celsius)
{
        return celsius * 9 / 5 + 32;
};

int main(int argc, char *argv[])
{
        int celsius=atol(argv[1]);
        printf ("%d\n", celsius_to_fahrenheit (celsius));
};
```

...est compilée de façon très prédictive, toutefois, si nous utilisons l'option d'optimisation de GCC (-03), nous voyons:

Listing 3.30: GCC 4.8.1 avec optimisation

```
_main :
        push    ebp
        mov     ebp, esp
        and     esp, -16
        sub     esp, 16
```

```
        call    ___main
        mov     eax, DWORD PTR [ebp+12]
        mov     eax, DWORD PTR [eax+4]
        mov     DWORD PTR [esp], eax
        call    _atol
        mov     edx, 1717986919
        mov     DWORD PTR [esp], OFFSET FLAT :LC2 ; "%d\12\0"
        lea     ecx, [eax+eax*8]
        mov     eax, ecx
        imul    edx
        sar     ecx, 31
        sar     edx
        sub     edx, ecx
        add     edx, 32
        mov     DWORD PTR [esp+4], edx
        call    _printf
        leave
        ret
```

(Ici la division est effectuée avec une multiplication(3.9 on page 501).)

Oui, notre petite fonction `celsius_to_fahrenheit()` a été placée juste avant l'appel à `printf()`.

Pourquoi? C'est plus rapide que d'exécuter la code de cette fonction plus le surcoût de l'appel/retour.

Les optimiseurs des compilateurs modernes choisissent de mettre en ligne les petites fonctions automatiquement. Mais il est possible de forcer le compilateur à mettre en ligne automatiquement certaines fonctions, en les marquants avec le mot clef « inline » dans sa déclaration.

3.11.1 Fonctions de chaînes et de mémoire

Une autre tactique courante d'optimisation automatique est la mise en ligne des fonctions de chaînes comme *strcpy()*, *strcmp()*, *strlen()*, *memset()*, *memcmp()*, *memcpy()*, etc..

Parfois, c'est plus rapide que d'appeler une fonction séparée.

Ce sont des patterns très fréquents et il est hautement recommandé aux rétro-ingénieurs d'apprendre à les détecter automatiquement.

strcmp()

Listing 3.31: exemple strcmp()

```
bool is_bool (char *s)
{
        if (strcmp (s, "true")==0)
                return true;
        if (strcmp (s, "false")==0)
                return false;

        assert(0);
};
```

Listing 3.32: avec optimisation GCC 4.8.1

```
.LC0 :
        .string "true"
.LC1 :
        .string "false"
is_bool :
.LFB0 :
        push    edi
        mov     ecx, 5
        push    esi
        mov     edi, OFFSET FLAT :.LC0
        sub     esp, 20
        mov     esi, DWORD PTR [esp+32]
        repz cmpsb
```

```
        je      .L3
        mov     esi, DWORD PTR [esp+32]
        mov     ecx, 6
        mov     edi, OFFSET FLAT :.LC1
        repz cmpsb
        seta    cl
        setb    dl
        xor     eax, eax
        cmp     cl, dl
        jne     .L8
        add     esp, 20
        pop     esi
        pop     edi
        ret
.L8 :
        mov     DWORD PTR [esp], 0
        call    assert
        add     esp, 20
        pop     esi
        pop     edi
        ret
.L3 :
        add     esp, 20
        mov     eax, 1
        pop     esi
        pop     edi
        ret
```

Listing 3.33: MSVC 2010 avec optimisation

```
$SG3454 DB      'true', 00H
$SG3456 DB      'false', 00H

_s$ = 8         ; taille = 4
?is_bool@@YA_NPAD@Z PROC ; is_bool
        push    esi
        mov     esi, DWORD PTR _s$[esp]
        mov     ecx, OFFSET $SG3454 ; 'true'
        mov     eax, esi
        npad    4 ; aligner le label suivant
$LL6@is_bool :
        mov     dl, BYTE PTR [eax]
        cmp     dl, BYTE PTR [ecx]
        jne     SHORT $LN7@is_bool
        test    dl, dl
        je      SHORT $LN8@is_bool
        mov     dl, BYTE PTR [eax+1]
        cmp     dl, BYTE PTR [ecx+1]
        jne     SHORT $LN7@is_bool
        add     eax, 2
        add     ecx, 2
        test    dl, dl
        jne     SHORT $LL6@is_bool
$LN8@is_bool :
        xor     eax, eax
        jmp     SHORT $LN9@is_bool
$LN7@is_bool :
        sbb     eax, eax
        sbb     eax, -1
$LN9@is_bool :
        test    eax, eax
        jne     SHORT $LN2@is_bool

        mov     al, 1
        pop     esi

        ret     0
$LN2@is_bool :

        mov     ecx, OFFSET $SG3456 ; 'false'
```

```
        mov     eax, esi
$LL10@is_bool :
        mov     dl, BYTE PTR [eax]
        cmp     dl, BYTE PTR [ecx]
        jne     SHORT $LN11@is_bool
        test    dl, dl
        je      SHORT $LN12@is_bool
        mov     dl, BYTE PTR [eax+1]
        cmp     dl, BYTE PTR [ecx+1]
        jne     SHORT $LN11@is_bool
        add     eax, 2
        add     ecx, 2
        test    dl, dl
        jne     SHORT $LL10@is_bool
$LN12@is_bool :
        xor     eax, eax
        jmp     SHORT $LN13@is_bool
$LN11@is_bool :
        sbb     eax, eax
        sbb     eax, -1
$LN13@is_bool :
        test    eax, eax
        jne     SHORT $LN1@is_bool

        xor     al, al
        pop     esi

        ret     0
$LN1@is_bool :

        push    11
        push    OFFSET $SG3458
        push    OFFSET $SG3459
        call    DWORD PTR __imp___wassert
        add     esp, 12
        pop     esi

        ret     0
?is_bool@@YA_NPAD@Z ENDP ; is_bool
```

strlen()

Listing 3.34: exemple strlen()

```
int strlen_test(char *s1)
{
        return strlen(s1);
};
```

Listing 3.35: avec optimisation MSVC 2010

```
_s1$ = 8 ; size = 4
_strlen_test PROC
        mov     eax, DWORD PTR _s1$[esp-4]
        lea     edx, DWORD PTR [eax+1]
$LL3@strlen_tes :
        mov     cl, BYTE PTR [eax]
        inc     eax
        test    cl, cl
        jne     SHORT $LL3@strlen_tes
        sub     eax, edx
        ret     0
_strlen_test ENDP
```

strcpy()

Listing 3.36: exemple strcpy()

```
void strcpy_test(char *s1, char *outbuf)
{
        strcpy(outbuf, s1);
};
```

Listing 3.37: avec optimisation MSVC 2010

```
_s1$ = 8        ; taille = 4
_outbuf$ = 12   ; taille = 4
_strcpy_test PROC
        mov     eax, DWORD PTR _s1$[esp-4]
        mov     edx, DWORD PTR _outbuf$[esp-4]
        sub     edx, eax
        npad    6 ; aligner le label suivant
$LL3@strcpy_tes :
        mov     cl, BYTE PTR [eax]
        mov     BYTE PTR [edx+eax], cl
        inc     eax
        test    cl, cl
        jne     SHORT $LL3@strcpy_tes
        ret     0
_strcpy_test ENDP
```

memset()

Exemple#1

Listing 3.38: 32 bytes

```
#include <stdio.h>

void f(char *out)
{
        memset(out, 0, 32);
};
```

De nombreux compilateurs ne génèrent pas un appel à memset() pour de petits blocs, mais insèrent plutôt un paquet de MOVs:

Listing 3.39: GCC 4.9.1 x64 avec optimisation

```
f :
        mov     QWORD PTR [rdi], 0
        mov     QWORD PTR [rdi+8], 0
        mov     QWORD PTR [rdi+16], 0
        mov     QWORD PTR [rdi+24], 0
        ret
```

À propos, ça nous rappelle le déroulement de boucles: 1.16.1 on page 192.

Exemple#2

Listing 3.40: 67 bytes

```
#include <stdio.h>

void f(char *out)
{
        memset(out, 0, 67);
};
```

Lorsque la taille du bloc n'est pas un multiple de 4 ou 8, les compilateurs se comportent différemment.

Par exemple, MSVC 2012 continue à insérer des MOVs:

Listing 3.41: MSVC 2012 x64 avec optimisation

```
out$ = 8
f       PROC
        xor     eax, eax
        mov     QWORD PTR [rcx], rax
        mov     QWORD PTR [rcx+8], rax
        mov     QWORD PTR [rcx+16], rax
        mov     QWORD PTR [rcx+24], rax
        mov     QWORD PTR [rcx+32], rax
        mov     QWORD PTR [rcx+40], rax
        mov     QWORD PTR [rcx+48], rax
        mov     QWORD PTR [rcx+56], rax
        mov     WORD PTR [rcx+64], ax
        mov     BYTE PTR [rcx+66], al
        ret     0
f       ENDP
```

...tandis que GCC utilise REP STOSQ, en concluant que cela sera plus petit qu'un paquet de MOVs:

Listing 3.42: GCC 4.9.1 x64 avec optimisation

```
f :
        mov     QWORD PTR [rdi], 0
        mov     QWORD PTR [rdi+59], 0
        mov     rcx, rdi
        lea     rdi, [rdi+8]
        xor     eax, eax
        and     rdi, -8
        sub     rcx, rdi
        add     ecx, 67
        shr     ecx, 3
        rep stosq
        ret
```

memcpy()

Petits blocs

La routine pour copier des blocs courts est souvent implémentée comme une séquence d'instructions MOV.

Listing 3.43: exemple memcpy()

```
void memcpy_7(char *inbuf, char *outbuf)
{
        memcpy(outbuf+10, inbuf, 7);
};
```

Listing 3.44: MSVC 2010 avec optimisation

```
_inbuf$ = 8     ; size = 4
_outbuf$ = 12   ; size = 4
_memcpy_7 PROC
        mov     ecx, DWORD PTR _inbuf$[esp-4]
        mov     edx, DWORD PTR [ecx]
        mov     eax, DWORD PTR _outbuf$[esp-4]
        mov     DWORD PTR [eax+10], edx
        mov     dx, WORD PTR [ecx+4]
        mov     WORD PTR [eax+14], dx
        mov     cl, BYTE PTR [ecx+6]
        mov     BYTE PTR [eax+16], cl
        ret     0
_memcpy_7 ENDP
```

Listing 3.45: GCC 4.8.1 avec optimisation

```
memcpy_7 :
```

```
        push    ebx
        mov     eax, DWORD PTR [esp+8]
        mov     ecx, DWORD PTR [esp+12]
        mov     ebx, DWORD PTR [eax]
        lea     edx, [ecx+10]
        mov     DWORD PTR [ecx+10], ebx
        movzx   ecx, WORD PTR [eax+4]
        mov     WORD PTR [edx+4], cx
        movzx   eax, BYTE PTR [eax+6]
        mov     BYTE PTR [edx+6], al
        pop     ebx
        ret
```

C'est effectué en général ainsi: des blocs de 4-octets sont d'abord copiés, puis un mot de 16-bit (si nécessaire) et enfin un dernier octet (si nécessaire).

Les structures sont aussi copiées en utilisant `MOV` : 1.24.4 on page 364.

Longs blocs

Les compilateurs se comportent différemment dans ce cas.

Listing 3.46: memcpy() exemple

```
void memcpy_128(char *inbuf, char *outbuf)
{
        memcpy(outbuf+10, inbuf, 128);
};

void memcpy_123(char *inbuf, char *outbuf)
{
        memcpy(outbuf+10, inbuf, 123);
};
```

Pour copier 128 octets, MSVC utilise une seule instruction `MOVSD` (car 128 est divisible par 4):

Listing 3.47: MSVC 2010 avec optimisation

```
_inbuf$ = 8             ; size = 4
_outbuf$ = 12           ; size = 4
_memcpy_128 PROC
        push    esi
        mov     esi, DWORD PTR _inbuf$[esp]
        push    edi
        mov     edi, DWORD PTR _outbuf$[esp+4]
        add     edi, 10
        mov     ecx, 32
        rep movsd
        pop     edi
        pop     esi
        ret     0
_memcpy_128 ENDP
```

Lors de la copie de 123 octets, 30 mots de 32-bit sont tout d'abord copiés en utilisant `MOVSD` (ce qui fait 120 octets), puis 2 octets sont copiés en utilisant `MOVSW`, puis un autre octet en utilisant `MOVSB`.

Listing 3.48: MSVC 2010 avec optimisation

```
_inbuf$ = 8             ; size = 4
_outbuf$ = 12           ; size = 4
_memcpy_123 PROC
        push    esi
        mov     esi, DWORD PTR _inbuf$[esp]
        push    edi
        mov     edi, DWORD PTR _outbuf$[esp+4]
        add     edi, 10
        mov     ecx, 30
        rep movsd
        movsw
```

```
        movsb
        pop     edi
        pop     esi
        ret     0
_memcpy_123 ENDP
```

GCC utilise une grosse fonction universelle, qui fonctionne pour n'importe quelle taille de bloc:

Listing 3.49: GCC 4.8.1 avec optimisation

```
memcpy_123 :
.LFB3 :
        push    edi
        mov     eax, 123
        push    esi
        mov     edx, DWORD PTR [esp+16]
        mov     esi, DWORD PTR [esp+12]
        lea     edi, [edx+10]
        test    edi, 1
        jne     .L24
        test    edi, 2
        jne     .L25
.L7 :
        mov     ecx, eax
        xor     edx, edx
        shr     ecx, 2
        test    al, 2
        rep movsd
        je      .L8
        movzx   edx, WORD PTR [esi]
        mov     WORD PTR [edi], dx
        mov     edx, 2
.L8 :
        test    al, 1
        je      .L5
        movzx   eax, BYTE PTR [esi+edx]
        mov     BYTE PTR [edi+edx], al
.L5 :
        pop     esi
        pop     edi
        ret
.L24 :
        movzx   eax, BYTE PTR [esi]
        lea     edi, [edx+11]
        add     esi, 1
        test    edi, 2
        mov     BYTE PTR [edx+10], al
        mov     eax, 122
        je      .L7
.L25 :
        movzx   edx, WORD PTR [esi]
        add     edi, 2
        add     esi, 2
        sub     eax, 2
        mov     WORD PTR [edi-2], dx
        jmp     .L7
.LFE3 :
```

Les fonctions de copie de mémoire universelles fonctionnent en général comme suit: calculer combien de mots de 32-bit peuvent être copiés, puis les copier en utilisant `MOVSD`, et enfin copier les octets restants.

Des fonctions de copie plus avancées et complexes utilisent les instructions SIMD et prennent aussi en compte l'alignement de la mémoire.

Voici un exemple de fonction strlen() SIMD: 1.29.2 on page 418.

memcmp()

Listing 3.50: exemple memcmp()

```
int memcmp_1235(char *buf1, char *buf2)
{
        return memcmp(buf1, buf2, 1235);
};
```

Pour n'importe quelle taille de bloc, MSVC 2013 insère la même fonction universelle:

Listing 3.51: avec optimisation MSVC 2010

```
_buf1$ = 8      ; size = 4
_buf2$ = 12     ; size = 4
_memcmp_1235 PROC
        mov     ecx, DWORD PTR _buf1$[esp-4]
        mov     edx, DWORD PTR _buf2$[esp-4]
        push    esi
        mov     esi, 1231
        npad    2
$LL5@memcmp_123 :
        mov     eax, DWORD PTR [ecx]
        cmp     eax, DWORD PTR [edx]
        jne     SHORT $LN4@memcmp_123
        add     ecx, 4
        add     edx, 4
        sub     esi, 4
        jae     SHORT $LL5@memcmp_123
$LN4@memcmp_123 :
        mov     al, BYTE PTR [ecx]
        cmp     al, BYTE PTR [edx]
        jne     SHORT $LN6@memcmp_123
        mov     al, BYTE PTR [ecx+1]
        cmp     al, BYTE PTR [edx+1]
        jne     SHORT $LN6@memcmp_123
        mov     al, BYTE PTR [ecx+2]
        cmp     al, BYTE PTR [edx+2]
        jne     SHORT $LN6@memcmp_123
        cmp     esi, -1
        je      SHORT $LN3@memcmp_123
        mov     al, BYTE PTR [ecx+3]
        cmp     al, BYTE PTR [edx+3]
        jne     SHORT $LN6@memcmp_123
$LN3@memcmp_123 :
        xor     eax, eax
        pop     esi
        ret     0
$LN6@memcmp_123 :
        sbb     eax, eax
        or      eax, 1
        pop     esi
        ret     0
_memcmp_1235 ENDP
```

strcat()

Ceci est un strcat() inline tel qu'il a été généré par MSVC 6.0. Il y a 3 parties visibles: 1) obtenir la longueur de la chaîne source (premier `scasb`); 2) obtenir la longueur de la chaîne destination (second `scasb`); 3) copier la chaîne source dans la fin de la chaîne de destination (paire `movsd`/`movsb`).

Listing 3.52: strcat()

```
                lea     edi, [src]
                or      ecx, 0FFFFFFFFh
                repne scasb
                not     ecx
                sub     edi, ecx
                mov     esi, edi
                mov     edi, [dst]
                mov     edx, ecx
```

```
                or      ecx, 0FFFFFFFFh
                repne scasb
                mov     ecx, edx
                dec     edi
                shr     ecx, 2
                rep movsd
                mov     ecx, edx
                and     ecx, 3
                rep movsb
```

Script IDA

Il y a aussi un petit script IDA pour chercher et suivre de tels morceaux de code inline, que l'on rencontre fréquemment:

GitHub.

3.12 C99 restrict

Voici une raison pour laquelle les programmes en Fortran, dans certains cas, fonctionnent plus vite que ceux en C/C++.

```
void f1 (int* x, int* y, int* sum, int* product, int* sum_product, int* update_me, size_t s)
{
        for (int i=0; i<s; i++)
        {
                sum[i]=x[i]+y[i];
                product[i]=x[i]*y[i];
                update_me[i]=i*123; // some dummy value
                sum_product[i]=sum[i]+product[i];
        };
};
```

C'est un exemple très simple, qui contient une spécificité: le pointeur sur le tableau `update_me` peut-être un pointeur sur le tableau `sum`, le tableau `product` ou même le tableau `sum_product`—rien ne l'interdit, n'est-ce pas?

Le compilateur est parfaitement conscient de ceci, donc il génère du code avec quatre étapes dans le corps de la boucle:

- calcule le `sum[i]` suivant
- calcule le `product[i]` suivant
- calcule le `update_me[i]` suivant
- calcule le `sum_product[i]` suivant—à cette étape, nous devons charger depuis la mémoire les valeurs `sum[i]` et `product[i]` déjà calculées

Et-il possible d'optimiser la dernière étape? Puisque nous avons déjà calculé `sum[i]` et `product[i]`, il n'est pas nécessaire de les charger à nouveau depuis la mémoire.

Oui, mais le compilateurs n'est pas sûr que rien n'a été récris à la 3ème étape! Ceci est appelé « pointer aliasing », une situation dans laquelle le compilateur ne peut pas être sûr que la mémoire sur laquelle le pointeur pointe n'a pas été modifiée.

restrict dans le standard C99 [*ISO/IEC 9899:TC3 (C C99 standard)*, (2007) 6.7.3/1] est une promesse faite par le programmeur au compilateur que les arguments de la fonction marqués par ce mot-clef vont toujours pointer vers des case mémoire différentes et ne vont jamais se recouper.

Pour être plus précis et décrire ceci formellement, *restrict* indique que seul ce pointeur est utilisé pour accéder un objet, et qu'aucun autre pointeur ne sera utilisé pour ceci.

On peut même dire que l'objet ne sera accéder que par un seul pointeur, si il est marqué comme *restrict*.

Ajoutons ce mot-clef à chaque argument pointeur:

```
void f2 (int* restrict x, int* restrict y, int* restrict sum, int* restrict product, int* ↙
    ↳ restrict sum_product,
        int* restrict update_me, size_t s)
{
        for (int i=0; i<s ; i++)
        {
                sum[i]=x[i]+y[i];
                product[i]=x[i]*y[i];
                update_me[i]=i*123; // some dummy value
                sum_product[i]=sum[i]+product[i];
        };
};
```

Regardons le résultat:

Listing 3.53: GCC x64: f1()

```
f1 :
        push    r15 r14 r13 r12 rbp rdi rsi rbx
        mov     r13, QWORD PTR 120[rsp]
        mov     rbp, QWORD PTR 104[rsp]
        mov     r12, QWORD PTR 112[rsp]
        test    r13, r13
        je      .L1
        add     r13, 1
        xor     ebx, ebx
        mov     edi, 1
        xor     r11d, r11d
        jmp     .L4
.L6 :
        mov     r11, rdi
        mov     rdi, rax
.L4 :
        lea     rax, 0[0+r11*4]
        lea     r10, [rcx+rax]
        lea     r14, [rdx+rax]
        lea     rsi, [r8+rax]
        add     rax, r9
        mov     r15d, DWORD PTR [r10]
        add     r15d, DWORD PTR [r14]
        mov     DWORD PTR [rsi], r15d         ; stocker dans sum[]
        mov     r10d, DWORD PTR [r10]
        imul    r10d, DWORD PTR [r14]
        mov     DWORD PTR [rax], r10d         ; stocker dans product[]
        mov     DWORD PTR [r12+r11*4], ebx    ; stocker dans update_me[]
        add     ebx, 123
        mov     r10d, DWORD PTR [rsi]         ; recharger sum[i]
        add     r10d, DWORD PTR [rax]         ; recharger product[i]
        lea     rax, 1[rdi]
        cmp     rax, r13
        mov     DWORD PTR 0[rbp+r11*4], r10d  ; stocker dans sum_product[]
        jne     .L6
.L1 :
        pop     rbx rsi rdi rbp r12 r13 r14 r15
        ret
```

Listing 3.54: GCC x64: f2()

```
f2 :
        push    r13 r12 rbp rdi rsi rbx
        mov     r13, QWORD PTR 104[rsp]
        mov     rbp, QWORD PTR 88[rsp]
        mov     r12, QWORD PTR 96[rsp]
        test    r13, r13
        je      .L7
        add     r13, 1
        xor     r10d, r10d
        mov     edi, 1
        xor     eax, eax
        jmp     .L10
```

```
.L11 :
        mov     rax, rdi
        mov     rdi, r11
.L10 :
        mov     esi, DWORD PTR [rcx+rax*4]
        mov     r11d, DWORD PTR [rdx+rax*4]
        mov     DWORD PTR [r12+rax*4], r10d  ; stocker dans update_me[]
        add     r10d, 123
        lea     ebx, [rsi+r11]
        imul    r11d, esi
        mov     DWORD PTR [r8+rax*4], ebx    ; stocker dans sum[]
        mov     DWORD PTR [r9+rax*4], r11d   ; stocker dans product[]
        add     r11d, ebx
        mov     DWORD PTR 0[rbp+rax*4], r11d ; stocker dans sum_product[]
        lea     r11, 1[rdi]
        cmp     r11, r13
        jne     .L11
.L7 :
        pop     rbx rsi rdi rbp r12 r13
        ret
```

La différence entre les fonctions `f1()` et `f2()` compilées est la suivante: dans `f1()`, `sum[i]` et `product[i]` sont rechargés au milieu de la boucle, et il n'y a rien de tel dans `f2()`, les valeurs déjà calculées sont utilisées, puisque nous avons « promis » au compilateur que rien ni personne ne changera les valeurs pendant l'exécution du corps de la boucle, donc il est « certain » qu'il n'y a pas besoin de recharger la valeur depuis la mémoire.

Étonnamment, le second exemple est plus rapide.

Mais que se passe-t-il si les pointeurs dans les arguments de la fonction se modifient d'une manière ou d'une autre?

Ceci est du ressort de la conscience du programmeur, et le résultat sera incorrect.

Retournons au Fortran.

Les compilateurs de ce langage traitent tous les pointeurs de cette façon, donc lorsqu'il n'est pas possible d'utiliser *restrict* en C, Fortran peut générer du code plus rapide dans ces cas.

À quel point est-ce pratique?

Dans les cas où le fonction travaille avec des gros blocs en mémoire.

C'est le cas en algèbre linéaire, par exemple.

Les superordinateurs/HPC[14] utilisent beaucoup d'algèbre linéaire, c'est probablement pourquoi, traditionnellement, Fortran y est encore utilisé [Eugene Loh, *The Ideal HPC Programming Language*, (2010)].

Mais lorsque le nombre d'itérations n'est pas très important, certainement, le gain en vitesse ne doit pas être significatif.

3.13 Fonction *abs()* sans branchement

Retravaillons un exemple que nous avons vu avant 1.14.2 on page 141 et demandons-nous, est-il possible de faire une version sans branchement de la fonction en code x86?

```
int my_abs (int i)
{
        if (i<0)
                return -i;
        else
                return i;
};
```

Et la réponse est oui.

[14] High-Performance Computing

3.13.1 GCC 4.9.1 x64 avec optimisation

Nous pouvons le voir si nous compilons en utilisant GCC 4.9 avec optimisation:

Listing 3.55: GCC 4.9 x64 avec optimisation

```
my_abs :
        mov     edx, edi
        mov     eax, edi
        sar     edx, 31
; EDX contient ici 0xFFFFFFFF si le signe de la valeur en entrée est moins
; EDX contient 0 si le signe de la valeur en entrée est plus (0 inclus)
; les deux instructions suivantes ont un effet seulement si EDX contient 0xFFFFFFFF
; et aucun si EDX contient 0
        xor     eax, edx
        sub     eax, edx
        ret
```

Voici comment ça fonctionne:

Décaler arithmétiquement la valeur en entrée par 31.

Le décalage arithmétique implique l'extension du signe, donc si le MSB est 1, les 32 bits seront tous remplis avec 1, ou avec 0 sinon.

Autrement dit, l'instruction `SAR REG, 31` donne `0xFFFFFFFF` si le signe était négatif et 0 s'il était positif.

Après l'exécution de SAR, nous avons cette valeur dans EDX.

Puis, si la valeur est `0xFFFFFFFF` (i.e., le signe est négatif), la valeur en entrée est inversée (car `XOR REG, 0xFFFFFFFF` est en effet une opération qui inverse tous les bits).

Puis, à nouveau, si la valeur est `0xFFFFFFFF` (i.e., le signe est négatif), 1 est ajouté au résultat final (car soustraire -1 d'une valeur revient à l'incrémenter).

Inverser tous les bits et incrémenter est exactement la façon dont une valeur en complément à deux est multipliée par -1. 2.2 on page 456.

Nous pouvons observer que les deux dernières instructions font quelque chose si le signe de la valeur en entrée est négatif.

Autrement (si le signe est positif) elles ne font rien du tout, laissant la valeur en entrée inchangée.

L'algorithme est expliqué dans [Henry S. Warren, *Hacker's Delight*, (2002)2-4].

Il est difficile de dire comment a fait GCC, l'a-t-il déduit lui-même ou un pattern correspondant parmi ceux connus?

3.13.2 GCC 4.9 ARM64 avec optimisation

GCC 4.9 pour ARM64 génère en gros la même chose, il décide juste d'utiliser des registres 64-bit complets.

Il y a moins d'instructions, car la valeur en entrée peut être décalée en utilisant un suffixe d'instruction (« asr ») au lieu d'une instruction séparée.

Listing 3.56: GCC 4.9 ARM64 avec optimisation

```
my_abs :
; étendre le signe de la valeur 32-bit en entrée dans le registre X0 64-bit :
        sxtw    x0, w0
        eor     x1, x0, x0, asr 63
; X1=X0^(X0>>63) (le décalage est arithmétique)
        sub     x0, x1, x0, asr 63
; X0=X1-(X0>>63)=X0^(X0>>63)-(X0>>63) (tous les décalages sont arithmétiques)
        ret
```

3.14 Fonctions variadiques

Les fonctions comme `printf()` et `scanf()` peuvent avoir un nombre variable d'arguments. Comment sont-ils accédés?

3.14.1 Calcul de la moyenne arithmétique

Imaginons que nous voulions calculer la moyenne arithmétique, et que pour une raison quelconque, nous voulions passer toutes les valeurs comme arguments de la fonction.

Mais il est impossible d'obtenir le nombre d'arguments dans une fonction variadique en C/C++, donc indiquons la valeur -1 comme terminateur.

Utilisation de la macro va_arg

Il y a le fichier d'entête standard stdarg.h qui défini des macros pour prendre en compte de tels arguments.

Les fonctions `printf()` et `scanf()` l'utilisent aussi.

```
#include <stdio.h>
#include <stdarg.h>

int arith_mean(int v, ...)
{
        va_list args;
        int sum=v, count=1, i;
        va_start(args, v);

        while(1)
        {
                i=va_arg(args, int);
                if (i==-1) // terminateur
                        break;
                sum=sum+i;
                count++;
        }

        va_end(args);
        return sum/count;
};

int main()
{
        printf ("%d\n", arith_mean (1, 2, 7, 10, 15, -1 /* terminateur */));
};
```

Le premier argument doit être traité comme un argument normal.

Tous les autres arguments sont chargés en utilisant la macro va_arg et ensuite ajoutés.

Qu'y a-t-il à l'intérieur?

Convention d'appel *cdecl*

Listing 3.57: MSVC 6.0 avec optimisation

```
_v$ = 8
_arith_mean PROC NEAR
        mov     eax, DWORD PTR _v$[esp-4] ; charger le 1er argument dans sum
        push    esi
        mov     esi, 1                    ; count=1
        lea     edx, DWORD PTR _v$[esp]   ; adresse du 1er argument
$L838 :
        mov     ecx, DWORD PTR [edx+4]    ; charger l'argument suivant
        add     edx, 4                    ; décaler le pointeur sur l'argument suivant
        cmp     ecx, -1                   ; est-ce -1?
        je      SHORT $L856               ; sortir si oui
        add     eax, ecx                  ; sum = sum + argument chargé
        inc     esi                       ; count++
        jmp     SHORT $L838
$L856 :
; calculer le quotient
```

```
        cdq
        idiv    esi
        pop     esi
        ret     0
_arith_mean ENDP

$SG851  DB      '%d', 0aH, 00H

_main   PROC NEAR
        push    -1
        push    15
        push    10
        push    7
        push    2
        push    1
        call    _arith_mean
        push    eax
        push    OFFSET FLAT :$SG851 ; '%d'
        call    _printf
        add     esp, 32
        ret     0
_main   ENDP
```

Les arguments, comme on le voit, sont passés à `main()` un par un.

Le premier argument est poussé sur la pile locale en premier.

La valeur terminale (-1) est poussée plus tard.

La fonction `arith_mean()` prend la valeur du premier argument et le stocke dans la variable sum.

Puis, elle met dans le registre EDX l'adresse du second argument, prend sa valeur, l'ajoute à sum, et fait cela dans une boucle infinie, jusqu'à ce que -1 soit trouvé.

Lorsqu'il est rencontré, la somme est divisée par le nombre de valeurs (en excluant -1) et le quotient est renvoyé.

Donc, autrement dit, la fonction traite le morceau de pile comme un tableau de valeurs entières d'une longueur infinie.

Maintenant nous pouvons comprendre pourquoi la convention d'appel *cdecl* nous force à pousser le premier argument au moins sur la pile.

Car sinon, il ne serait pas possible de trouver le premier argument, ou, pour les fonctions du genre de printf, il ne serait pas possible de trouver l'adresse de la chaîne de format.

Convention d'appel basée sur les registres

Le lecteur attentif pourrait demander, qu'en est-il de la convention d'appel où les tous premiers arguments sont passés dans des registres? Regardons:

Listing 3.58: MSVC 2012 x64 avec optimisation

```
$SG3013 DB      '%d', 0aH, 00H

v$ = 8
arith_mean PROC
        mov     DWORD PTR [rsp+8], ecx    ; 1er argument
        mov     QWORD PTR [rsp+16], rdx   ; 2nd argument
        mov     QWORD PTR [rsp+24], r8    ; 3ème argument
        mov     eax, ecx                  ; sum = 1er argument
        lea     rcx, QWORD PTR v$[rsp+8]  ; pointeur sur le 2nd argument
        mov     QWORD PTR [rsp+32], r9    ; 4ème argument
        mov     edx, DWORD PTR [rcx]      ; charger le 2nd argument
        mov     r8d, 1                    ; count=1
        cmp     edx, -1                   ; est-ce que le 2nd argument est -1?
        je      SHORT $LN8@arith_mean     ; sortir si oui
$LL3@arith_mean :
        add     eax, edx                  ; sum = sum + argument chargé
        mov     edx, DWORD PTR [rcx+8]    ; charger l'argument suivant
        lea     rcx, QWORD PTR [rcx+8]    ; décaler le pointeur pour pointer
```

```
                                        ; sur l'argument après le suivant
        inc     r8d                     ; count++
        cmp     edx, -1                 ; est-ce que l'argument chargé est -1?
        jne     SHORT $LL3@arith_mean   ; aller au début de la boucle si non
$LN8@arith_mean :
; calculer le quotient
        cdq
        idiv    r8d
        ret     0
arith_mean ENDP

main    PROC
        sub     rsp, 56
        mov     edx, 2
        mov     DWORD PTR [rsp+40], -1
        mov     DWORD PTR [rsp+32], 15
        lea     r9d, QWORD PTR [rdx+8]
        lea     r8d, QWORD PTR [rdx+5]
        lea     ecx, QWORD PTR [rdx-1]
        call    arith_mean
        lea     rcx, OFFSET FLAT :$SG3013
        mov     edx, eax
        call    printf
        xor     eax, eax
        add     rsp, 56
        ret     0
main    ENDP
```

Nous voyons que les 4 premiers arguments sont passés dans des registres, et les deux autres—par la pile.

La fonction `arith_mean()` place d'abord ces 4 arguments dans le *Shadow Space* puis traite le *Shadow Space* et la pile derrière comme s'il s'agissait d'un tableau continu!

Qu'en est-il de GCC? Les choses sont légèrement plus maladroites ici, car maintenant la fonction est divisée en deux parties: la première partie sauve les registres dans la « zone rouge », traite cet espace, et la seconde partie traite la pile:

Listing 3.59: GCC 4.9.1 x64 avec optimisation

```
arith_mean :
        lea     rax, [rsp+8]
        ; sauver les 6 registrers en entrée dans
        ; la red zone  sur la pile locale
        mov     QWORD PTR [rsp-40], rsi
        mov     QWORD PTR [rsp-32], rdx
        mov     QWORD PTR [rsp-16], r8
        mov     QWORD PTR [rsp-24], rcx
        mov     esi, 8
        mov     QWORD PTR [rsp-64], rax
        lea     rax, [rsp-48]
        mov     QWORD PTR [rsp-8], r9
        mov     DWORD PTR [rsp-72], 8
        lea     rdx, [rsp+8]
        mov     r8d, 1
        mov     QWORD PTR [rsp-56], rax
        jmp     .L5
.L7 :
        ; traiter les arguments sauvés
        lea     rax, [rsp-48]
        mov     ecx, esi
        add     esi, 8
        add     rcx, rax
        mov     ecx, DWORD PTR [rcx]
        cmp     ecx, -1
        je      .L4
.L8 :
        add     edi, ecx
        add     r8d, 1
.L5 :
        ; décider, quelle partie traiter maintenant.
        ; est-ce que le nombre d'arguments actuel est inférieur ou égal à 6?
```

```
        cmp     esi, 47
        jbe     .L7             ; non, traiter les arguments sauvegardés;
        ; traiter les arguments de la pile
        mov     rcx, rdx
        add     rdx, 8
        mov     ecx, DWORD PTR [rcx]
        cmp     ecx, -1
        jne     .L8
.L4 :
        mov     eax, edi
        cdq
        idiv    r8d
        ret

.LC1 :
        .string "%d\n"
main :
        sub     rsp, 8
        mov     edx, 7
        mov     esi, 2
        mov     edi, 1
        mov     r9d, -1
        mov     r8d, 15
        mov     ecx, 10
        xor     eax, eax
        call    arith_mean
        mov     esi, OFFSET FLAT :.LC1
        mov     edx, eax
        mov     edi, 1
        xor     eax, eax
        add     rsp, 8
        jmp     __printf_chk
```

À propos, un usage similaire du *Shadow Space* est aussi considéré ici: 4.1.8 on page 551.

Utilisation du pointeur sur le premier argument de la fonction

L'exemple peut être récrit sans la macro va_arg :

```
#include <stdio.h>

int arith_mean(int v, ...)
{
        int *i=&v;
        int sum=*i, count=1;
        i++;

        while(1)
        {
                if ((*i)==-1) // terminator
                        break;
                sum=sum+(*i);
                count++;
                i++;
        }

        return sum/count;
};

int main()
{
        printf ("%d\n", arith_mean (1, 2, 7, 10, 15, -1 /* terminator */));
        // test : https ://www.wolframalpha.com/input/?i=mean(1,2,7,10,15)
};
```

Autrement dit, si l'argument mis est un tableau de mots (32-bit ou 64-bit), nous devons juste énumérer les éléments du tableau en commençant par le premier.

3.14.2 Cas de la fonction *vprintf()*

De nombreux programmeurs définissent leur propre fonction de logging, qui prend une chaîne de format du type de celle de printf+une liste variable d'arguments.

Un autre exemple répandu est la fonction die(), qui affiche un message et sort.

Nous avons besoin d'un moyen de transmettre un nombre d'arguments inconnu et de les passer à la fonction `printf()`. Mais comment?

À l'aide des fonctions avec un « v » dans le nom.

Une d'entre elles est *vprintf()* : elle prend une chaîne de format et un pointeur sur une variable du type `va_list` :

```
#include <stdlib.h>
#include <stdarg.h>

void die (const char * fmt, ...)
{
        va_list va;
        va_start (va, fmt);

        vprintf (fmt, va);
        exit(0);
};
```

En examinant plus précisément, nous voyons que `va_list` est un pointeur sur un tableau. Compilons:

Listing 3.60: MSVC 2010 avec optimisation

```
_fmt$ = 8
_die    PROC
        ; charger le 1er argument (format-string)
        mov     ecx, DWORD PTR _fmt$[esp-4]
        ; obtenir un pointeur sur le 2nd argument
        lea     eax, DWORD PTR _fmt$[esp]
        push    eax             ; passer un pointeur
        push    ecx
        call    _vprintf
        add     esp, 8
        push    0
        call    _exit
$LN3@die :
        int     3
_die    ENDP
```

Nous voyons que tout ce que fait notre fonction est de prendre un pointeur sur les arguments et le passe à la fonction *vprintf()*, et que cette fonction le traite comme un tableau infini d'arguments!

Listing 3.61: MSVC 2012 x64 avec optimisation

```
fmt$ = 48
die     PROC
        ; sauver les 4 premiers arguments dans le Shadow Space
        mov     QWORD PTR [rsp+8], rcx
        mov     QWORD PTR [rsp+16], rdx
        mov     QWORD PTR [rsp+24], r8
        mov     QWORD PTR [rsp+32], r9
        sub     rsp, 40
        lea     rdx, QWORD PTR fmt$[rsp+8] ; passer un pointeur sur le 1er argument
        ; ici RCX pointe toujours sur le 1er argument (format-string) de die()
        ; donc vprintf() va prendre son argument dans RCX
        call    vprintf
        xor     ecx, ecx
        call    exit
        int     3
die     ENDP
```

3.14.3 Cas Pin

Il est intéressant de noter que certaines fonctions du framework DBI[15] Pin prennent un nombre variable d'arguments:

```
            INS_InsertPredicatedCall(
                ins, IPOINT_BEFORE, (AFUNPTR)RecordMemRead,
                IARG_INST_PTR,
                IARG_MEMORYOP_EA, memOp,
                IARG_END);
```

(pinatrace.cpp)

Et voici comment la fonction INS_InsertPredicatedCall() est déclarée:

```
extern VOID INS_InsertPredicatedCall(INS ins, IPOINT ipoint, AFUNPTR funptr, ...);
```

(pin_client.PH)

Ainsi, les constantes avec un nom débutant par IARG_ sont des sortes d'arguments pour la fonction, qui sont manipulés à l'intérieur de INS_InsertPredicatedCall(). Vous pouvez passer autant d'arguments que vous en avez besoin. Certaines commandes ont des arguments additionnels, d'autres non. Liste complète des arguments: https://software.intel.com/sites/landingpage/pintool/docs/58423/Pin/html/group__INST__ARGS.html. Et il faut un moyen pour détecter la fin de la liste des arguments, donc la liste doit être terminée par la constante IARG_END, sans laquelle, la fonction essayerait de traiter les données indéterminées dans la pile locale comme des arguments additionnels.

Aussi, dans [Brian W. Kernighan, Rob Pike, *Practice of Programming*, (1999)] nous pouvons trouver un bel exemple de routines C/C++ très similaires à *pack/unpack*[16] en Python.

3.14.4 Exploitation de chaîne de format

Il y a une erreur courante, celle d'écrire printf(string) au lieu de puts(string) ou printf("%s", string). Si l'attaquant peut mettre son propre texte dans string, il peut planter le processus ou accéder aux variables de la pile locale.

Regardons ceci:

```
#include <stdio.h>

int main()
{
        char *s1="hello";
        char *s2="world";
        char buf[128];

        // do something mundane here
        strcpy (buf, s1);
        strcpy (buf, " ");
        strcpy (buf, s2);

        printf ("%s");
};
```

Veuillez noter que printf() n'a pas d'argument supplémentaire autre que la chaîne de format.

Maintenant, imaginons que c'est l'attaquant qui a mis la chaîne %s dans le premier argument du dernier printf(). Je compile cet exemple en utilisant GCC 5.4.0 sous Ubuntu x86, et l'exécutable résultant affiche la chaîne « world » s'il est exécuté!

Si je compile avec l'optimisation, printf() affiche n'importe quoi, aussi—probablement, l'appel à strcpy() a été optimisé et/ou les variables locales également. De même, le résultat pour du code x64 sera différent, pour différents compilateurs, OS, etc.

Maintenant, disons que l'attaquant peut passer la chaîne suivante à l'appel de printf() : %x %x %x %x %x. Dans mon cas, la sortie est: « 80485c6 b7751b48 1 0 80485c0 » (ce sont simplement des valeurs

[15] Dynamic Binary Instrumentation
[16] https://docs.python.org/3/library/struct.html

de la pile locale). Vous voyez, il y a les valeurs 1 et 0, et des pointeurs (le premier est probablement un pointeur sur la chaîne « world »). Donc si l'attaquant passe la chaîne `%s %s %s %s %s`, le processus va se planter, car `printf()` traite 1 et/ou 0 comme des pointeurs sur une chaîne, essaye de lire des caractères et échoue.

Encore pire, il pourrait y avoir `sprintf (buf, string)` dans le code, où `buf` est un buffer dans la pile locale avec un taille de 1024 octets ou autre, l'attaquant pourrait préparer une chaîne de telle sorte que `buf` serait débordé, peut-être même de façon à conduire à l'exécution de code.

De nombreux logiciels bien connus et très utilisés étaient (ou sont encore) vulnérables:

> QuakeWorld went up, got to around 4000 users, then the master server exploded.
> (QuakeWorld est arrivé, monté à environ 4000 utilisateurs, puis le serveur master a explosé.)
> Disrupter and cohorts are working on more robust code now.
> (Les perturbateurs et cohortes travaillent maintenant sur un code plus robuste.)
> If anyone did it on purpose, how about letting us know... (It wasn't all the people that tried %s as a name)
> (Si quelqu'un l'a fait exprès, pourquoi ne pas nous le faire savoir... (Ce n'est pas tout le monde qui a essayé %s comme nom))

(John Carmack's .plan file, 17-Dec-1996[17])

De nos jours, tous les compilateurs dignes de ce nom avertissent à propos de ceci.

Un autre problème qui est moins connu, c'est l'argument `%n` de `printf()` : lorsque `printf()` l'atteint dans la chaîne de format, il écrit le nombre de caractères écrits jusqu'ici dans l'argument correspondant: http://stackoverflow.com/questions/3401156/what-is-the-use-of-the-n-format-specifier-in-c. Ainsi, un attaquant peut zapper les variables locales en passant plusieurs commandes `%n` dans la chaîne de format.

3.15 Ajustement de chaînes

Un traitement de chaîne très courant est la suppression de certains caractères au début et/où à la fin.

Dans cet exemple, nous allons travailler avec une fonction qui supprime tous les caractères newline (CR[18]/LF[19]) à la fin de la chaîne entrée:

```
#include <stdio.h>
#include <string.h>

char* str_trim (char *s)
{
        char c;
        size_t str_len;

        // fonctionne tant que \r ou \n se trouve en fin de la chaîne
        // s'arrête si un autre caractère s'y trouve ou si la chaîne est vide
        // (au début ou au cours de notre opération)
        for (str_len=strlen(s); str_len>0 && (c=s[str_len-1]); str_len--)
        {
                if (c=='\r' || c=='\n')
                        s[str_len-1]=0;
                else
                        break;
        };
        return s;
};

int main()
{
        // test
```

[17] https://github.com/ESWAT/john-carmack-plan-archive/blob/33ae52fdba46aa0d1abfed6fc7598233748541c0/by_day/johnc_plan_19961217.txt
[18] Carriage return (13 ou '\r' en C/C++)
[19] Line feed (10 ou '\n' en C/C++)

```
        // strdup() est utilisé pour copier du texte de chaîne dans le segment de données,
        // car autrement ça va crasher sur Linux,
        // où les chaîne de texte sont allouées dans le segment de données constantes,
        // et n'est pas modifiable.

        printf ("[%s]\n", str_trim (strdup("")));
        printf ("[%s]\n", str_trim (strdup("\n")));
        printf ("[%s]\n", str_trim (strdup("\r")));
        printf ("[%s]\n", str_trim (strdup("\n\r")));
        printf ("[%s]\n", str_trim (strdup("\r\n")));
        printf ("[%s]\n", str_trim (strdup("test1\r\n")));
        printf ("[%s]\n", str_trim (strdup("test2\n\r")));
        printf ("[%s]\n", str_trim (strdup("test3\n\r\n\r")));
        printf ("[%s]\n", str_trim (strdup("test4\n")));
        printf ("[%s]\n", str_trim (strdup("test5\r")));
        printf ("[%s]\n", str_trim (strdup("test6\r\r\r")));
};
```

L'argument en entrée est toujours renvoyé en sortie, ceci est pratique lorsque vous voulez chaîner les fonctions de traitement de chaîne, comme c'est fait ici dans la fonction `main()`.

La seconde partie de for() (`str_len>0 && (c=s[str_len-1])`) est appelé le « short-circuit » en C/C++ et est très pratique [Dennis Yurichev, *C/C++ programming language notes*1.3.8].

Les compilateurs C/C++ garantissent une séquence d'évaluation de gauche à droite.

Donc, si la première clause est fausse après l'évaluation, la seconde n'est pas évaluée.

3.15.1 x64: MSVC 2013 avec optimisation

Listing 3.62: MSVC 2013 x64 avec optimisation

```
s$ = 8
str_trim PROC

; RCX est le premier argument de la fonction et il contient toujours un pointeur sur la chaîne
        mov     rdx, rcx
; ceci est la fonction strlen() inlined juste ici :
; mettre RAX à 0xFFFFFFFFFFFFFFFF (-1)
        or      rax, -1
$LL14@str_trim :
        inc     rax
        cmp     BYTE PTR [rcx+rax], 0
        jne     SHORT $LL14@str_trim
; est-ce que la chaîne en entrée est de longueur zéro? alors sortir :
        test    rax, rax
        je      SHORT $LN15@str_trim
; RAX contient la longueur de la chaîne
        dec     rcx
; RCX = s-1
        mov     r8d, 1
        add     rcx, rax
; RCX = s-1+strlen(s), i.e., ceci est l'adresse du dernier caractère de la chaîne
        sub     r8, rdx
; R8 = 1-s
$LL6@str_trim :
; charger le dernier caractère de la chaîne :
; sauter si son code est 13 ou 10:
        movzx   eax, BYTE PTR [rcx]
        cmp     al, 13
        je      SHORT $LN2@str_trim
        cmp     al, 10
        jne     SHORT $LN15@str_trim
$LN2@str_trim :
; le dernier caractère a un code de 13 ou 10
; écrire zéro à cet endroit :
        mov     BYTE PTR [rcx], 0
; décrémenter l'adresse du dernier caractère,
; donc il pointera sur le caractère précédent celui qui vient d'être effacé :
```

```
        dec     rcx
        lea     rax, QWORD PTR [r8+rcx]
; RAX = 1 - s + adresse du dernier caractère courant
; ainsi nous pouvons déterminer si nous avons atteint le premier caractère et
; nous devons arrêter, si c'est le cas
        test    rax, rax
        jne     SHORT $LL6@str_trim
$LN15@str_trim :
        mov     rax, rdx
        ret     0
str_trim ENDP
```

Tout d'abord, MSVC a inliné le code la fonction `strlen()`, car il en a conclus que ceci était plus rapide que le `strlen()` habituel + le coût de l'appel et du retour. Ceci est appelé de l'inlining: 3.11 on page 511.

La première instruction de `strlen()` mis en ligne est `OR RAX, 0xFFFFFFFFFFFFFFFF`.

MSVC utilise souvent `OR` au lieu de `MOV RAX, 0xFFFFFFFFFFFFFFFF`, car l'opcode résultant est plus court.

Et bien sûr, c'est équivalent: tous les bits sont mis à 1, et un nombre avec tous les bits mis vaut -1 en complément à 2: 2.2 on page 456.

On peut se demander pourquoi le nombre -1 est utilisé dans `strlen()`. À des fins d'optimisation, bien sûr. Voici le code que MSVC a généré:

Listing 3.63: Inlined `strlen()` by MSVC 2013 x64

```
; RCX = pointeur sur la chaîne en entrée
; RAX = longueur actuelle de la chaîne
        or      rax, -1
label :
        inc     rax
        cmp     BYTE PTR [rcx+rax], 0
        jne     SHORT label
; RAX = longueur de la chaîne
```

Essayez d'écrite plus court si vous voulez initialiser le compteur à 0! OK, essayons:

Listing 3.64: Our version of `strlen()`

```
; RCX = pointeur sur la chaîne en entrée
; RAX = longueur actuelle de la chaîne
        xor     rax, rax
label :
        cmp     byte ptr [rcx+rax], 0
        jz      exit
        inc     rax
        jmp     label
exit :
; RAX = longueur de la chaîne
```

Nous avons échoué. Nous devons utilisé une instruction `JMP` additionnelle!

Donc, ce que le compilateur de MSVC 2013 a fait, c'est de déplacer l'instruction `INC` avant le chargement du caractère courant.

Si le premier caractère est 0, c'est OK, `RAX` contient 0 à ce moment, donc la longueur de la chaîne est 0.

Le reste de cette fonction semble facile à comprendre.

3.15.2 x64: GCC 4.9.1 sans optimisation

```
str_trim :
        push    rbp
        mov     rbp, rsp
        sub     rsp, 32
        mov     QWORD PTR [rbp-24], rdi
; la première partie de for() commence ici
        mov     rax, QWORD PTR [rbp-24]
```

```
        mov     rdi, rax
        call    strlen
        mov     QWORD PTR [rbp-8], rax   ; str_len
; la première partie de for() se termine ici
        jmp     .L2
; le corps de for() commence ici
.L5 :
        cmp     BYTE PTR [rbp-9], 13     ; c=='\r'?
        je      .L3
        cmp     BYTE PTR [rbp-9], 10     ; c=='\n'?
        jne     .L4
.L3 :
        mov     rax, QWORD PTR [rbp-8]   ; str_len
        lea     rdx, [rax-1]             ; EDX=str_len-1
        mov     rax, QWORD PTR [rbp-24]  ; s
        add     rax, rdx                 ; RAX=s+str_len-1
        mov     BYTE PTR [rax], 0        ; s[str_len-1]=0
; le corps de for() se termine ici
; la troisième partie de for() commence ici
        sub     QWORD PTR [rbp-8], 1     ; str_len--
; la troisième partie de for() se termine ici
.L2 :
; la deuxième partie de for() commence ici
        cmp     QWORD PTR [rbp-8], 0     ; str_len==0?
        je      .L4                      ; alors sortir
; tester la seconde clause, et charger "c"
        mov     rax, QWORD PTR [rbp-8]   ; RAX=str_len
        lea     rdx, [rax-1]             ; RDX=str_len-1
        mov     rax, QWORD PTR [rbp-24]  ; RAX=s
        add     rax, rdx                 ; RAX=s+str_len-1
        movzx   eax, BYTE PTR [rax]  ; AL=s[str_len-1]
        mov     BYTE PTR [rbp-9], al     ; stocker l caractère chargé dans "c"
        cmp     BYTE PTR [rbp-9], 0      ; est-ce zéro?
        jne     .L5                      ; oui? alors sortir
; la deuxième partie de for() se termine ici
.L4 :
; renvoyer "s"
        mov     rax, QWORD PTR [rbp-24]
        leave
        ret
```

Les commentaires ont été ajoutés par l'auteur du livre.

Après l'exécution de `strlen()`, le contrôle est passé au label L2, et ici deux clauses sont vérifiées, l'une après l'autre.

La seconde ne sera jamais vérifiée, si la première (*str_len==0*) est fausse (ceci est un « short-circuit » (court-circuit)).

Maintenant regardons la forme courte de cette fonction:

- Première partie de for() (appel à `strlen()`)
- goto L2
- L5: corps de for(). sauter à la fin, si besoin
- troisième partie de for() (décrémenter str_len)
- L2: deuxième partie de for(): vérifier la première clause, puis la seconde. sauter au début du corps de la boucle ou sortir.
- L4: // sortir
- renvoyer s

3.15.3 x64: GCC 4.9.1 avec optimisation

```
str_trim :
        push    rbx
        mov     rbx, rdi
```

```
; RBX sera toujours "s"
        call    strlen
; tester si str_len==0 et sortir si c'est la cas
        test    rax, rax
        je      .L9
        lea     rdx, [rax-1]
; RDX contiendra toujours la valeur str_len-1, pas str_len
; donc RDX est plutôt comme une variable sur l'index du buffer
        lea     rsi, [rbx+rdx]      ; RSI=s+str_len-1
        movzx   ecx, BYTE PTR [rsi] ; charger le caractère
        test    cl, cl
        je      .L9                 ; sortir si c'est zéro
        cmp     cl, 10
        je      .L4
        cmp     cl, 13              ; sortir si ce n'est ni '\n' ni '\r'
        jne     .L9
.L4 :
; ceci est une instruction bizarre, nous voulons RSI=s-1 ici.
; c'est possible de l'obtenir avec MOV RSI, EBX / DEC RSI
; mais ce sont deux instructions au lieu d'une
        sub     rsi, rax
; RSI = s+str_len-1-str_len = s-1
; la boucle principale commence
.L12 :
        test    rdx, rdx
; stocker zéro à l'adresse s-1+str_len-1+1 = s-1+str_len = s+str_len-1
        mov     BYTE PTR [rsi+1+rdx], 0
; tester si str_len-1==0. sortir si oui.
        je      .L9
        sub     rdx, 1                  ; équivalent à str_len--
; charger le caractère suivant à l'adresse s+str_len-1
        movzx   ecx, BYTE PTR [rbx+rdx]
        test    cl, cl                  ; est-ce zéro? sortir si oui
        je      .L9
        cmp     cl, 10                  ; est-ce '\n'?
        je      .L12
        cmp     cl, 13                  ; est-ce '\r'?
        je      .L12
.L9 :
; renvoyer "s"
        mov     rax, rbx
        pop     rbx
        ret
```

Maintenant, c'est plus complexe.

Le code avant le début du corps de la boucle est exécuté une seule fois, mais il contient le test des caractères CR/LF aussi! À quoi sert cette duplication du code?

La façon courante d'implémenter la boucle principale est sans doute ceci:

- (début de la boucle) tester la présence des caractères CR/LF, décider
- stocker le caractère zéro

Mais GCC a décidé d'inverser ces deux étapes.

Bien sûr, *stocker le caractère zéro* ne peut pas être la première étape, donc un autre test est nécessaire:

- traiter le premier caractère. matcher avec CR/LF, sortir si le caractère n'est pas CR/LF
- (début de la boucle) stocker le caractère zéro
- tester la présence des caractères CR/LF, décider

Maintenant la boucle principale est très courte, ce qui est bon pour les derniers CPUs.

Le code n'utilise pas la variable str_len, mais str_len-1. Donc c'est plus comme un index dans un buffer.

Apparemment, GCC a remarqué que l'expression str_len-1 est utilisée deux fois.

Donc, c'est mieux d'allouer une variable qui contient toujours une valeur qui est plus petite que la longueur actuelle de la chaîne de un, et la décrémente (ceci a le même effet que de décrémenter la variable str_len).

3.15.4 ARM64: GCC (Linaro) 4.9 sans optimisation

Cette implémentation est simple:

Listing 3.65: GCC (Linaro) 4.9 sans optimisation

```
str_trim :
        stp     x29, x30, [sp, -48]!
        add     x29, sp, 0
        str     x0, [x29,24] ; copier l'argument en entrée dans la pile locale
        ldr     x0, [x29,24] ; s
        bl      strlen
        str     x0, [x29,40] ; la variable str_len est dans la pile locale
        b       .L2
; la boucle principale commence
.L5 :
        ldrb    w0, [x29,39]
; W0=c
        cmp     w0, 13       ; est-ce '\r'?
        beq     .L3
        ldrb    w0, [x29,39]
; W0=c
        cmp     w0, 10       ; est-ce '\n'?
        bne     .L4          ; sauter à la sortie si non
.L3 :
        ldr     x0, [x29,40]
; X0=str_len
        sub     x0, x0, #1
; X0=str_len-1
        ldr     x1, [x29,24]
; X1=s
        add     x0, x1, x0
; X0=s+str_len-1
        strb    wzr, [x0]    ; écrire l'octet à s+str_len-1
; décrémenter str_len :
        ldr     x0, [x29,40]
; X0=str_len
        sub     x0, x0, #1
; X0=str_len-1
        str     x0, [x29,40]
; sauver X0 (or str_len-1) dans la pile locale
.L2 :
        ldr     x0, [x29,40]
; str_len==0?
        cmp     x0, xzr
; sauter alors à la sortie
        beq     .L4
        ldr     x0, [x29,40]
; X0=str_len
        sub     x0, x0, #1
; X0=str_len-1
        ldr     x1, [x29,24]
; X1=s
        add     x0, x1, x0
; X0=s+str_len-1
; charger l'octet à l'adresse s+str_len-1 dans W0
        ldrb    w0, [x0]
        strb    w0, [x29,39] ; stocker l'octet chargé dans "c"
        ldrb    w0, [x29,39] ; le recharger
; est-ce l'octet zéro?
        cmp     w0, wzr
; sauter à la sortie, si c'est zéro ou en L5 sinon
        bne     .L5
.L4 :
; renvoyer s
        ldr     x0, [x29,24]
        ldp     x29, x30, [sp], 48
        ret
```

3.15.5 ARM64: GCC (Linaro) 4.9 avec optimisation

Ceci est une optimisation plus avancée.

Le premier caractère est chargé au début, et comparé avec 10 (le caractère LF).

Les caractères sont ensuite chargés dans la boucle principale, pour les caractères après le premier.

Ceci est quelque peu similaire à l'exemple 3.15.3 on page 533.

Listing 3.66: GCC (Linaro) 4.9 avec optimisation

```
str_trim :
        stp     x29, x30, [sp, -32]!
        add     x29, sp, 0
        str     x19, [sp,16]
        mov     x19, x0
; X19 contiendra toujours la valeur de "s"
        bl      strlen
; X0=str_len
        cbz     x0, .L9         ; sauter en L9 (sortir) si str_len==0
        sub     x1, x0, #1
; X1=X0-1=str_len-1
        add     x3, x19, x1
; X3=X19+X1=s+str_len-1
        ldrb    w2, [x19,x1]    ; charger l'octet à l'adresse X19+X1=s+str_len-1
; W2=octet chargé
        cbz     w2, .L9         ; est-ce zéro? sauter alors à la sortie
        cmp     w2, 10          ; est-ce '\n'?
        bne     .L15
.L12 :
; corps de la boucle principale. Le caractère chargé est toujours 10 ou 13 à ce moment!
        sub     x2, x1, x0
; X2=X1-X0=str_len-1-str_len=-1
        add     x2, x3, x2
; X2=X3+X2=s+str_len-1+(-1)=s+str_len-2
        strb    wzr, [x2,1]     ; stocker l'octet zéro à l'adresse s+str_len-2+1=s+str_len-1
        cbz     x1, .L9         ; str_len-1==0? sauter à la sortie si oui
        sub     x1, x1, #1      ; str_len--
        ldrb    w2, [x19,x1]    ; charger le caractère suivant à l'adresse X19+X1=s+str_len-1
        cmp     w2, 10          ; est-ce '\n'?
        cbz     w2, .L9         ; sauter à la sortie, si c'est zéro
        beq     .L12            ; sauter au début du corps de la boucle, si c'est '\n'
.L15 :
        cmp     w2, 13          ; est-ce '\r'?
        beq     .L12            ; oui, sauter au début du corps de la boucle
.L9 :
; renvoyer "s"
        mov     x0, x19
        ldr     x19, [sp,16]
        ldp     x29, x30, [sp], 32
        ret
```

3.15.6 ARM: avec optimisation Keil 6/2013 (Mode ARM)

À nouveau, le compilateur tire partie des instructions conditionnelles du mode ARM, donc le code est bien plus compact.

Listing 3.67: avec optimisation Keil 6/2013 (Mode ARM)

```
str_trim PROC
        PUSH    {r4,lr}
; R0=s
        MOV     r4,r0
; R4=s
        BL      strlen          ; strlen() prend la valeur de "s" dans R0
; R0=str_len
        MOV     r3,#0
; R3 contiendra toujours 0
|L0.16|
```

```
        CMP      r0,#0         ; str_len==0?
        ADDNE    r2,r4,r0      ; (si str_len!=0) R2=R4+R0=s+str_len
        LDRBNE   r1,[r2,#-1]   ; (si str_len!=0) R1=charger l'octet à l'adresse R2-1=s+str_len↙
    ↳ -1
        CMPNE    r1,#0         ; (si str_len!=0) comparer l'octet chargé avec 0
        BEQ      |L0.56|       ; sauter à la sortie si str_len==0 ou si l'octet chargé est 0
        CMP      r1,#0xd       ; est-ce que l'octet chargé est '\r'?
        CMPNE    r1,#0xa       ; (si l'octet chargé n'est pas '\r') est-ce '\r'?
        SUBEQ    r0,r0,#1      ; (si l'octet chargé  est '\r' ou '\n') R0-- ou str_len--
        STRBEQ   r3,[r2,#-1]   ; (si l'octet chargé est '\r' ou '\n') stocker R3 (zéero)
    à l'adresse R2-1=s+str_len-1
        BEQ      |L0.16|       ; sauter au début de a boucle si l'octet chargé
    était '\r' ou '\n'
|L0.56|
; renvoyer "s"
        MOV      r0,r4
        POP      {r4,pc}
        ENDP
```

3.15.7 ARM: avec optimisation Keil 6/2013 (Mode Thumb)

Il y a moins d'instructions conditionnelles en mode Thumb, donc le code est plus simple.

Mais il y a des choses vraiment étranges aux offsets 0x20 et 0x1F (lignes 22 et 23). Pourquoi diable le compilateur Keil a-t-il fait ça? Honnêtement, c'est difficile de le dire.

Ça doit être une bizarrerie du processus d'optimisation de Keil. Néanmoins, le code fonctionne correctement

Listing 3.68: avec optimisation Keil 6/2013 (Mode Thumb)

```
 1 str_trim PROC
 2         PUSH     {r4,lr}
 3         MOVS     r4,r0
 4 ; R4=s
 5         BL       strlen        ; strlen() prend la valeur de "s" dans R0
 6 ; R0=str_len
 7         MOVS     r3,#0
 8 ; R3 contiendra toujours 0
 9         B        |L0.24|
10 |L0.12|
11         CMP      r1,#0xd       ; est-ce que l'octet chargé est '\r'?
12         BEQ      |L0.20|
13         CMP      r1,#0xa       ; est-ce que l'octet chargé est '\n'?
14         BNE      |L0.38|       ; sauter à la sortie si non
15 |L0.20|
16         SUBS     r0,r0,#1      ; R0-- ou str_len--
17         STRB     r3,[r2,#0x1f] ; stocker 0 à l'adresse R2+0x1F=s+str_len-0x20+0x1F=s+str_len-1
18 |L0.24|
19         CMP      r0,#0         ; str_len==0?
20         BEQ      |L0.38|       ; oui? sauter à la sortie
21         ADDS     r2,r4,r0      ; R2=R4+R0=s+str_len
22         SUBS     r2,r2,#0x20   ; R2=R2-0x20=s+str_len-0x20
23         LDRB     r1,[r2,#0x1f] ; charger l'octet à l'adresse R2+0x1F=s+str_len-0x20+0x1F=s+↙
       ↳ str_len-1 dans R1
24         CMP      r1,#0         ; est-ce que l'octet chargé est 0?
25         BNE      |L0.12|       ; sauter au début de la boucle, si ce n'est pas 0
26 |L0.38|
27 ; renvoyer "s"
28         MOVS     r0,r4
29         POP      {r4,pc}
30         ENDP
```

3.15.8 MIPS

Listing 3.69: GCC 4.4.5 avec optimisation (IDA)

```
str_trim :
; IDA n'a pas connaissance des noms des variables locales, nous les entrons manuellement
saved_GP          = -0x10
saved_S0           = -8
saved_RA           = -4

                lui     $gp, (__gnu_local_gp >> 16)
                addiu   $sp, -0x20
                la      $gp, (__gnu_local_gp & 0xFFFF)
                sw      $ra, 0x20+saved_RA($sp)
                sw      $s0, 0x20+saved_S0($sp)
                sw      $gp, 0x20+saved_GP($sp)
; appeler strlen(). l'adresse de la chaîne en entrée est toujours dans $a0,
; strlen() la prendra d'ici :
                lw      $t9, (strlen & 0xFFFF)($gp)
                or      $at, $zero  ; slot de délai de chargement, NOP
                jalr    $t9
; l'adresse de la chaîne en entrée est toujours dans $a0, mettons la dans $s0 :
                move    $s0, $a0    ; slot de délai de branchement
; le résultat de strlen() (i.e., la longueur de la chaîne) est maintenant dans $v0
; sauter à la sortie si $v0==0 (i.e., la longueur de la chaîne est 0) :
                beqz    $v0, exit
                or      $at, $zero  ; slot de délai de branchement, NOP
                addiu   $a1, $v0, -1
; $a1 = $v0-1 = str_len-1
                addu    $a1, $s0, $a1
; $a1 = adresse de la chaîne en entrée + $a1 = s+strlen-1
; charger l'octet à l'adresse $a1 :
                lb      $a0, 0($a1)
                or      $at, $zero  ; slot de délai de chargement, NOP
; est-ce que l'octet est zéro? sauter à la sortie si oui :
                beqz    $a0, exit
                or      $at, $zero  ; slot de délai de branchement, NOP
                addiu   $v1, $v0, -2
; $v1 = str_len-2
                addu    $v1, $s0, $v1
; $v1 = $s0+$v1 = s+str_len-2
                li      $a2, 0xD
; sauter le corps de boucle :
                b       loc_6C
                li      $a3, 0xA    ; slot de délai de branchement
loc_5C :
; charger l'octet suivant de la mémoire dans $a0 :
                lb      $a0, 0($v1)
                move    $a1, $v1
; $a1=s+str_len-2
; sauter à la sortie si l'octet chargé est zéro :
                beqz    $a0, exit
; décrémenter str_len :
                addiu   $v1, -1     ; slot de délai de branchement
loc_6C :
; à ce moment, $a0=octet chargé,  $a2=0xD (symbole CR) et $a3=0xA (symbole LF)
; l'octet chargé est CR? sauter alors en loc_7C :
                beq     $a0, $a2, loc_7C
                addiu   $v0, -1     ; slot de délai de branchement
; l'octet chargé est LF? sauter à la sortie si ce n'est pas LF :
                bne     $a0, $a3, exit
                or      $at, $zero  ; slot de délai de branchement, NOP
loc_7C :
; l'octet chargé est CR à ce moment
; sauter en loc_5c (début du corps de la boucle) si str_len (dans $v0) n'est pas zéro :
                bnez    $v0, loc_5C
; simultanément, stocker zéro à cet endroit en mémoire :
                sb      $zero, 0($a1) ; slot de délai de branchement
; le label "exit" à été renseigné manuellemnt :
exit :
                lw      $ra, 0x20+saved_RA($sp)
                move    $v0, $s0
                lw      $s0, 0x20+saved_S0($sp)
                jr      $ra
```

```
                addiu   $sp, 0x20     ; slot de délai de branchement
```

Les registres préfixés avec S- sont aussi appelés « saved temporaries » (sauvé temporairement), donc la valeur de $S0 est sauvée dans la pile locale et restaurée à la fin.

3.16 Fonction toupper()

Une autre fonction courante transforme un symbole de minuscule en majuscule, si besoin:

```
char toupper (char c)
{
        if(c>='a' && c<='z')
                return c-'a'+'A';
        else
                return c;
}
```

L'expression `'a'+'A'` est laissée dans le code source pour améliorer la lisibilité, elle sera optimisée par le compilateur, bien sûr. [20].

Le code ASCII de « a » est 97 (ou 0x61), et celui de « A », 65 (ou 0x41).

La différence (ou distance) entre les deux dans la table ASCII est 32 (ou 0x20).

Pour une meilleure compréhension, le lecteur peut regarder la table ASCII 7-bit standard:

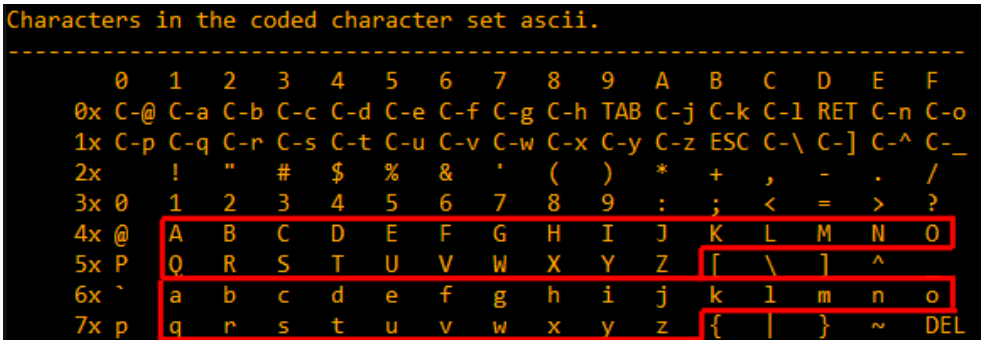


Fig. 3.3: table ASCII 7-bit dans Emacs

3.16.1 x64

Deux opérations de comparaison

MSVC sans optimisation est direct: le code vérifie si le symbole en entrée est dans l'intervalle [97..122] (ou dans l'intervalle ['a'..'z']) et soustrait 32 si c'est le cas.

Il y a quelques artefacts du compilateur:

Listing 3.70: MSVC 2013 (x64) sans optimisation

```
1  c$ = 8
2  toupper PROC
3          mov     BYTE PTR [rsp+8], cl
4          movsx   eax, BYTE PTR c$[rsp]
5          cmp     eax, 97
6          jl      SHORT $LN2@toupper
7          movsx   eax, BYTE PTR c$[rsp]
8          cmp     eax, 122
9          jg      SHORT $LN2@toupper
10         movsx   eax, BYTE PTR c$[rsp]
11         sub     eax, 32
12         jmp     SHORT $LN3@toupper
```

[20] Toutefois, pour être méticuleux, il y a toujours des compilateurs qui ne peuvent pas optimiser de telles expressions et les laissent telles quelles dans le code.

```
13          jmp     SHORT $LN1@toupper      ; artefact du compilateur
14  $LN2@toupper :
15          movzx   eax, BYTE PTR c$[rsp]   ; casting inutile
16  $LN1@toupper :
17  $LN3@toupper :                           ; artefact du compilateur
18          ret     0
19  toupper ENDP
```

Il est important de remarquer que l'octet en entrée est chargé dans un slot 64-bit de la pile locale à la ligne 3.

Tous les bits restants ([8..e3]) ne sont pas touchés, i.e., contiennent du bruit indéterminé (vous le verrez dans le débogueur).

Toutes les instructions opèrent seulement au niveau de l'octet, donc c'est bon.

La dernière instruction `MOVZX` à la ligne 15 prend un octet de la pile locale et l'étend avec des zéro à un type de donnée *int* 32-bit.

GCC sans optimisation fait essentiellement la même chose:

Listing 3.71: GCC 4.9 (x64) sans optimisation

```
toupper :
        push    rbp
        mov     rbp, rsp
        mov     eax, edi
        mov     BYTE PTR [rbp-4], al
        cmp     BYTE PTR [rbp-4], 96
        jle     .L2
        cmp     BYTE PTR [rbp-4], 122
        jg      .L2
        movzx   eax, BYTE PTR [rbp-4]
        sub     eax, 32
        jmp     .L3
.L2 :
        movzx   eax, BYTE PTR [rbp-4]
.L3 :
        pop     rbp
        ret
```

Une opération de comparaison

MSVC avec optimisation fait un meilleur travail, il ne génère qu'une seule opération de comparaison:

Listing 3.72: MSVC 2013 (x64) avec optimisation

```
toupper PROC
        lea     eax, DWORD PTR [rcx-97]
        cmp     al, 25
        ja      SHORT $LN2@toupper
        movsx   eax, cl
        sub     eax, 32
        ret     0
$LN2@toupper :
        movzx   eax, cl
        ret     0
toupper ENDP
```

Il a déjà été expliqué comment remplacer les deux opérations de comparaison par une seule: 3.10.2 on page 510.

Nous allons maintenant récrire ceci en C/C++ :

```
int tmp=c-97;

if (tmp>25)
        return c;
else
        return c-32;
```

La variable *tmp* doit être signée.

Cela fait deux opérations de soustraction en cas de transformation plus une comparaison.

Par contraste, l'algorithme original utilise deux opérations de comparaison plus une soustraction.

GCC avec optimisation est encore meilleur, il supprime le saut (ce qui est bien: 2.10.1 on page 470) en utilisant l'instruction CMOVcc:

Listing 3.73: GCC 4.9 (x64) avec optimisation

```
1 toupper :
2         lea     edx, [rdi-97] ; 0x61
3         lea     eax, [rdi-32] ; 0x20
4         cmp     dl, 25
5         cmova   eax, edi
6         ret
```

À la ligne 3 le code prépare la valeur soustraite en avance, comme si la conversion avait toujours lieu.

À la ligne 5 la valeur soustraite dans EAX est remplacée par la valeur en entrée non modifiée si la conversion n'est pas nécessaire. Et ensuite cette valeur (bien sûr incorrecte) est abandonnée.

La soustraction en avance est le prix que le compilateur paye pour l'absence de saut conditionnel.

3.16.2 ARM

Keil avec optimisation pour le mode ARM génère aussi une seule comparaison:

Listing 3.74: avec optimisation Keil 6/2013 (Mode ARM)

```
toupper PROC
        SUB      r1,r0,#0x61
        CMP      r1,#0x19
        SUBLS    r0,r0,#0x20
        ANDLS    r0,r0,#0xff
        BX       lr
        ENDP
```

Les instructions SUBLS et ANDLS ne sont exécutées que si la valeur dans R1 est inférieure à 0x19 (ou égale).

Keil avec optimisation pour le mode Thumb génère lui aussi une seule opération de comparaison:

Listing 3.75: avec optimisation Keil 6/2013 (Mode Thumb)

```
toupper PROC
        MOVS     r1,r0
        SUBS     r1,r1,#0x61
        CMP      r1,#0x19
        BHI      |L0.14|
        SUBS     r0,r0,#0x20
        LSLS     r0,r0,#24
        LSRS     r0,r0,#24
|L0.14|
        BX       lr
        ENDP
```

Les deux dernières instructions LSLS et LSRS fonctionnent comme `AND reg, 0xFF` : elles sont équivalentes à l'expression C/C++ $(i << 24) >> 24$.

Il semble que Keil pour le mode Thumb déduit que ces deux instructions de 2-octets sont plus courtes que le code qui charge la constante 0xFF dans un registre plus une instruction AND.

GCC pour ARM64

Listing 3.76: GCC 4.9 (ARM64) sans optimisation

```
toupper :
        sub     sp, sp, #16
        strb    w0, [sp,15]
        ldrb    w0, [sp,15]
        cmp     w0, 96
        bls     .L2
        ldrb    w0, [sp,15]
        cmp     w0, 122
        bhi     .L2
        ldrb    w0, [sp,15]
        sub     w0, w0, #32
        uxtb    w0, w0
        b       .L3
.L2 :
        ldrb    w0, [sp,15]
.L3 :
        add     sp, sp, 16
        ret
```

Listing 3.77: GCC 4.9 (ARM64) avec optimisation

```
toupper :
        uxtb    w0, w0
        sub     w1, w0, #97
        uxtb    w1, w1
        cmp     w1, 25
        bhi     .L2
        sub     w0, w0, #32
        uxtb    w0, w0
.L2 :
        ret
```

3.16.3 Utilisation d'opérations sur les bits

Étant donné le fait que le bit d'indice 5 (en partant depuis 0) est toujours présent après le test, soustraire revient juste à effacer ce seul bit, mais la même chose peut être effectuée avec un AND (2.5 on page 462).

Encore plus simple, en XOR-ant:

```
char toupper (char c)
{
        if(c>='a' && c<='z')
                return c^0x20;
        else
                return c;
}
```

Le code est proche de ce GCC avec optimisation a produit pour l'exemple précédent (3.73 on the previous page):

Listing 3.78: GCC 5.4 (x86) avec optimisation

```
toupper :
        mov     edx, DWORD PTR [esp+4]
        lea     ecx, [edx-97]
        mov     eax, edx
        xor     eax, 32
        cmp     cl, 25
        cmova   eax, edx
        ret
```

...mais XOR est utilisé au lieu de SUB.

Changer le bit d'indice 5 est juste déplacer un *curseur* dans la table ASCII en haut ou en bas de deux lignes.

Certains disent que les lettres minuscules/majuscules ont été placées de cette façon dans la table ASCII intentionnellement, car:

> Very old keyboards used to do Shift just by toggling the 32 or 16 bit, depending on the key; this is why the relationship between small and capital letters in ASCII is so regular, and the relationship between numbers and symbols, and some pairs of symbols, is sort of regular if you squint at it.

(Eric S. Raymond, http://www.catb.org/esr/faqs/things-every-hacker-once-knew/)

Donc, nous pouvons écrire ce morceau de code, qui change juste la casse des lettres:

```
#include <stdio.h>

char flip (char c)
{
        if((c>='a' && c<='z') || (c>='A' && c<='Z'))
                return c^0x20;
        else
                return c;
}

int main()
{
        // affichera "hELLO, WORLD!"
        for (char *s="Hello, world!"; *s; s++)
                printf ("%c", flip(*s));
};
```

3.16.4 Summary

Toutes ces optimisations de compilateurs sont aujourd'hui courantes et un rétro-ingénieur pratiquant voit souvent ce genre de patterns de code.

Chapitre 4

Spécifique aux OS

4.1 Méthodes de transmission d'arguments (calling conventions)

4.1.1 cdecl

Il s'agit de la méthode la plus courante pour passer des arguments à une fonction en langage C/C++.

Après que le callee ait rendu la main, le caller doit ajuster la valeur du stack pointer (ESP) pour lui redonner la valeur qu'elle avait avant l'appel du callee.

Listing 4.1: cdecl

```
push arg3
push arg2
push arg1
call function
add esp, 12 ; returns ESP
```

4.1.2 stdcall

Similaire à *cdecl*, sauf que c'est le callee qui doit réinitialise ESP à sa valeur d'origine en utilisant l'instruction RET x et non pas RET,
avec `x = nb arguments * sizeof(int)`[1]. Après le retour du callee, le caller ne modifie pas le stack pointer. Il ny' a donc pas d'instruction `add esp, x`.

Listing 4.2: stdcall

```
push arg3
push arg2
push arg1
call function

function :
... do something ...
ret 12
```

Cette méthode est omniprésente dans les librairies standard win32, mais pas dans celles win64 (voir ci-dessous pour win64).

Prenons par exemple la fonction 1.84 on page 97 et modifions la légèrement en utilisant la convention `__stdcall` :

```
int __stdcall f2 (int a, int b, int c)
{
        return a*b+c;
};
```

[1] La taille d'une variable de type *int* est de 4 octets sur les systèmes x86 et de 8 octets sur les systèmes x64

Le résultat de la compilation sera quasiment identique à celui de 1.85 on page 97, mais vous constatez la présence de RET 12 au lieu de RET. Le caller ne met pas à jour SP après l'appel.

Avec la convention __stdcall, on peut facilement déduire le nombre d'arguments de la fonction appelée à partir de l'instruction RETN n : divisez n par 4.

Listing 4.3: MSVC 2010

```
_a$ = 8         ; size = 4
_b$ = 12        ; size = 4
_c$ = 16        ; size = 4
_f2@12  PROC
        push    ebp
        mov     ebp, esp
        mov     eax, DWORD PTR _a$[ebp]
        imul    eax, DWORD PTR _b$[ebp]
        add     eax, DWORD PTR _c$[ebp]
        pop     ebp
        ret     12
_f2@12  ENDP

; ...
        push    3
        push    2
        push    1
        call    _f2@12
        push    eax
        push    OFFSET $SG81369
        call    _printf
        add     esp, 8
```

Fonctions à nombre d'arguments variables

Les fonctions du style printf() sont un des rares cas de fonctions à nombre d'arguments variables en C/C++. Elles permettent d'illustrer une différence importante entre les conventions *cdecl* et *stdcall*. Partons du principe que le compilateur connait le nombre d'arguments utilisés à chaque fois que la fonction printf() est appelée.

En revanche, la fonction printf() est déjà compilée dans MSVCRT.DLL (si l'on parle de Windows) et ne possède aucune information sur le nombre d'arguments qu'elle va recevoir. Elle peut cependant le deviner à partir du contenu du paramètre format.

Si la convention *stdcall* était utilisé pour la fonction printf(), elle devrait réajuster le stack pointer à sa valeur initiale en comptant le nombre d'arguments dans la chaîne de format. Cette situation serait dangereuse et pourrait provoquer un crash du programme à la moindre faute de frappe du programmeur. La convention *stdcall* n'est donc pas adaptée à ce type de fonction. La convention *cdecl* est préférable.

4.1.3 fastcall

Il s'agit d'un nom générique pour désigner les conventions qui passent une partie des paramètres dans des registres et le reste sur la pile. Historiquement, cette méthode était plus performante que les conventions *cdecl*/*stdcall* - car elle met moins de pression sur la pile. Ce n'est cependant probablement plus le cas sur les processeurs actuels qui sont beaucoup plus complexes.

Cette convention n'est pas standardisée. Les compilateurs peuvent donc l'implémenter à leur guise. Prenez une DLL qui en utilise une autre compilée avec une interprétation différente de la convention *fastcall*. Vous avez un cas d'école et des problèmes en perspective.

Les compilateurs MSVC et GCC passent les deux premiers arguments dans ECX et EDX, et le reste des arguments sur la pile.

Le stack pointer doit être restauré à sa valeur initiale par le callee (comme pour la convention *stdcall*).

Listing 4.4: fastcall

```
push arg3
mov edx, arg2
mov ecx, arg1
```

```
call function

function :
.. do something ..
ret 4
```

Prenons par exemple la fonction 1.84 on page 97 et modifions la légèrement en utilisant la convention `__fastcall` :

```
int __fastcall f3 (int a, int b, int c)
{
        return a*b+c;
};
```

Le résultat de la compilation est le suivant:

Listing 4.5: MSVC 2010 optimisé/Ob0

```
_c$ = 8         ; size = 4
@f3@12  PROC
; _a$ = ecx
; _b$ = edx
        mov     eax, ecx
        imul    eax, edx
        add     eax, DWORD PTR _c$[esp-4]
        ret     4
@f3@12  ENDP

; ...

        mov     edx, 2
        push    3
        lea     ecx, DWORD PTR [edx-1]
        call    @f3@12
        push    eax
        push    OFFSET $SG81390
        call    _printf
        add     esp, 8
```

Nous voyons que le callee ajuste SP en utilisant l'instruction RETN suivie d'un opérande.

Le nombre d'arguments peut, encore une fois, être facilement déduit.

GCC regparm

Il s'agit en quelque sorte d'une évolution de la convention *fastcall*[2]. L'option `-mregparm` permet de définir le nombre d'arguments (3 au maximum) qui doivent être passés dans les registres. EAX, EDX et ECX sont alors utilisés.

Bien entendu si le nombre d'arguments est inférieur à 3, seuls les premiers registres sont utilisés.

C'est le caller qui effectue l'ajustement du stack pointer.

Pour un exemple, voire (1.22.1 on page 308).

Watcom/OpenWatcom

Dans le cas de ce compilateur, on parle de « register calling convention ». Les 4 premiers arguments sont passés dans les registres EAX, EDX, EBX et ECX. Les paramètres suivant sont passés sur la pile.

Un suffixe constitué d'un caractère de soulignement est ajouté par le compilateur au nom de la fonction afin de les distinguer de celles qui utilisent une autre convention d'appel.

[2] http://go.yurichev.com/17040

4.1.4 thiscall

Cette convention passe le pointeur d'objet *this* à une méthode en C++.

Le compilateur MSVC, passe généralement le pointeur *this* dans le registre ECX.

Le compilateur GCC passe le pointeur *this* comme le premier argument de la fonction. Thus it will be very visible that internally: all function-methods have an extra argument.

Pour un example, voir (**??** on page ??).

4.1.5 x86-64

Windows x64

La méthode utilisée pour passer les arguments aux fonctions en environnement Win64 ressemble beaucoup à la convention `fastcall`. Les 4 premiers arguments sont passés dans les registres RCX, RDX, R8 et R9, les arguments suivants sont passés sur la pile. Le caller doit également préparer un espace sur la pile pour 32 octets, soit 4 mots de 64 bits, le callee pourra y sauvegarder les 4 premiers arguments. Les fonctions suffisamment simples peuvent utiliser les paramètres directement depuis les registres. Les fonctions plus complexes doivent sauvegarder les valeurs de paramètres afin de les utiliser plus tard.

Le caller est aussi responsable de rétablir la valeur du stack pointer à la valeur qu'il avait avant l'appel de la fonction.

Cette convention est utilisée dans les DLLs Windows x86-64 (à la place de *stdcall* en win32).

Exemple:

```
#include <stdio.h>

void f1(int a, int b, int c, int d, int e, int f, int g)
{
        printf ("%d %d %d %d %d %d %d\n", a, b, c, d, e, f, g);
};

int main()
{
        f1(1,2,3,4,5,6,7);
};
```

Listing 4.6: MSVC 2012 /0b

```
$SG2937 DB      '%d %d %d %d %d %d %d', 0aH, 00H

main    PROC
        sub     rsp, 72

        mov     DWORD PTR [rsp+48], 7
        mov     DWORD PTR [rsp+40], 6
        mov     DWORD PTR [rsp+32], 5
        mov     r9d, 4
        mov     r8d, 3
        mov     edx, 2
        mov     ecx, 1
        call    f1

        xor     eax, eax
        add     rsp, 72
        ret     0
main    ENDP

a$ = 80
b$ = 88
c$ = 96
d$ = 104
e$ = 112
f$ = 120
g$ = 128
```

```
f1      PROC
$LN3 :
        mov     DWORD PTR [rsp+32], r9d
        mov     DWORD PTR [rsp+24], r8d
        mov     DWORD PTR [rsp+16], edx
        mov     DWORD PTR [rsp+8], ecx
        sub     rsp, 72

        mov     eax, DWORD PTR g$[rsp]
        mov     DWORD PTR [rsp+56], eax
        mov     eax, DWORD PTR f$[rsp]
        mov     DWORD PTR [rsp+48], eax
        mov     eax, DWORD PTR e$[rsp]
        mov     DWORD PTR [rsp+40], eax
        mov     eax, DWORD PTR d$[rsp]
        mov     DWORD PTR [rsp+32], eax
        mov     r9d, DWORD PTR c$[rsp]
        mov     r8d, DWORD PTR b$[rsp]
        mov     edx, DWORD PTR a$[rsp]
        lea     rcx, OFFSET FLAT :$SG2937
        call    printf

        add     rsp, 72
        ret     0
f1      ENDP
```

Nous vyons ici clairement que des 7 arguments, 4 sont passés dans les registres et les 3 suivants sur la pile.

Le prologue du code de la fonction f1() sauvegarde les arguments dans le « scratch space »—un espace sur la pile précisément prévu à cet effet.

Le compilateur agit ainsi car il n'est pas certain par avance qu'il disposera de suffisamment de registres pour toute la fonction en l'absence des 4 utilisés par les paramètres.

L'appelant est responsable de l'allocation du « scratch space » sur la pile.

Listing 4.7: avec optimisation MSVC 2012 /0b

```
$SG2777 DB      '%d %d %d %d %d %d %d', 0aH, 00H

a$ = 80
b$ = 88
c$ = 96
d$ = 104
e$ = 112
f$ = 120
g$ = 128
f1      PROC
$LN3 :
        sub     rsp, 72

        mov     eax, DWORD PTR g$[rsp]
        mov     DWORD PTR [rsp+56], eax
        mov     eax, DWORD PTR f$[rsp]
        mov     DWORD PTR [rsp+48], eax
        mov     eax, DWORD PTR e$[rsp]
        mov     DWORD PTR [rsp+40], eax
        mov     DWORD PTR [rsp+32], r9d
        mov     r9d, r8d
        mov     r8d, edx
        mov     edx, ecx
        lea     rcx, OFFSET FLAT :$SG2777
        call    printf

        add     rsp, 72
        ret     0
f1      ENDP

main    PROC
        sub     rsp, 72
```

```
        mov     edx, 2
        mov     DWORD PTR [rsp+48], 7
        mov     DWORD PTR [rsp+40], 6
        lea     r9d, QWORD PTR [rdx+2]
        lea     r8d, QWORD PTR [rdx+1]
        lea     ecx, QWORD PTR [rdx-1]
        mov     DWORD PTR [rsp+32], 5
        call    f1

        xor     eax, eax
        add     rsp, 72
        ret     0
main    ENDP
```

L'exemple compilé en branchant les optimisations, sera quasiment identique, si ce n'est que le « scratch space » ne sera pas utilisé car inutile.

Notez également la manière dont MSVC 2012 optimise le chargement de certaines valeurs litérales dans les registres en utilisant LEA (**??** on page ??). L'instruction MOV utiliserait 1 octet de plus (5 au lieu de 4).

?? on page ?? est un autre exemple de cette pratique.

Windows x64: Passage de *this* (C/C++)

Le pointeur *this* est passé dans le registre RCX, le premier argument de la méthode dans RDX, etc. Pour un exemple, voir : **??** on page ??.

Linux x64

Le passage d'arguments par Linux pour x86-64 est quasiment identique à celui de Windows, si ce n'est que 6 registres sont utilisés au lieu de 4 (RDI, RSI, RDX, RCX, R8, R9) et qu'il n'existe pas de « scratch space ». Le callee conserve la possibilité de sauvegarder les registres sur la pile s'il le souhaite ou en a besoin.

Listing 4.8: GCC 4.7.3 avec optimisation

```
.LC0 :
        .string "%d %d %d %d %d %d %d\n"
f1 :
        sub     rsp, 40
        mov     eax, DWORD PTR [rsp+48]
        mov     DWORD PTR [rsp+8], r9d
        mov     r9d, ecx
        mov     DWORD PTR [rsp], r8d
        mov     ecx, esi
        mov     r8d, edx
        mov     esi, OFFSET FLAT :.LC0
        mov     edx, edi
        mov     edi, 1
        mov     DWORD PTR [rsp+16], eax
        xor     eax, eax
        call    __printf_chk
        add     rsp, 40
        ret
main :
        sub     rsp, 24
        mov     r9d, 6
        mov     r8d, 5
        mov     DWORD PTR [rsp], 7
        mov     ecx, 4
        mov     edx, 3
        mov     esi, 2
        mov     edi, 1
        call    f1
        add     rsp, 24
        ret
```

N.B.: Les valeurs sont enregistrées dans la partie basse des registres (e.g., EAX) et non pas dans la totalité du registre 64 bits (RAX). Ceci s'explique par le fait que l'écriture des 32 bits de poids faible du registre remet automatiquement à zéro les 32 bits de poids fort. On suppose qu'AMD a pris cette décision afin de simplifier le portage du code 32 bits vers l'architecture x86-64.

4.1.6 Valeur de retour de type *float* et *double*

Dans toutes les conventions, à l'exception de l'environnement Win64, les valeurs de type *float* ou *double* sont retournées dans le registre FPU `ST(0)`.

En environnement Win64, les valeurs de type *float* et *double* sont respectivement retournées dans les 32 bits de poids faible et dans les 64 bits du registre `XMM0`.

4.1.7 Modification des arguments

Il arrive que les programmeurs C/C++ (et ceux d'autres LPs aussi) se demandent ce qui se passe s'ils modifient la valeur des arguments dans la fonction appelée.

La réponse est simple. Les arguments sont stockés sur la pile, et c'est elle qui est modifiée.

Une fois que le callee s'achève, la fonction appelante ne les utilise pas. (L'auteur de ces lignes n'a jamais constaté qu'il en aille autrement).

```
#include <stdio.h>

void f(int a, int b)
{
        a=a+b;
        printf ("%d\n", a);
};
```

Listing 4.9: MSVC 2012

```
_a$ = 8                                                 ; size = 4
_b$ = 12                                                ; size = 4
_f      PROC
        push    ebp
        mov     ebp, esp
        mov     eax, DWORD PTR _a$[ebp]
        add     eax, DWORD PTR _b$[ebp]
        mov     DWORD PTR _a$[ebp], eax
        mov     ecx, DWORD PTR _a$[ebp]
        push    ecx
        push    OFFSET $SG2938 ; '%d', 0aH
        call    _printf
        add     esp, 8
        pop     ebp
        ret     0
_f      ENDP
```

Donc, oui, la valeur des arguments peut être modifiée sans problème. Sous réserver que l'argument ne soit pas une *references* en C++ (**??** on page ??), et que vous ne modifiez pas la valeur qui est référencée par un pointeur, l'effet de votre modification ne se propagera pas au dehors de la fonction.

En théorie, après le retour du callee, le caller pourrait récupérer l'argument modifié et l'utiliser à sa guise. Ceci pourrait peut être se faire dans un programme rédigé en assembleur.

Par exemple, un compilateur C/C++ génèrera un code comme celui-ci :

```
        push    456     ; will be b
        push    123     ; will be a
        call    f       ; f() modifies its first argument
        add     esp, 2*4
```

Nous pouvons réécrire le code ainsi :

```
        push    456     ; will be b
        push    123     ; will be a
        call    f       ; f() modifies its first argument
        pop     eax
        add     esp, 4
        ; EAX=1st argument of f() modified in f()
```

Il est difficile d'imaginer pourquoi quelqu'un aurait besoin d'agir ainsi, mais en pratique c'est possible. Toujours est-il que les langages C/C++ ne permettent pas de faire ainsi.

4.1.8 Recevoir un argument par adresse

...mieux que cela, il est possible de passer à une fonction l'adresse d'un argument, plutôt que la valeur de l'argument lui-même:

```
#include <stdio.h>

// located in some other file
void modify_a (int *a);

void f (int a)
{
        modify_a (&a);
        printf ("%d\n", a);
};
```

Il est difficile de comprendre ce fonctionnement jusqu'à ce que l'on regarde le code:

Listing 4.10: MSVC 2010 optimisé

```
$SG2796 DB      '%d', 0aH, 00H

_a$ = 8
_f      PROC
        lea     eax, DWORD PTR _a$[esp-4] ; just get the address of value in local stack
        push    eax                       ; and pass it to modify_a()
        call    _modify_a
        mov     ecx, DWORD PTR _a$[esp]   ; reload it from the local stack
        push    ecx                       ; and pass it to printf()
        push    OFFSET $SG2796 ; '%d'
        call    _printf
        add     esp, 12
        ret     0
_f      ENDP
```

L'argument a est placé sur la pile et l'adresse de cet emplacement de pile est passé à une autre fonction. Celle-ci modifie la valeur à l'adresse référencée par le pointeur, puis `printf()` affiche la valeur après modification.

Le lecteur attentif se demandera peut-être ce qu'il en est avec les conventions d'appel qui utilisent les registres pour passer les arguments.

C'est justement une des utilisations du *Shadow Space*.

La valeur en entrée est copiée du registre vers le *Shadow Space* dans la pile locale, puis l'adresse de la pile locale est passée à la fonction appelée:

Listing 4.11: MSVC 2012 x64 optimisé

```
$SG2994 DB      '%d', 0aH, 00H

a$ = 48
f       PROC
        mov     DWORD PTR [rsp+8], ecx    ; save input value in Shadow Space
        sub     rsp, 40
        lea     rcx, QWORD PTR a$[rsp]    ; get address of value and pass it to modify_a()
        call    modify_a
        mov     edx, DWORD PTR a$[rsp]    ; reload value from Shadow Space and pass it to printf↙
    ↳ ()
```

```
        lea     rcx, OFFSET FLAT :$SG2994 ; '%d'
        call    printf
        add     rsp, 40
        ret     0
f       ENDP
```

Le compilateur GCC lui aussi sauvegarde la valeur sur la pile locale:

Listing 4.12: GCC 4.9.1 optimisé x64

```
.LC0 :
        .string "%d\n"
f :
        sub     rsp, 24
        mov     DWORD PTR [rsp+12], edi   ; store input value to the local stack
        lea     rdi, [rsp+12]             ; take an address of the value and pass it to modify_a↙
    ↳ ()
        call    modify_a
        mov     edx, DWORD PTR [rsp+12]   ; reload value from the local stack and pass it to ↙
    ↳ printf()
        mov     esi, OFFSET FLAT :.LC0    ; '%d'
        mov     edi, 1
        xor     eax, eax
        call    __printf_chk
        add     rsp, 24
        ret
```

Le compilateur GCC pour ARM64 se comporte de la même manière, mais l'espace de sauvegarde sur la pile est dénommé *Register Save Area* :

Listing 4.13: GCC 4.9.1 optimisé ARM64

```
f :
        stp     x29, x30, [sp, -32]!
        add     x29, sp, 0          ; setup FP
        add     x1, x29, 32         ; calculate address of variable in Register Save Area
        str     w0, [x1,-4]!        ; store input value there
        mov     x0, x1              ; pass address of variable to the modify_a()
        bl      modify_a
        ldr     w1, [x29,28]        ; load value from the variable and pass it to printf()
        adrp    x0, .LC0 ; '%d'
        add     x0, x0,  :lo12 :.LC0
        bl      printf              ; call printf()
        ldp     x29, x30, [sp], 32
        ret
.LC0 :
        .string "%d\n"
```

Enfin, nous constatons un usage similaire du *Shadow Space* ici: 3.14.1 on page 525.

4.2 Thread Local Storage

TLS est un espace de données propre à chaque thread, qui peut y conserver ce qu'il estime utile. Un exemple d'utilisation bien connu en est la variable globale *errno* du standard C.

Plusieurs threads peuvent invoquer simultanément différentes fonctions qui retournent toutes un code erreur dans la variable *errno*. L'utilisation d'une variable globale dans le contexte d'un programme comportant plusieurs threads serait donc inadaptée dans ce cas. C'est pourquoi la variable *errno* est conservée dans l'espace TLS.

La version 11 du standard C++ a ajouté un nouveau modificateur *thread_local* qui indique que chaque thread possède sa propre copie de la variable décorée par ce modificateur. La variable en question est alors conservée dans l'espace TLS [3] :

[3] C11 also has thread support, optional though

Listing 4.14: C++11

```
#include <iostream>
#include <thread>

thread_local int tmp=3;

int main()
{
        std ::cout << tmp << std ::endl;
};
```

Compilé avec MinGW GCC 4.8.1, mais pas avec MSVC 2012.

Dans le contexte des fichiers au format PE, la variable *tmp* sera allouée dans la section dédiée au TLS du fichier exécutable résultant de la compilation.

4.2.1 Amélioration du générateur linéaire congruent

Le générateur de nombres pseudo-aléatoires 1.23 on page 340 que nous avons étudié précédemment contient une faiblesse. Son comportement est buggé dans un environnement multi-thread. Il utilise en effet une variable d'état dont la valeur peut être lue et/ou modifiée par plusieurs threads simultanément.

Win32

Données TLS non initialisées

Une solution consiste à décorer la variable globale avec le modificateur `__declspec( thread )`. Elle sera alors allouée dans le TLS (ligne 9):

```
1  #include <stdint.h>
2  #include <windows.h>
3  #include <winnt.h>
4
5  // from the Numerical Recipes book :
6  #define RNG_a 1664525
7  #define RNG_c 1013904223
8
9  __declspec( thread ) uint32_t rand_state;
10
11 void my_srand (uint32_t init)
12 {
13         rand_state=init;
14 }
15
16 int my_rand ()
17 {
18         rand_state=rand_state*RNG_a;
19         rand_state=rand_state+RNG_c;
20         return rand_state & 0x7fff;
21 }
22
23 int main()
24 {
25         my_srand(0x12345678);
26         printf ("%d\n", my_rand());
27 };
```

Hiew nous montre alors qu'il existe une nouvelle section nommée `.tls` dans le fichier PE.

Listing 4.15: avec optimisation MSVC 2013 x86

```
_TLS    SEGMENT
_rand_state DD  01H DUP (?)
_TLS    ENDS

_DATA   SEGMENT
```

```
$SG84851 DB     '%d', 0aH, 00H
_DATA   ENDS
_TEXT   SEGMENT

_init$ = 8      ; size = 4
_my_srand PROC
; FS :0=address of TIB
        mov     eax, DWORD PTR fs :__tls_array  ; displayed in IDA as FS :2Ch
; EAX=address of TLS of process
        mov     ecx, DWORD PTR __tls_index
        mov     ecx, DWORD PTR [eax+ecx*4]
; ECX=current TLS segment
        mov     eax, DWORD PTR _init$[esp-4]
        mov     DWORD PTR _rand_state[ecx], eax
        ret     0
_my_srand ENDP

_my_rand PROC
; FS :0=address of TIB
        mov     eax, DWORD PTR fs :__tls_array  ; displayed in IDA as FS :2Ch
; EAX=address of TLS of process
        mov     ecx, DWORD PTR __tls_index
        mov     ecx, DWORD PTR [eax+ecx*4]
; ECX=current TLS segment
        imul    eax, DWORD PTR _rand_state[ecx], 1664525
        add     eax, 1013904223         ; 3c6ef35fH
        mov     DWORD PTR _rand_state[ecx], eax
        and     eax, 32767              ; 00007fffH
        ret     0
_my_rand ENDP

_TEXT   ENDS
```

La variable `rand_state` se trouve donc maintenant dans le segment TLS et chaque thread en possède sa propre copie.

Voyons comment elle est accédée. Pour cela, chargeons l'adresse du TIB depuis FS:2Ch. Ajoutons-y (si nécessaire) un index puis calculons l'adresse du segment TLS.

Il est ainsi possible d'accéder la valeur de la variable `rand_state` au travers du registre ECX qui contient une adresse propre à chaque thread.

Le sélecteur `FS:` est connu de tous les reverse engineer. Il est spécifiquement utilisé pour toujours contenir l'adresse du TIB du thread en cours d'exécution. L'accès aux données propres à chaque thread est donc une opération performante.

En environnement Win64, c'est le sélecteur `GS:` qui est utilisé pour ce faire. L'adresse de l'espace TLS y est conservé à l'offset 0x58 :

Listing 4.16: avec optimisation MSVC 2013 x64

```
_TLS    SEGMENT
rand_state DD   01H DUP (?)
_TLS    ENDS

_DATA   SEGMENT
$SG85451 DB     '%d', 0aH, 00H
_DATA   ENDS

_TEXT   SEGMENT

init$ = 8
my_srand PROC
        mov     edx, DWORD PTR _tls_index
        mov     rax, QWORD PTR gs :88 ; 58h
        mov     r8d, OFFSET FLAT :rand_state
        mov     rax, QWORD PTR [rax+rdx*8]
        mov     DWORD PTR [r8+rax], ecx
        ret     0
my_srand ENDP
```

```
my_rand PROC
        mov     rax, QWORD PTR gs :88 ; 58h
        mov     ecx, DWORD PTR _tls_index
        mov     edx, OFFSET FLAT :rand_state
        mov     rcx, QWORD PTR [rax+rcx*8]
        imul    eax, DWORD PTR [rcx+rdx], 1664525 ; 0019660dH
        add     eax, 1013904223         ; 3c6ef35fH
        mov     DWORD PTR [rcx+rdx], eax
        and     eax, 32767              ; 00007fffH
        ret     0
my_rand ENDP

_TEXT   ENDS
```

Initialisation des données TLS

Imaginons maintenant que nous voulons nous prémunir des erreurs de programmation en initialisant systématiquement la variable rand_state avec une valeur constante (ligne 9):

```
1  #include <stdint.h>
2  #include <windows.h>
3  #include <winnt.h>
4
5  // from the Numerical Recipes book :
6  #define RNG_a 1664525
7  #define RNG_c 1013904223
8
9  __declspec( thread ) uint32_t rand_state=1234;
10
11 void my_srand (uint32_t init)
12 {
13         rand_state=init;
14 }
15
16 int my_rand ()
17 {
18         rand_state=rand_state*RNG_a;
19         rand_state=rand_state+RNG_c;
20         return rand_state & 0x7fff;
21 }
22
23 int main()
24 {
25         printf ("%d\n", my_rand());
26 };
```

Le code ne semble pas différent de celui que nous avons étudié. Pourtant dans IDA nous constatons:

```
.tls :00404000 ; Segment type : Pure data
.tls :00404000 ; Segment permissions : Read/Write
.tls :00404000 _tls            segment para public 'DATA' use32
.tls :00404000                 assume cs :_tls
.tls :00404000                 ;org 404000h
.tls :00404000 TlsStart        db    0         ; DATA XREF : .rdata :TlsDirectory
.tls :00404001                 db    0
.tls :00404002                 db    0
.tls :00404003                 db    0
.tls :00404004                 dd 1234
.tls :00404008 TlsEnd          db    0         ; DATA XREF : .rdata :TlsEnd_ptr
...
```

La valeur 1234 est bien présente. Chaque fois qu'un nouveau thread est créé, un nouvel espace TLS est alloué pour ses besoins et toutes les données - y compris 1234 - y sont copiées.

Considérons le scénario hypothétique suivant:

- Le thread A démarre. Un espace TLS est créé pour ses besoins et la valeur 1234 est copiée dans rand_state.

- La fonction `my_rand()` est invoquée plusieurs fois par le thread A.
  La valeur de la variable `rand_state` est maintenant différente de 1234.
- Le thread B démarre. Un espace TLS est créé pour ses besoins et la valeur 1234 est copiée dans `rand_state`. Dans le thread A, la valeur de `rand_state` demeure différente de 1234.

Fonctions de rappel TLS

Mais comment procédons-nous si les variables conservées dans l'environnement TLS doivent être initialisées avec des valeurs qui ne sont pas constantes ?

Imaginons le scénario suivant: Il se peut que le programmeur oublie d'invoquer la fonction `my_srand()` pour initialiser le PRNG. Malgré cela, le générateur doit être initialisé avec une valeur réellement aléatoire et non pas avec 1234. C'est précisément dans ce genre de cas que les fonctions de rappel TLS sont utilisées.

Le code ci-dessous n'est pas vraiment portable du fait du bricolage, mais vous devriez comprendre l'idée.

Nous définissons une fonction (`tls_callback()`) qui doit être invoquée avant le démarrage du processus et/ou d'un thread.

Cette fonction initialise le PRNG avec la valeur retournée par la fonction `GetTickCount()`.

```
#include <stdint.h>
#include <windows.h>
#include <winnt.h>

// from the Numerical Recipes book :
#define RNG_a 1664525
#define RNG_c 1013904223

__declspec( thread ) uint32_t rand_state;

void my_srand (uint32_t init)
{
        rand_state=init;
}

void NTAPI tls_callback(PVOID a, DWORD dwReason, PVOID b)
{
        my_srand (GetTickCount());
}

#pragma data_seg(".CRT$XLB")
PIMAGE_TLS_CALLBACK p_thread_callback = tls_callback;
#pragma data_seg()

int my_rand ()
{
        rand_state=rand_state*RNG_a;
        rand_state=rand_state+RNG_c;
        return rand_state & 0x7fff;
}

int main()
{
        // rand_state is already initialized at the moment (using GetTickCount())
        printf ("%d\n", my_rand());
};
```

Voyons cela dans IDA:

Listing 4.17: avec optimisation MSVC 2013

```
.text :00401020 TlsCallback_0   proc near               ; DATA XREF : .rdata :TlsCallbacks
.text :00401020                 call    ds :GetTickCount
.text :00401026                 push    eax
.text :00401027                 call    my_srand
.text :0040102C                 pop     ecx
.text :0040102D                 retn    0Ch
```

```
.text :0040102D TlsCallback_0   endp

...

.rdata :004020C0 TlsCallbacks    dd offset TlsCallback_0 ; DATA XREF : .rdata :TlsCallbacks_ptr

...

.rdata :00402118 TlsDirectory    dd offset TlsStart
.rdata :0040211C TlsEnd_ptr      dd offset TlsEnd
.rdata :00402120 TlsIndex_ptr    dd offset TlsIndex
.rdata :00402124 TlsCallbacks_ptr dd offset TlsCallbacks
.rdata :00402128 TlsSizeOfZeroFill dd 0
.rdata :0040212C TlsCharacteristics dd 300000h
```

Les fonctions de rappel TLS sont parfois utilisées par les mécanismes de décompression d'exécutable afin d'en rendre le fonctionnement plus obscure.

Cette pratique peut en laisser certains dans le noir parce qu'ils auront omis de considérer qu'un fragment de code a pu s'exécuter avant l'OEP[4].

Linux

Voyons maintenant comment une variable globale conservée dans l'espace de stockage propre au thread est déclarée avec GCC:

```
__thread uint32_t rand_state=1234;
```

Il ne s'agit pas du modificateur standard C/C++ modifier, mais bien d'un modificateur spécifique à GCC [5].

Le sélecteur `GS:` est utilisé lui aussi pour accéder au TLS, mais d'une manière un peu différente:

Listing 4.18: avec optimisation GCC 4.8.1 x86

```
.text :08048460 my_srand        proc near
.text :08048460
.text :08048460 arg_0           = dword ptr  4
.text :08048460
.text :08048460                 mov     eax, [esp+arg_0]
.text :08048464                 mov     gs :0FFFFFFFCh, eax
.text :0804846A                 retn
.text :0804846A my_srand        endp

.text :08048470 my_rand         proc near
.text :08048470                 imul    eax, gs :0FFFFFFFCh, 19660Dh
.text :0804847B                 add     eax, 3C6EF35Fh
.text :08048480                 mov     gs :0FFFFFFFCh, eax
.text :08048486                 and     eax, 7FFFh
.text :0804848B                 retn
.text :0804848B my_rand         endp
```

Pour en savoir plus: [Ulrich Drepper, *ELF Handling For Thread-Local Storage*, (2013)][6].

4.3 Appels systèmes (syscall-s)

Comme nous le savons, tous les processus d'un OS sont divisés en deux catégories: ceux qui ont un accès complet au matériel (« kernel space » espace noyau) et ceux qui ne l'ont pas (« user space » espace utilisateur).

Le noyau de l'OS et, en général, les drivers sont dans la première catégorie.

Toutes les applications sont d'habitude dans la seconde catégorie.

[4] Original Entry Point
[5] http://go.yurichev.com/17062
[6] Aussi disponible en http://go.yurichev.com/17272

Par exemple, le noyai Linux est dans le *kernel space*, mais la Glibc est dans le *user space*.

Cette séparation est cruciale pour la sécurité de l'OS : il est très important de ne pas donner la possibilité à n'importe quel processus de casser quelque chose dans un autre processus ou même dans le noyau de l'OS. D'un autre côté, un driver qui plante ou une erreur dans le noyau de l'OS se termine en général par un kernel panic ou un BSOD[7].

La protection en anneau dans les processeurs x86 permet de séparer l'ensemble dans quatre niveaux de protection (rings), mais tant dans Linux que Windows, seuls deux d'entre eux sont utilisés: ring0 (« kernel space ») et ring3 (« user space »).

Les appels systèmes (syscall-s) sont un point où deux espaces sont connectés.

On peut dire que c'est la principale API[8] fournie aux applications.

Comme dans Windows NT, la table des appels système se trouve dans la SSDT[9].

L'utilisation des appels système est très répandue parmis les auteurs de shellcode et de virus, car il est difficile de déterminer l'adresse des fonctions nécessaires dans les bibliothèques système, mais il est simple d'utiliser les appels système. Toutefois, il est nécessaire d'écrire plus de code à cause du niveau d'abstraction plus bas de l'API.

Il faut aussi noter que les numéros des appels systèmes peuvent être différent dans différentes versions d'un OS.

4.3.1 Linux

Sous Linux, un appel système s'effectue d'habitude par `int 0x80`. Le numéro de l'appel est passé dans le registre EAX, et tout autre paramètre —dans les autres registres.

Listing 4.19: Un exemple simple de l'utilisation de deux appels système

```
section .text
global  _start

_start :
        mov     edx,len ; buffer len
        mov     ecx,msg ; buffer
        mov     ebx,1   ; file descriptor. 1 is for stdout
        mov     eax,4   ; syscall number. 4 is for sys_write
        int     0x80

        mov     eax,1   ; syscall number. 1 is for sys_exit
        int     0x80

section .data

msg     db  'Hello, world!',0xa
len     equ $ - msg
```

Compilation:

```
nasm -f elf32 1.s
ld 1.o
```

La liste complète des appels systèmes sous Linux: http://go.yurichev.com/17319.

Pour intercepter les appels système et les suivre sous Linux, strace(5.2.3 on page 602) peut être utilisé.

4.3.2 Windows

Ici, ils sont appelés via `int 0x2e` ou en utilisant l'instruction x86 spéciale SYSENTER.

La liste complète des appels systèmes sous Windows: http://go.yurichev.com/17320.

Pour aller plus loin:

[7] Blue Screen of Death
[8] Application Programming Interface
[9] System Service Dispatch Table

« Windows Syscall Shellcode » par Piotr Bania: http://go.yurichev.com/17321.

4.4 Linux

4.4.1 Code indépendant de la position

Lorsque l'on analyse des bibliothèques partagées sous Linux (.so), on rencontre souvent ce genre de code:

Listing 4.20: libc-2.17.so x86

```
.text :0012D5E3 __x86_get_pc_thunk_bx proc near         ; CODE XREF : sub_17350+3
.text :0012D5E3                                         ; sub_173CC+4 ...
.text :0012D5E3                 mov     ebx, [esp+0]
.text :0012D5E6                 retn
.text :0012D5E6 __x86_get_pc_thunk_bx endp

...

.text :000576C0 sub_576C0       proc near               ; CODE XREF : tmpfile+73

...

.text :000576C0                 push    ebp
.text :000576C1                 mov     ecx, large gs :0
.text :000576C8                 push    edi
.text :000576C9                 push    esi
.text :000576CA                 push    ebx
.text :000576CB                 call    __x86_get_pc_thunk_bx
.text :000576D0                 add     ebx, 157930h
.text :000576D6                 sub     esp, 9Ch

...

.text :000579F0                 lea     eax, (a__gen_tempname - 1AF000h)[ebx] ; "__gen_tempname↙
    ↳ "
.text :000579F6                 mov     [esp+0ACh+var_A0], eax
.text :000579FA                 lea     eax, (a__SysdepsPosix - 1AF000h)[ebx] ; "../sysdeps/↙
    ↳ posix/tempname.c"
.text :00057A00                 mov     [esp+0ACh+var_A8], eax
.text :00057A04                 lea     eax, (aInvalidKindIn_ - 1AF000h)[ebx] ; "! \"invalid ↙
    ↳ KIND in __gen_tempname\""
.text :00057A0A                 mov     [esp+0ACh+var_A4], 14Ah
.text :00057A12                 mov     [esp+0ACh+var_AC], eax
.text :00057A15                 call    __assert_fail
```

Tous les pointeurs sur des chaînes sont corrigés avec une certaine constante et la valeur de EBX, qui est calculée au début de chaque fonction.

C'est ce que l'on appelle du code PIC[10], qui peut être exécuté à n'importe quelle adresse mémoire, c'est pourquoi il ne doit contenir aucune adresse absolue.

PIC était crucial dans les premiers ordinateurs, et l'est encore aujourd'hui dans les systèmes embarqués sans support de la mémoire virtuelle (où tous les processus se trouvent dans un seul bloc continu de mémoire).

C'est encore utilisé sur les systèmes *NIX pour les bibliothèques partagées, car elles sont utilisées par de nombreux processus mais ne sont chargées qu'une seule fois en mémoire. Mais tous ces processus peuvent mapper la même bibliothèque partagée à des adresses différentes, c'est pourquoi elles doivent fonctionner correctement sans aucune adresse absolue.

Faisons un petit essai:

```
#include <stdio.h>

int global_variable=123;
```

[10] Position Independent Code

```
int f1(int var)
{
    int rt=global_variable+var;
    printf ("returning %d\n", rt);
    return rt;
};
```

Compilons le avec GCC 4.7.3 et examinons le fichier .so généré dans IDA :

```
gcc -fPIC -shared -O3 -o 1.so 1.c
```

Listing 4.21: GCC 4.7.3

```
.text :00000440                 public __x86_get_pc_thunk_bx
.text :00000440 __x86_get_pc_thunk_bx proc near         ; CODE XREF : _init_proc+4
.text :00000440                                         ; deregister_tm_clones+4 ...
.text :00000440                 mov     ebx, [esp+0]
.text :00000443                 retn
.text :00000443 __x86_get_pc_thunk_bx endp

.text :00000570                 public f1
.text :00000570 f1              proc near
.text :00000570
.text :00000570 var_1C          = dword ptr -1Ch
.text :00000570 var_18          = dword ptr -18h
.text :00000570 var_14          = dword ptr -14h
.text :00000570 var_8           = dword ptr -8
.text :00000570 var_4           = dword ptr -4
.text :00000570 arg_0           = dword ptr  4
.text :00000570
.text :00000570                 sub     esp, 1Ch
.text :00000573                 mov     [esp+1Ch+var_8], ebx
.text :00000577                 call    __x86_get_pc_thunk_bx
.text :0000057C                 add     ebx, 1A84h
.text :00000582                 mov     [esp+1Ch+var_4], esi
.text :00000586                 mov     eax, ds :(global_variable_ptr - 2000h)[ebx]
.text :0000058C                 mov     esi, [eax]
.text :0000058E                 lea     eax, (aReturningD - 2000h)[ebx] ; "returning %d\n"
.text :00000594                 add     esi, [esp+1Ch+arg_0]
.text :00000598                 mov     [esp+1Ch+var_18], eax
.text :0000059C                 mov     [esp+1Ch+var_1C], 1
.text :000005A3                 mov     [esp+1Ch+var_14], esi
.text :000005A7                 call    ___printf_chk
.text :000005AC                 mov     eax, esi
.text :000005AE                 mov     ebx, [esp+1Ch+var_8]
.text :000005B2                 mov     esi, [esp+1Ch+var_4]
.text :000005B6                 add     esp, 1Ch
.text :000005B9                 retn
.text :000005B9 f1              endp
```

C'est ça: les pointeurs sur *«returning %d\n»* et *global_variable* sont corrigés à chaque exécution de la fonction.

La fonction `__x86_get_pc_thunk_bx()` renvoie dans EBX l'adresse de l'instruction après son appel (`0x57C` ici).

C'est un moyen simple d'obtenir la valeur du compteur de programme (EIP) à un endroit quelconque. La constante `0x1A84` est relative à la différence entre le début de cette fonction et ce que l'on appelle *Global Offset Table Procedure Linkage Table* (GOT PLT), la section juste après la *Global Offset Table* (GOT), où se trouve le pointeur sur *global_variable*. IDA montre ces offsets sous leur forme calculée pour rendre les choses plus facile à comprendre, mais en fait, le code est:

```
.text :00000577                 call    __x86_get_pc_thunk_bx
.text :0000057C                 add     ebx, 1A84h
.text :00000582                 mov     [esp+1Ch+var_4], esi
.text :00000586                 mov     eax, [ebx-0Ch]
.text :0000058C                 mov     esi, [eax]
.text :0000058E                 lea     eax, [ebx-1A30h]
```

Ici, EBX pointe sur la section GOT PLT et pour calculer le pointeur sur *global_variable* (qui est stocké dans la GOT), il faut soustraire 0xC.

Pour calculer la valeur du pointeur sur la chaîne *«returning %d\n»*, il faut soustraire 0x1A30.

A propos, c'est la raison pour laquelle le jeu d'instructions AMD64 ajoute le support d'adressage relatif de RIP[11] — pour simplifier le code PIC.

Compilons le même code C en utilisant la même version de GCC, mais pour x64.

IDA simplifierait le code en supprimant les détails de l'adressage relatif à RIP, donc utilisons *objdump* à la place d'IDA pour tout voir:

```
0000000000000720 <f1> :
 720:   48 8b 05 b9 08 20 00    mov    rax,QWORD PTR [rip+0x2008b9]        # 200fe0 <_DYNAMIC+0⤶
    ↳ x1d0>
 727:   53                      push   rbx
 728:   89 fb                   mov    ebx,edi
 72a :   48 8d 35 20 00 00 00    lea    rsi,[rip+0x20]        # 751 <_fini+0x9>
 731:   bf 01 00 00 00          mov    edi,0x1
 736:   03 18                   add    ebx,DWORD PTR [rax]
 738:   31 c0                   xor    eax,eax
 73a :   89 da                   mov    edx,ebx
 73c :   e8 df fe ff ff          call   620 <__printf_chk@plt>
 741:   89 d8                   mov    eax,ebx
 743:   5b                      pop    rbx
 744:   c3                      ret
```

0x2008b9 est la différence entre l'adresse de l'instruction en 0x720 et *global_variable*, et 0x20 est la différence entre l'adresse de l'instruction en 0x72A et la chaîne *«returning %d\n»*.

Comme on le voit, le fait d'avoir à recalculer fréquemment les adresses rend l'exécution plus lente (cela dit, ça c'est amélioré en x64).

Donc, il est probablement mieux de lier statiquement si vous vous préoccupez des performances [voir: Agner Fog, *Optimizing software in C++* (2015)].

Windows

Le mécanisme PIC n'est pas utilisé dans les DLLs de Windows. Si le chargeur de Windows doit charger une DLL à une autre adresse, il « patche » la DLL en mémoire (aux places *FIXUP*) afin de corriger toutes les adresses.

Cela implique que plusieurs processus Windows ne peuvent pas partager une DLL déjà chargée à une adresse différente dans des blocs mémoire de différents processus — puisque chaque instance qui est chargée en mémoire est *fixé* pour fonctionner uniquement à ces adresses...

4.4.2 Hack *LD_PRELOAD* sur Linux

Cela permet de charger nos propres bibilothèques dynamiques avant les autres, même celle du système, comme libc.so.6.

Cela permet de « substituer » nos propres fonctions, avant celles originales des bibliothèques du système. Par exemple, il est facile d'intercepter tous les appels à time(), read(), write(), etc.

Regardons si l'on peut tromper l'utilitaire *uptime*. Comme chacun le sait, il indique depuis combien de temps l'ordinateur est démarré. Avec l'aide de strace(5.2.3 on page 602), on voit que ces informations proviennent du fichier /proc/uptime :

```
$ strace uptime
...
open("/proc/uptime", O_RDONLY)          = 3
lseek(3, 0, SEEK_SET)                   = 0
read(3, "416166.86 414629.38\n", 2047)  = 20
...
```

[11] compteur de programme sur AMD64

Ce n'est pas un fichier réel sur le disque, il est virtuel et généré au vol par le noyau Linux. Il y a seulement deux nombres:

```
$ cat /proc/uptime
416690.91 415152.03
```

Ce que l'on peut apprendre depuis Wikipédia [12] :

> Le premier est le nombre total de seconde depuis lequel le système est démarré. Le second est la part de ce temps pendant lequel la machine était inoccupée, en secondes.

Essayons d'écrire notre propre bibliothèque dynamique, avec les fonctions open(), read(), close() fonctionnants comme nous en avons besoin.

Tout d'abord, notre fonction open() va comparer le nom du fichier à ouvrir avec celui que l'on veut modifier, et si c'est le cas, sauver le descripteur du fichier ouvert.

Ensuite, si read() est appelé avec ce descripteur de fichier, nous allons substituer la sortie, et dans les autres cas, nous appelerons la fonction read() originale de libc.so.6. Et enfin close() fermera le fichier que nous suivons.

Nous allons utiliser les fonctions dlopen() et dlsym() pour déterminer les adresses des fonctions originales dans libc.so.6.

Nous en avons besoin pour appeler les fonctions « réelles ».

D'un autre côté, si nous interceptons strcmp() et surveillons chaque comparaison de chaînes dans le programme, nous pouvons alors implémenter notre propre fonction strcmp(), et ne pas utiliser la fonction originale du tout.

[13].

```
#include <stdio.h>
#include <stdarg.h>
#include <stdlib.h>
#include <stdbool.h>
#include <unistd.h>
#include <dlfcn.h>
#include <string.h>

void *libc_handle = NULL;
int (*open_ptr)(const char *, int) = NULL;
int (*close_ptr)(int) = NULL;
ssize_t (*read_ptr)(int, void*, size_t) = NULL;

bool inited = false;

_Noreturn void die (const char * fmt, ...)
{
        va_list va;
        va_start (va, fmt);

        vprintf (fmt, va);
        exit(0);
};

static void find_original_functions ()
{
        if (inited)
                return;

        libc_handle = dlopen ("libc.so.6", RTLD_LAZY);
        if (libc_handle==NULL)
                die ("can't open libc.so.6\n");

        open_ptr = dlsym (libc_handle, "open");
        if (open_ptr==NULL)
                die ("can't find open()\n");
```

[12] Wikipédia

[13] Par exemple, voilà comment implémenter une interception simple de strcmp() dans cet article [14] écrit par Yong Huang

```
        close_ptr = dlsym (libc_handle, "close");
        if (close_ptr==NULL)
                die ("can't find close()\n");

        read_ptr = dlsym (libc_handle, "read");
        if (read_ptr==NULL)
                die ("can't find read()\n");

        inited = true;
}

static int opened_fd=0;

int open(const char *pathname, int flags)
{
        find_original_functions();

        int fd=(*open_ptr)(pathname, flags);
        if (strcmp(pathname, "/proc/uptime")==0)
                opened_fd=fd; // that's our file! record its file descriptor
        else
                opened_fd=0;
        return fd;
};

int close(int fd)
{
        find_original_functions();

        if (fd==opened_fd)
                opened_fd=0; // the file is not opened anymore
        return (*close_ptr)(fd);
};

ssize_t read(int fd, void *buf, size_t count)
{
        find_original_functions();

        if (opened_fd!=0 && fd==opened_fd)
        {
                // that's our file!
                return snprintf (buf, count, "%d %d", 0x7fffffff, 0x7fffffff)+1;
        };
        // not our file, go to real read() function
        return (*read_ptr)(fd, buf, count);
};
```

(Code source sur GitHub)

Compilons le comme une bibliothèque dynamique standard:

```
gcc -fpic -shared -Wall -o fool_uptime.so fool_uptime.c -ldl
```

Lançons *uptime* en chargeant notre bibliothèque avant les autres:

```
LD_PRELOAD=`pwd`/fool_uptime.so uptime
```

Et nous voyons:

```
 01:23:02 up 24855 days,  3:14,  3 users,  load average : 0.00, 0.01, 0.05
```

Si la variable d'environnement *LD_PRELOAD*

pointe toujours sur le nom et le chemin de notre bibliothèque, elle sera chargée pour tous les programmes démarrés.

Plus d'exemples:

- Interception simple de strcmp() (Yong Huang) http://go.yurichev.com/17143

- Kevin Pulo—Jouons avec LD_PRELOAD. De nombreux exemples et idées. yurichev.com
- Interception des fonctions de fichier pour compresser/décompresser les fichiers au vol (zlibc) http://go.yurichev.com/17146

4.5 Windows NT

4.5.1 CRT (win32)

L'exécution du programme débute donc avec la fonction `main()` ? Non, pas du tout.

Ouvrez un exécutable dans IDA ou HIEW, et vous constaterez que l'OEP pointe sur un bloc de code qui se situe ailleurs.

Ce code prépare l'environnement avant de passer le contrôle à notre propre code. Ce fragment de code initial est dénommé CRT (pour C RunTime).

La fonction `main()` accepte en arguments un tableau des paramètres passés en ligne de commande, et un autre contenant les variables d'environnement. En réalité, ce qui est passé au programme est une simple chaîne de caractères. Le code du CRT repère les espaces dans la chaîne et découpe celle-ci en plusieurs morceaux qui constitueront le tableau des paramètres. Le CRT est également responsable de la préparation du tableau des variables d'environnement envp.

En ce qui concerne les applications win32 dotées d'un interface graphique (GUI[15]). La fonction principale est nommée `WinMain` et non pas `main()`. Elle possède aussi ses propres arguments:

```
int CALLBACK WinMain(
  _In_  HINSTANCE hInstance,
  _In_  HINSTANCE hPrevInstance,
  _In_  LPSTR lpCmdLine,
  _In_  int nCmdShow
);
```

C'est toujours le CRT qui est responsable de leur préparation.

La valeur retournée par la fonction `main()` est également ce qui constitue le code retour du programme.

Le CRT peut utiliser cette valeur lors de l'appel de la fonction `ExitProcess()` qui prend le code retour du programme en paramètre.

En règle générale, le code du CRT est propre à chaque compilateur.

Voici celui du MSVC 2008.

```
1  ___tmainCRTStartup proc near
2
3  var_24 = dword ptr -24h
4  var_20 = dword ptr -20h
5  var_1C = dword ptr -1Ch
6  ms_exc = CPPEH_RECORD ptr -18h
7
8         push    14h
9         push    offset stru_4092D0
10        call    __SEH_prolog4
11        mov     eax, 5A4Dh
12        cmp     ds :400000h, ax
13        jnz     short loc_401096
14        mov     eax, ds :40003Ch
15        cmp     dword ptr [eax+400000h], 4550h
16        jnz     short loc_401096
17        mov     ecx, 10Bh
18        cmp     [eax+400018h], cx
19        jnz     short loc_401096
20        cmp     dword ptr [eax+400074h], 0Eh
21        jbe     short loc_401096
22        xor     ecx, ecx
```

[15] Graphical User Interface

```
23        cmp     [eax+4000E8h], ecx
24        setnz   cl
25        mov     [ebp+var_1C], ecx
26        jmp     short loc_40109A
27
28
29  loc_401096 : ; CODE XREF : ___tmainCRTStartup+18
30               ; ___tmainCRTStartup+29 ...
31        and     [ebp+var_1C], 0
32
33  loc_40109A : ; CODE XREF : ___tmainCRTStartup+50
34        push    1
35        call    __heap_init
36        pop     ecx
37        test    eax, eax
38        jnz     short loc_4010AE
39        push    1Ch
40        call    _fast_error_exit
41        pop     ecx
42
43  loc_4010AE : ; CODE XREF : ___tmainCRTStartup+60
44        call    __mtinit
45        test    eax, eax
46        jnz     short loc_4010BF
47        push    10h
48        call    _fast_error_exit
49        pop     ecx
50
51  loc_4010BF : ; CODE XREF : ___tmainCRTStartup+71
52        call    sub_401F2B
53        and     [ebp+ms_exc.disabled], 0
54        call    __ioinit
55        test    eax, eax
56        jge     short loc_4010D9
57        push    1Bh
58        call    __amsg_exit
59        pop     ecx
60
61  loc_4010D9 : ; CODE XREF : ___tmainCRTStartup+8B
62        call    ds :GetCommandLineA
63        mov     dword_40B7F8, eax
64        call    ___crtGetEnvironmentStringsA
65        mov     dword_40AC60, eax
66        call    __setargv
67        test    eax, eax
68        jge     short loc_4010FF
69        push    8
70        call    __amsg_exit
71        pop     ecx
72
73  loc_4010FF : ; CODE XREF : ___tmainCRTStartup+B1
74        call    __setenvp
75        test    eax, eax
76        jge     short loc_401110
77        push    9
78        call    __amsg_exit
79        pop     ecx
80
81  loc_401110 : ; CODE XREF : ___tmainCRTStartup+C2
82        push    1
83        call    __cinit
84        pop     ecx
85        test    eax, eax
86        jz      short loc_401123
87        push    eax
88        call    __amsg_exit
89        pop     ecx
90
91  loc_401123 : ; CODE XREF : ___tmainCRTStartup+D6
92        mov     eax, envp
```

```
93        mov     dword_40AC80, eax
94        push    eax             ; envp
95        push    argv            ; argv
96        push    argc            ; argc
97        call    _main
98        add     esp, 0Ch
99        mov     [ebp+var_20], eax
100       cmp     [ebp+var_1C], 0
101       jnz     short $LN28
102       push    eax             ; uExitCode
103       call    $LN32
104
105   $LN28 :       ; CODE XREF : ___tmainCRTStartup+105
106       call    __cexit
107       jmp     short loc_401186
108
109
110   $LN27 :       ; DATA XREF : .rdata :stru_4092D0
111       mov     eax, [ebp+ms_exc.exc_ptr]  ; Exception filter 0 for function 401044
112       mov     ecx, [eax]
113       mov     ecx, [ecx]
114       mov     [ebp+var_24], ecx
115       push    eax
116       push    ecx
117       call    __XcptFilter
118       pop     ecx
119       pop     ecx
120
121   $LN24 :
122       retn
123
124
125   $LN14 :       ; DATA XREF : .rdata :stru_4092D0
126       mov     esp, [ebp+ms_exc.old_esp]  ; Exception handler 0 for function 401044
127       mov     eax, [ebp+var_24]
128       mov     [ebp+var_20], eax
129       cmp     [ebp+var_1C], 0
130       jnz     short $LN29
131       push    eax             ; int
132       call    __exit
133
134
135   $LN29 :       ; CODE XREF : ___tmainCRTStartup+135
136       call    __c_exit
137
138   loc_401186 : ; CODE XREF : ___tmainCRTStartup+112
139       mov     [ebp+ms_exc.disabled], 0FFFFFFFEh
140       mov     eax, [ebp+var_20]
141       call    __SEH_epilog4
142       retn
```

Nous y trouvons des appels à `GetCommandLineA()` (line 62), puis `setargv()` (line 66) et `setenvp()` (line 74), qui semblent utilisés pour initialiser les variables globales `argc`, `argv`, `envp`.

Enfin, `main()` est invoqué avec ces arguments (line 97).

Nous observons également des appels à des fonctions au nom évoquateur telles que `heap_init()` (line 35), `ioinit()` (line 54).

Le tas heap est lui aussi initialisé par le CRT. Si vous tentez d'utiliser la fonction `malloc()` dans un programme ou le CRT n'a pas été ajouté, le programme va s'achever de manière anormale avec l'erreur suivante:

```
runtime error R6030
- CRT not initialized
```

L'initialisation des objets globaux en C++ est également de la responsabilité du CRT avant qu'il ne passe la main à `main()` : **??** on page ??.

La valeur retournée par `main()` est passée soit à la fonction `cexit()`, soit à `$LN32` qui à son tour invoque `doexit()`.

Mais pourrait-on se passe du CRT ? Oui, si vous savez ce que vous faites.

L'éditeur de lien MSVC accepte une option /ENTRY qui permet de définir le point d'entrée du programme.

```
#include <windows.h>

int main()
{
        MessageBox (NULL, "hello, world", "caption", MB_OK);
};
```

Compilons ce programme avec MSVC 2008.

```
cl no_crt.c user32.lib /link /entry :main
```

Nous obtenons un programme exécutable .exe de 2560 octets qui contient en tout et pour tout l'en-tête PE, les instructions pour invoquer MessageBox et deux chaînes de caractères dans le segment de données: le nom de la fonction MessageBox et celui de sa DLL d'origine user32.dll.

Cela fonctionne, mais vous ne pouvez pas déclarer WinMain et ses 4 arguments à la place de main().

Pour être précis, vous pourriez, mais les arguments ne seraient pas préparés au moment de l'exécution du programme.

Cela étant, il est possible de réduire encore la taille de l'exécutable en utilisant une valeur pour l'alignement des sections du fichier au format PE une valeur inférieur à celle par défaut de 4096 octets.

```
cl no_crt.c user32.lib /link /entry :main /align :16
```

L'éditeur de lien prévient cependant:

```
LINK  : warning LNK4108 : /ALIGN specified without /DRIVER; image may not run
```

Nous pouvons ainsi obtenir un exécutable de 720 octets. Son exécution est possible sur Windows 7 x86, mais pas en environnement x64 (un message d'erreur apparaîtra alors si vous tentez de l'exécuter).

Avec des efforts accrus il est possible de réduire encore la taille, mais avec des problèmes de compatibilité comme vous le voyez.

4.5.2 Win32 PE

PE est un format de fichier exécutable utilisé sur Windows. La différence entre .exe, .dll et .sys est que les .exe et .sys n'ont en général pas d'exports, uniquement des imports.

Une DLL[16], tout comme n'importe quel fichier PE, possède un point d'entrée (OEP) (la fonction DllMain() se trouve là) mais cette fonction ne fait généralement rien. .sys est généralement un pilote de périphérique. Pour les pilotes, Windows exige que la somme de contrôle soit présente dans le fichier PE et qu'elle soit correcte [17].

A partir de Windows Vista, un fichier de pilote doit aussi être signé avec une signature numérique. Le chargement échouera sinon.

Chaque fichier PE commence avec un petit programme DOS qui affiche un message comme « Ce programme ne peut pas être lancé en mode DOS. »—si vous lancez sous DOS ou Windows 3.1 (OS-es qui ne supportent pas le format PE), ce message sera affiché.

Terminologie

- Module—un fichier séparé, .exe ou .dll.
- Process—un programme chargé dans la mémoire et fonctionnant actuellement. Consiste en général d'un fichier .exe et d'un paquet de fichiers .dll.
- Process memory—la mémoire utilisée par un processus. Chaque processus a la sienne. Il y a en général des modules chargés, la mémoire pour la pile, heap(s), etc.

[16] Dynamic-Link Library
[17] Par exemple, Hiew(5.1 on page 600) peut la calculer

- VA[18]—une adresse qui est utilisée pendant le déroulement du programme.
- Base address (du module)—l'adresse dans la mémoire du processus à laquelle le module est chargé. Le chargeur de l'OS peut la changer, si l'adresse de base est occupée par un module déjà chargé.
- RVA[19]—l'adresse VA-moins l'adresse de base.

  De nombreuses adresses dans les fichiers PE utilisent l'adresse RVA.
- IAT[20]—un tableau d'adresses des symboles importés [21]. Parfois, le répertoire de données `IMAGE_DIRECTORY_E` pointe sur l'IAT. Il est utile de noter qu'IDA (à partir de 6.1) peut allouer une pseudo-section nommée `.idata` pour l'IAT, même si l'IAT fait partie d'une autre section!
- INT[22]—une tableau de noms de symboles qui doivent être importés[23].

Adresse de base

Le problème est que plusieurs auteurs de module pevent préparer des fichiers DLL pour que d'autres les utilisent et qu'il n'est pas possible de s'accorder pour assigner une adresse à ces modules.

C'est pourquoi si deux DLLs nécessaires pour un processus ont la même adresse de base, une des deux sera chargée à cette adresse de base, et l'autre—à un autre espace libre de la mémoire du processus et chaque adresse virtuelle de la seconde DLL sera corrigée.

Avec MSVC l'éditeur de lien génère souvent les fichiers .exe avec une adresse de base en `0x400000`[24], et avec une section de code débutant en `0x401000`. Cela signifie que la RVA du début de la section de code est `0x1000`.

Les DLLs sont souvent générées par l'éditeur de lien de MSVC avec une adresse de base de `0x10000000`[25].

Il y a une autre raison de charger les modules a des adresses de base variées, aléatoires dans ce cas. C'est l'ASLR[26][27].

Un shellcode essayant d'être exécuté sur une système compromis doit appeler des fonctions systèmes, par conséquent, connaitre leurs adresses.

Dans les anciens OS (dans la série Windows NT : avant Windows Vista), les DLLs système (comme kernel32.dll, user32.dll) étaient toujours chargées à une adresse connue, et si nous nous rappelons que leur version changeait rarement, l'adresse des fonctions était fixe et le shellcode pouvait les appeler directement.

Pour éviter ceci, la méthode ASLR charge votre programme et tous les modules dont il a besoin à des adresses de base aléatoires, différentes à chaque fois.

Le support de l'ASLR est indiqué dans un fichier PE en mettant le flag

`IMAGE_DLL_CHARACTERISTICS_DYNAMIC_BASE` [voir Mark Russinovich, *Microsoft Windows Internals*].

Sous-système

Il a aussi un champ *sous-système*, d'habitude c'est:

- natif[28] (.sys-driver),
- console (application en console) ou
- GUI (non-console).

[18] Virtual Address
[19] Relative Virtual Address
[20] Import Address Table
[21] Matt Pietrek, *An In-Depth Look into the Win32 Portable Executable File Format*, (2002)]
[22] Import Name Table
[23] Matt Pietrek, *An In-Depth Look into the Win32 Portable Executable File Format*, (2002)]
[24] L'origine de ce choix d'adresse est décrit ici: MSDN
[25] Cela peut être changé avec l'option /BASE de l'éditeur de liens
[26] Address Space Layout Randomization
[27] Wikipédia
[28] Signifiant que le module utilise l'API Native au lieu de Win32

Version d'OS

Un fichier PE indique aussi la version de Windows minimale requise pour pouvoir être chargé.

La table des numéros de version stockés dans un fichier PE et les codes de Windows correspondants est ici[29].

Par exemple, MSVC 2005 compile les fichiers .exe pour tourner sur Windows NT4 (version 4.00), mais MSVC 2008 ne le fait pas (les fichiers générés ont une version de 5.00, il faut au moins Windows 2000 pour les utiliser).

MSVC 2012 génère des fichiers .exe avec la version 6.00 par défaut, visant au moins Windows Vista. Toutefois, en changeant l'option du compilateur[30], il est possible de le forcer à compiler pour Windows XP.

Sections

Il semble que la division en section soit présente dans tous les formats de fichier exécutable.

C'est divisé afin de séparer le code des données, et les données—des données constantes.

- Si l'un des flags *IMAGE_SCN_CNT_CODE* ou *IMAGE_SCN_MEM_EXECUTE* est mis dans la section de code—c'est de code exécutable.
- Dans la section de données—les flags *IMAGE_SCN_CNT_INITIALIZED_DATA*, *IMAGE_SCN_MEM_READ* et *IMAGE_SCN_MEM_WRITE*.
- Dans une section vide avec des données non initialisées—*IMAGE_SCN_CNT_UNINITIALIZED_DATA*, *IMAGE_SCN_MEM_READ* et *IMAGE_SCN_MEM_WRITE*.
- Dans une section de données constantes (une qui est protégée contre l'écriture), les flags *IMAGE_SCN_CNT_IN* et *IMAGE_SCN_MEM_READ* peuvent être mis, mais pas *IMAGE_SCN_MEM_WRITE*. Un process plantera s'il essaye d'écrire dans cette section.

Chaque section dans un fichier PE peut avoir un nom, toutefois, ce n'est pas très important. Souvent (mais pas toujours) la section de code est appelée `.text`, la section de données—`.data`, la section de données constante — `.rdata` *(readable data)*. D'autres noms de section courants sont:

- `.idata`—section d'imports. IDA peut créer une pseudo-setion appelée ainsi: 4.5.2 on the previous page.
- `.edata`—section d'exports section (rare)
- `.pdata`—section contenant toutes les informations sur les exceptions dans Windows NT pout MIPS, IA64 et x64: 4.5.3 on page 594
- `.reloc`—section de relocalisation
- `.bss`—données non initialisées (BSS)
- `.tls`—stockage local d'un thread (TLS)
- `.rsrc`—ressources
- `.CRT`—peut être présente dans les fichiers binaire compilés avec des anciennes versions de MSVC

Les packeurs/chiffreurs rendent souvent les noms de sections initelligibles ou les remplacent avec des noms qui leurs sont propres.

MSVC permet de déclarer des données dans une section de n'importe quel nom [31].

Certains compilateurs et éditeurs de liens peuvent ajouter une section avec des symboles et d'autres informations de débogage (MinGW par exemple). Toutefois ce n'est pas comme cela dans les dernières versions de MSVC (des fichiers PDB séparés sont utilisés dans ce but).

Voici comment une section PE est décrite dans le fichier:

```
typedef struct _IMAGE_SECTION_HEADER {
  BYTE  Name[IMAGE_SIZEOF_SHORT_NAME];
  union {
```

[29] Wikipédia
[30] MSDN
[31] MSDN

```
    DWORD PhysicalAddress;
    DWORD VirtualSize;
  } Misc;
  DWORD VirtualAddress;
  DWORD SizeOfRawData;
  DWORD PointerToRawData;
  DWORD PointerToRelocations;
  DWORD PointerToLinenumbers;
  WORD  NumberOfRelocations;
  WORD  NumberOfLinenumbers;
  DWORD Characteristics;
} IMAGE_SECTION_HEADER, *PIMAGE_SECTION_HEADER;
```

[32]

Un mot à propos de le terminilogie: *PointerToRawData* est appelé « Offset » dans Hiew et *VirtualAddress* est appelé « RVA » ici.

Relocs (relocalisation)

AKA FIXUP-s (au moins dans Hiew).

Elles sont aussi présentes dans presque tous les formats de fichiers exécutables [33]. Les exceptions sont les bibliothèques dynamiques partagées compilées avec PIC, ou tout autre code PIC.

À quoi servent-elles?

Évidemment, les modules peuvent être chargés à différentes adresse de base, mais comment traiter les variables globales, par exemple? Elles doivent être accèdées par adresse. Une solution est le code indépendant de la position (4.4.1 on page 559). Mais ce n'est pas toujours pratique.

C'est pourquoi une table des relocalisations est présente. Elle contient les adresses des points qui doivent être corrigés en cas de chargement à une adresse de base différente.

Par exemple, il y a une variable globale à l'adresse `0x410000` et voici comment elle est accédée:

```
A1 00 00 41 00          mov         eax,[000410000]
```

L'adresse de base du module est `0x400000`, le RVA de la variable globale est `0x10000`.

Si le module est chargé à l'adresse de base `0x500000`, l'adresse réelle de la variable globale doit être `0x510000`.

Comme nous pouvons le voir, l'adresse de la variable est encodée dans l'instruction `MOV`, après l'octet `0xA1`.

C'est pourquoi l'adresse des 4 octets aprés `0xA1` est écrite dans la table de relocalisations.

Si le module est chargé à une adresse de base différente, le chargeur de l'OS parcourt toutes les adresses dans la table, trouve chaque mot de 32-bit vers lequel elles pointent, soustrait l'adresse de base d'origine (nous obtenons le RVA ici), et ajoute la nouvelle adresse de base.

Si un module est chargé à son adresse de base originale, rien ne se passe.

Toutes les variables globales sont traitées ainsi.

Relocs peut avoir différent types, toutefois, dans Windows pour processeurs x86, le type est généralement *IMAGE_REL_BASED_HIGHLOW*.

Á propos, les relocs sont plus foncés dans Hiew, par exemple: fig.1.21.

OllyDbg souligne les endroits en mémoire où sont appliqués les relocs, par exemple: fig.1.52.

Exports et imports

Comme nous le savons, tout programme exécutable doit utiliser les services de l'OS et autres bibliothèques d'une manière ou d'une autre.

[32] MSDN

[33] Même dans les fichiers .exe pour MS-DOS

On peut dire que les fonctions d'un module (en général DLL) doivent être reliées aux points de leur appel dans les autres modules (fichier .exe ou eutre DLL).

Pour cela, chaque DLL a une table d'« exports », qui consiste en les fonctions plus leur adresse dans le module.

Et chaque fichier .exe ou DLL possède des « imports », une table des fonctions requises pour l'exécution incluant une liste des noms de DLL.

Après le chargement du fichier .exe principal, le chargeur de l'OS traite la table des imports: il charge les fichiers DLL additionnels, trouve les noms des fonctions parmis les exports des DLL et écrit leur adresse dans le module IAT de l'.exe principal.

Comme on le voit, pendant le chargement, le chargeur doit comparer de nombreux noms de fonctions, mais la comparaison des chaînes n'est pas une procédure très rapide, donc il y a un support pour « ordinaux » ou « hints », qui sont les numéros de fonctions stockés dans une table au lieu de leurs noms.

C'est ainsi qu'elles peuvent être localisées plus rapidement lors du chargement d'une DLL. Les ordinaux sont toujours présents dans la table d'« export ».

Par exemple, un programme utilisant la bibliothèque MFC[34] charge en général mfc*.dll par ordinaux, et dans un programme n'ayant aucun nom de fonction MFC dans INT.

Lors du chargement d'un tel programme dans IDA, il va demander le chemin vers les fichiers mfc*.dll, afin de déterminer le nom des fonctions.

Si vous ne donnez pas le chemin vers ces DLLs à IDA, il y aura *mfc80_123* à la place des noms de fonctions.

Section d'imports

Souvent, une section séparée est allouée pour la table d'imports et tout ce qui y est relatif (avec un nom comme `.idata`), toutefois, ce n'est pas une règle stricte.

Les imports sont un sujet prêtant à confusion à cause de la terminologie confuse. Essayons de regrouper toutes les informations au même endroit.

[34] Microsoft Foundation Classes

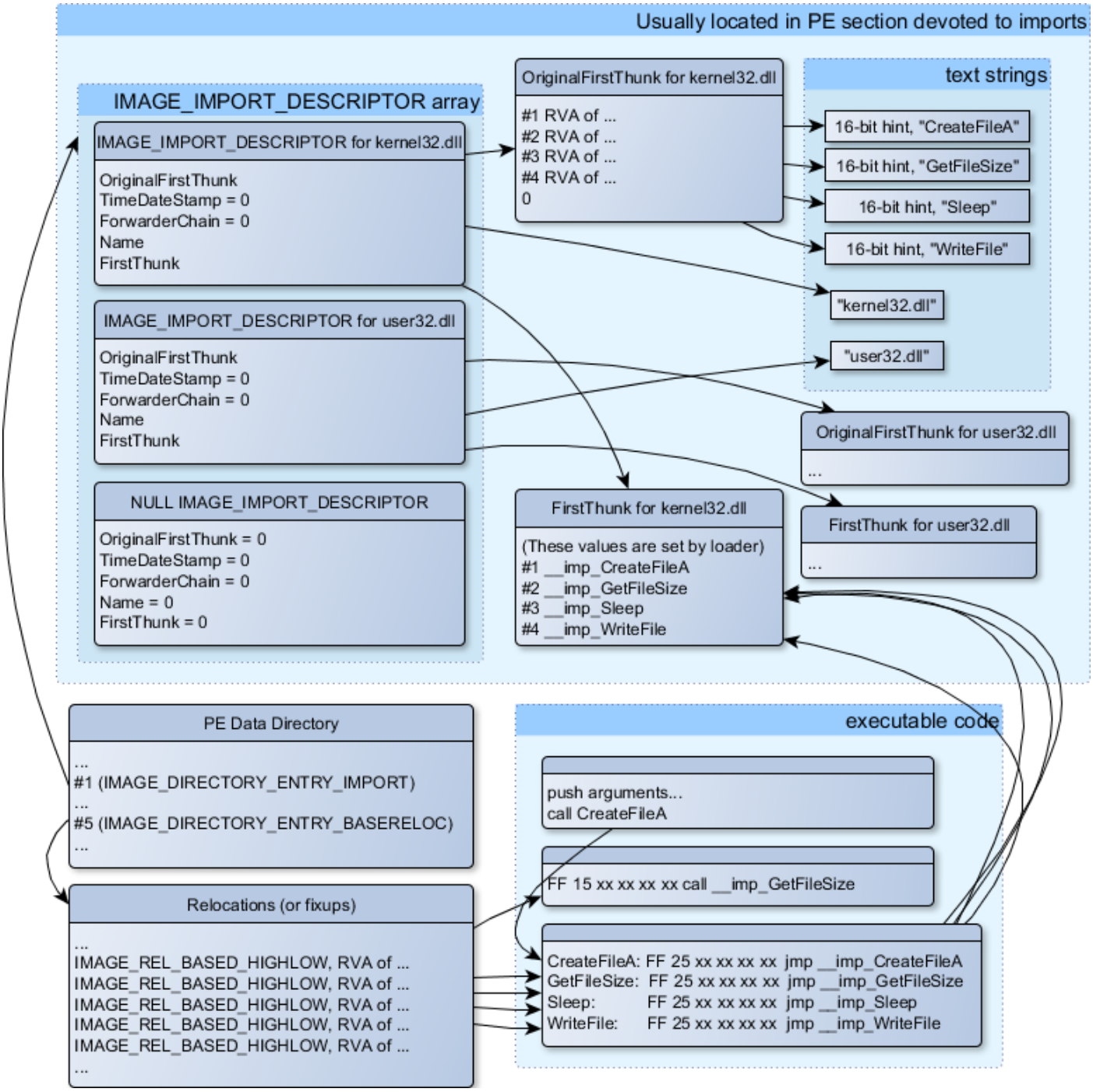


Fig. 4.1: Un schéma qui regroupe toutes les structures concernant les imports d'un fichier PE

La structure principale est le tableau *IMAGE_IMPORT_DESCRIPTOR*, qui comprend un élément pour chaque DLL importée.

Chaque élément contient l'adresse RVA de la chaîne de texte (nom de la DLL) (*Name*).

OriginalFirstThunk est l'adresse RVA de la table INT. C'est un tableau d'adresses RVA, qui pointe chacune sur une chaîne de texte avec le nom d'une fonction. Chaque chaîne est préfixée par un entier 16-bit (« hint »)—« ordinal » de la fonction.

Pendant le chargement, s'il est possible de trouver une fonction par ordinal, la comparaison de chaînes n'a pas lieu. Le tableau est terminé par zéro.

Il y a aussi un pointeur sur la table IAT appelé *FirstThunk*, qui est juste l'adresse RVA de l'endroit où le chargeur écrit l'adresse résolue de la fonction.

Les endroits où le chargeur écrit les adresses sont marqués par IDA comme ceci: *__imp_CreateFileA*, etc.

Il y a au moins deux façons d'utiliser les adresses écritent par le chargeur.

- Le code aura des instructions comme *call __imp_CreateFileA*, et puisque le champ avec l'adresse de la fonction importée est en un certain sens une variable globale, l'adresse de l'instruction *call* (plus 1 ou 2) doit être ajoutée à la table de relocs, pour le cas où le module est chargé à une adresse de base différente.

  Mais, évidemment, ceci augmente significativement la table de relocs.

  Car il peut y avoir un grand nombre d'appels aux fonctions importées dans le module.

  En outre, une grosse table de relocs ralenti le processus de chargement des modules.

- Pour chaque fonction importée, il y a seulement un saut alloué, en utilisant l'instruction `JMP` ainsi qu'un reloc sur lui. De tel point sont aussi appelés « thunks ».

  Tous les appels aux fonction importées sont juste des instructions `CALL` au « thunk » correspondant. Dans ce cas, des relocs supplémentaires ne sont pas nécessaire car cex `CALL`-s ont des adresses relatives et ne doivent pas être corrigés.

Ces deux méthodes peuvent être combinées.

Il est possible que l'éditeur de liens créé des « thunk »s individuels si il n'y a pas trop d'appels à la fonction, mais ce n'est pas fait par défaut

Á propos, le tableau d'adresses de fonction sur lequel pointe FirstThunk ne doit pas nécessairement se trouver dans la section IAT. Par exemple, j'ai écrit une fois l'utilitaire PE_add_import[35] pour ajouter des imports à un fichier .exe existant.

Quelques temps plus tôt, dans une version précédente de cet utilitaire, à la place de la fonction que vous vouliez substituer par un appel à une autre DLL, mon utilitaire écrivait le code suivant:

```
MOV EAX, [yourdll.dll!function]
JMP EAX
```

FirstThunk pointe sur la première instruction. En d'autres mots, en chargeant yourdll.dll, le chargeur écrit l'adresse de la fonction *function* juste après le code.

Il est à noter qu'une section de code est en général protégée contre l'écriture, donc mon utilitaire ajoute le flag
IMAGE_SCN_MEM_WRITE pour la section de code. Autrement, le programme planterait pendant le chargement avec le code d'erreur 5 (access denied).

On pourrait demander: pourrais-je fournir à un programme un ensemble de fichiers DLL qui n'est pas supposé changer (incluant les adresses des fonctions DLL), est-il possible d'accélérer le processus de chargement?

Oui, il est possible d'écrire les adresses des fonctions qui doivent être importées en avance dans le tableau FirstThunk. Le champ *Timestamp* est présent dans la structure *IMAGE_IMPORT_DESCRIPTOR*.

Si une valeur est présente ici, alors le loader compare cette valeur avec la date et l'heure du fichier DLL.

Si les valeurs sont égales, alors le loader ne fait rien, et le processus de chargement peut être plus rapide. Ceci est appelé « old-style binding » [36].

L'utilitaire BIND.EXE dans le SDK est fait pour ça. Pour accélérer le chargement de votre programme, Matt Pietrek dans Matt Pietrek, *An In-Depth Look into the Win32 Portable Executable File Format*, (2002)][37], suggère de faire le binding juste après l'installation de votre programme sur l'ordinateur de l'utilisateur final.

Les packeurs/chiffreurs de fichier PE peuvent également compresser/chiffrer la table des imports.

Dans ce cas, le loader de Windows, bien sûr, ne va pas charger les DLLs nécessaires.

Par conséquent, le packeur/chiffreur fait cela lui même, avec l'aide des fonctions *LoadLibrary()* et *GetProcAddress()*.

C'est pourquoi ces deux fonctions sont souvent présentes dans l'IAT des fichiers packés.

Dans les DLLs standards de l'installation de Windows, IAT es souvent situé juste après le début du fichier PE. Supposément, c'est fait par soucis d'optimisation.

[35] yurichev.com
[36] MSDN. Il y a aussi le « new-style binding ».
[37] Aussi disponible en http://go.yurichev.com/17318

Pendant le chargement, le fichier .exe n'est pas chargé dans la mémoire d'un seul coup (rappelez-vous ces fichiers d'installation énormes qui sont démarrés de façon douteusement rapide), ils sont « mappés », et chargés en mémoire par parties lorsqu'elles sont accèdées.

Les développeurs de Microsoft ont sûrement décidé que ça serait plus rapide.

Ressources

Les ressources dans un fichier PE sont seulement un ensemble d'icônes, d'images, de chaînes de texte, de descriptions de boîtes de dialogues.

Peut-être qu'elles ont été séparées du code principal afin de pouvoir être multilingue, et il est plus simple de prendre du texte ou une image pour la langue qui est définie dans l'OS.

Par effet de bord, elles peuvent être éditées facilement et sauvées dans le fichier exécutable, même si on n'a pas de connaissance particulière, en utilisant l'éditeur ResHack, par exemple (4.5.2).

.NET

Les programmes .NET ne sont pas compilés en code machine, mais dans un bytecode spécial. Strictement parlant, il y a du bytecode à la place du code x86 usuel dans le fichier .eexe, toutefois, le point d'entrée (OEP) point sur un petit fragment de code x86:

```
jmp         mscoree.dll!_CorExeMain
```

Le loader de .NET se trouve dans mscoree.dll, qui traite le fichier PE.

Il en était ainsi dans tous les OSes pre-Windows XP. Á partir de XP, le loader de l'OS est capable de détecter les fichiers .NET et le lance sans exécuter cette instruction JMP[38].

TLS

Cette section contient les données initialisées pour le TLS(4.2 on page 552) (si nécessaire). Lorsque qu'une nouvelle thread démarre, ses données TLS sont initialisées en utilisant les données de cette section.

Á part ça, la spécification des fichiers PE fourni aussi l'initialisation de la section TLS, aussi appelée TLS callbacks.

Si elles sont présentes, elles doivent être appelées avant que le contrôle ne soit passé au point d'entrée principal (OEP).

Ceci est largement utilisé dans les packeurs/chiffreurs de fichiers PE.

Outils

- objdump (présent dans cygwin) pour afficher toutes les structures d'un fichier PE.
- Hiew(5.1 on page 600) comme éditeur.
- pefile—bibliothèque-Python pour le traitement des fichiers PE [39].
- ResHack AKA Resource Hacker—éditeur de ressources[40].
- PE_add_import[41]—outil simple pour ajouter des symboles à la table d'imports d'un exécutable au format PE.
- PE_patcher[42]—outil simple pour patcher les exécutables PE.

[38] MSDN
[39] http://go.yurichev.com/17052
[40] http://go.yurichev.com/17052
[41] http://go.yurichev.com/17049
[42] yurichev.com

- PE_search_str_refs[43]—outil simple pour chercher une fonction dans un exécutable PE qui utilise une certaine chaîne de texte.

Autres lectures

- Daniel Pistelli—The .NET File Format [44]

4.5.3 Windows SEH

Oublions MSVC

Sous Windows, SEH concerne la gestion d'exceptions. Ce mécanisme est indépendant du langage de programmation et n'est en aucun cas spécifique ni à C++, ni à l'POO[45].

Nous nous intéressons donc au SEH indépendamment du C++ et des extensions du langage dans MSVC.

A chaque processus est associé une chaîne de gestionnaires SEH. A tout moment, chaque TIB référence le gestionnaire le plus récent de la chaîne.

Dès qu'une exception intervient (division par zéro, violation d'accès mémoire, exception explicitement déclenchée par le programe en appelant la fonction `RaiseException()` ...), l'OS retrouve dans le TIB le gestionnaire le plus récemment déclaré et l'appelle. Il lui fourni l'état de la CPU (contenu des registres ...) tels qu'ils étaient lors du déclenchement de l'exception.

Le gestionnaire examine le type de l'exception et décide de la traiter s'il la reconnaît. Sinon, il signale à l'OS qu'il passe la main et celui-ci appelle le prochain gestionnaire dans la chaîne, et ainsi de suite jusuqu'à ce qu'il trouve un gestionnaire qui soit capable de la traiter.

A la toute fin de la chaîne se trouve un gestionnaire standard qui affiche la fameuse boîte de dialogue qui informe l'utilisateur que le processus va être avorté. Ce dialogue fournit quelques informations techniques au sujet de l'état de la CPU au moment du crash, et propose de collecter des informations techniques qui seront envoyées aux développeurs de Microsoft.

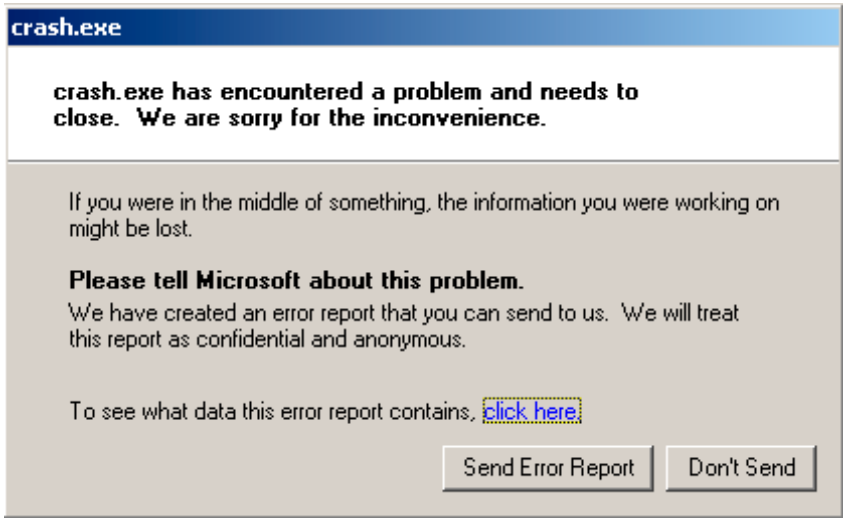


Fig. 4.2: Windows XP

[43] yurichev.com
[44] http://go.yurichev.com/17056
[45] Programmation orientée objet

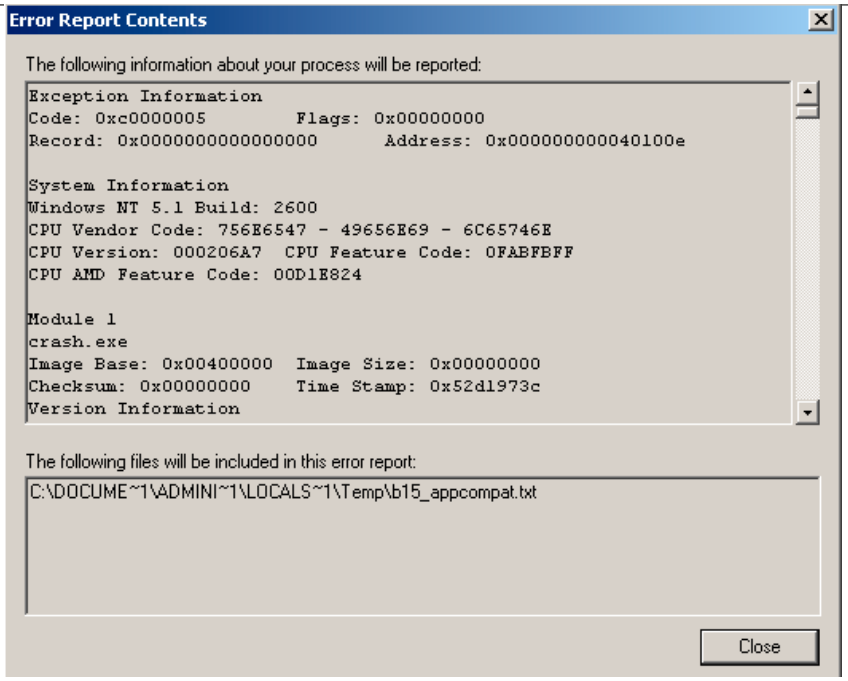


Fig. 4.3: Windows XP

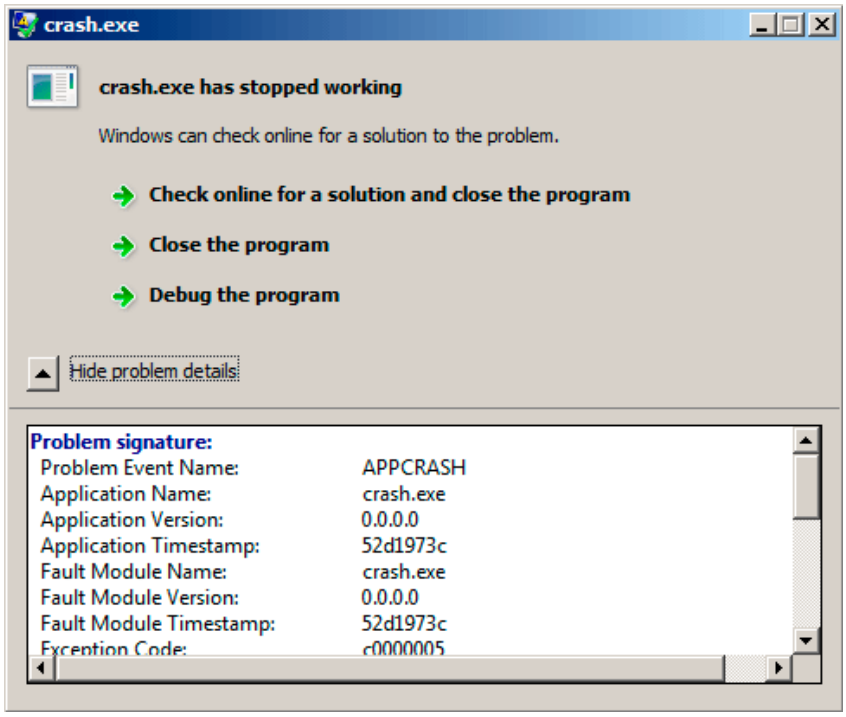


Fig. 4.4: Windows 7

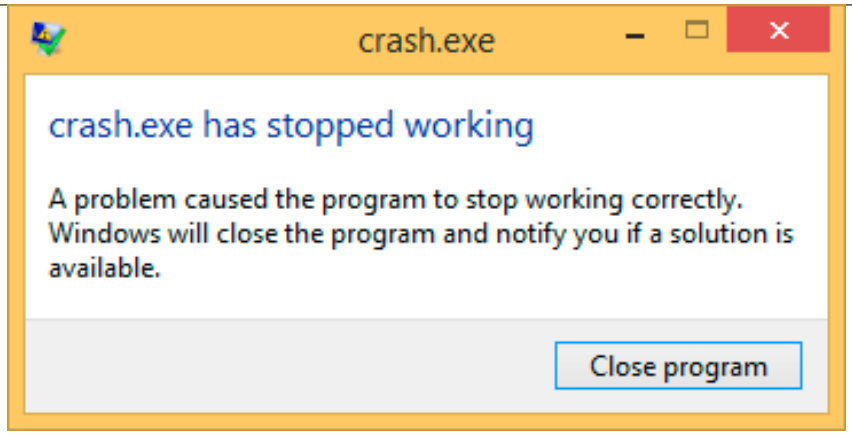


Fig. 4.5: Windows 8.1

Historiquement, cette chaîne de gestionnaires était connue sous l'appelation Dr. Watson [46].

Certains développeurs ont eu l'idée d'écrire leur propre gestionnaire d'exceptions pour recevoir les informations relatives au crash. Ils enregistrent leur gestionnaire en appelant la fonction SetUnhandledExceptionFilt qui sera alors appelée si l'OS ne trouve aucun autre gestionnaire qui souhaite gérer l'exception.

Oracle RDBMS— en est un bon exemple qui sauvegarde un énorme fichier dump collectant toutes les informations possibles concernant la CPU et l'état de la mémoire.

Ecrivons notre propre gestionnaire d'exception. Cet exemple s'appuie sur celui de [Matt Pietrek, *A Crash Course on the Depths of Win32™ Structured Exception Handling*, (1997)][47]. Pour le compiler, il faut utiliser l'option SAFESEH : `cl seh1.cpp /link /safeseh:no`. Vous trouverez plus d'informations concernant SAFESEH à: MSDN.

```
#include <windows.h>
#include <stdio.h>

DWORD new_value=1234;

EXCEPTION_DISPOSITION __cdecl except_handler(
                struct _EXCEPTION_RECORD *ExceptionRecord,
                void * EstablisherFrame,
                struct _CONTEXT *ContextRecord,
                void * DispatcherContext )
{
        unsigned i;

        printf ("%s\n", __FUNCTION__);
        printf ("ExceptionRecord->ExceptionCode=0x%p\n", ExceptionRecord->ExceptionCode);
        printf ("ExceptionRecord->ExceptionFlags=0x%p\n", ExceptionRecord->ExceptionFlags);
        printf ("ExceptionRecord->ExceptionAddress=0x%p\n", ExceptionRecord->ExceptionAddress);

        if (ExceptionRecord->ExceptionCode==0xE1223344)
        {
                printf ("That's for us\n");
                // yes, we "handled" the exception
                return ExceptionContinueExecution;
        }
        else if (ExceptionRecord->ExceptionCode==EXCEPTION_ACCESS_VIOLATION)
        {
                printf ("ContextRecord->Eax=0x%08X\n", ContextRecord->Eax);
                // will it be possible to 'fix' it?
                printf ("Trying to fix wrong pointer address\n");
                ContextRecord->Eax=(DWORD)&new_value;
                // yes, we "handled" the exception
                return ExceptionContinueExecution;
        }
        else
        {
                printf ("We do not handle this\n");
```

[46] Wikipédia
[47] Aussi disponible en http://go.yurichev.com/17293

```
                // someone else's problem
                return ExceptionContinueSearch;
        };
}

int main()
{
        DWORD handler = (DWORD)except_handler; // take a pointer to our handler

        // install exception handler
        __asm
        {                               // make EXCEPTION_REGISTRATION record :
                push    handler         // address of handler function
                push    FS :[0]          // address of previous handler
                mov     FS :[0],ESP      // add new EXECEPTION_REGISTRATION
        }

        RaiseException (0xE1223344, 0, 0, NULL);

        // now do something very bad
        int* ptr=NULL;
        int val=0;
        val=*ptr;
        printf ("val=%d\n", val);

        // deinstall exception handler
        __asm
        {                               // remove our EXECEPTION_REGISTRATION record
                mov     eax,[ESP]       // get pointer to previous record
                mov     FS :[0], EAX     // install previous record
                add     esp, 8          // clean our EXECEPTION_REGISTRATION off stack
        }

        return 0;
}
```

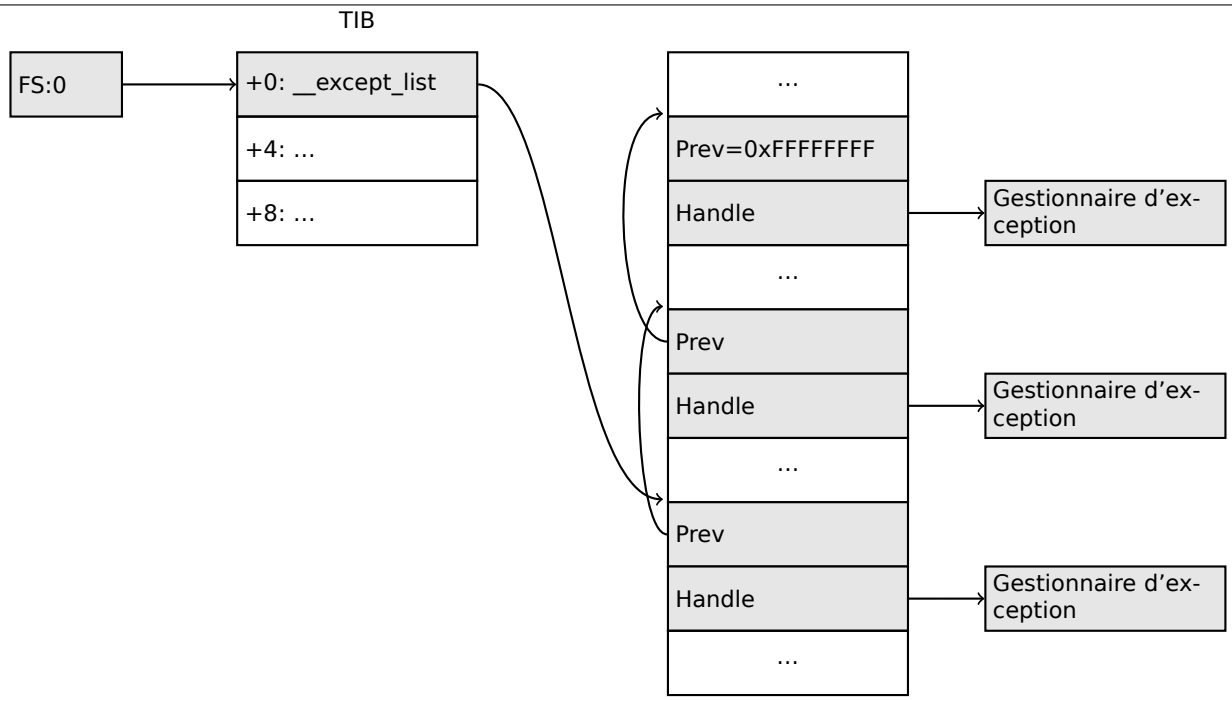
En environnement win32, le registre de segment FS: contient l'adresse du TIB.

Le tout premier élément de la structure TIB est un pointeur sur le premier gestionnaire de la chaîne de traitement des exceptions. Nous le sauvegardons sur la pile et remplaçons la valeur par celle de notre propre gestionnaire. La structure est du type _EXCEPTION_REGISTRATION. Il s'agit d'une simple liste chaînée dont les éléments sont conservés sur la pile.

Listing 4.22: MSVC/VC/crt/src/exsup.inc

```
\_EXCEPTION\_REGISTRATION struc
     prev    dd      ?
     handler dd      ?
\_EXCEPTION\_REGISTRATION ends
```

Le champ « handler » contient l'adresse du gestionnaire et le champ « prev » celle de l'enregistrement suivant dans la chaîne. Le dernier enregistremThe last record contient la valeur 0xFFFFFFFF (-1) dans son champ « prev ».



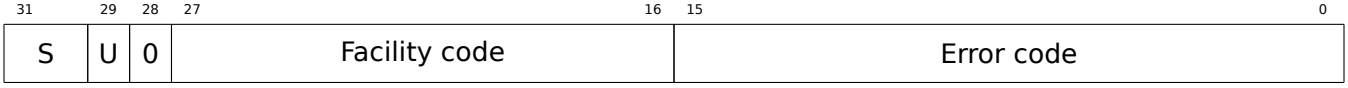
Une fois notre gestionnaire installé, nous invoquons la fonction `RaiseException()` [48]. Il s'agit d'une exception utilisateur. Le gestionnaire vérifie le code. Si le code est égal à `0xE1223344`, il retourne la valeur `ExceptionContinueExecution` qui signifie que le gestionnaire a corrigé le contenu de la structure passée en paramètre qui décrit l'état de la CPU. La modification concerne généralement les registres EIP/ESP. L'OS peut alors reprendre l'exécution du thread.

Si vous modifiez légèrement le code pour que le gestionaire retourne la valeur `ExceptionContinueSearch`, l'OS appelera les gestionnaires suivants dans la liste. Il est peu probable que l'un d'eux sache la gérer puisqu'aucun d'eux ne la comprend, ni ne connait le code exception. Vous verrez donc apparaître la boîte de dialogue Windows précurseure du crash.

Quelles sont les différences entre les exceptions système et les exceptions utilisateur ? Les exceptions système sont listées ci-dessous:

as defined in WinBase.h	as defined in ntstatus.h	value
EXCEPTION_ACCESS_VIOLATION	STATUS_ACCESS_VIOLATION	0xC0000005
EXCEPTION_DATATYPE_MISALIGNMENT	STATUS_DATATYPE_MISALIGNMENT	0x80000002
EXCEPTION_BREAKPOINT	STATUS_BREAKPOINT	0x80000003
EXCEPTION_SINGLE_STEP	STATUS_SINGLE_STEP	0x80000004
EXCEPTION_ARRAY_BOUNDS_EXCEEDED	STATUS_ARRAY_BOUNDS_EXCEEDED	0xC000008C
EXCEPTION_FLT_DENORMAL_OPERAND	STATUS_FLOAT_DENORMAL_OPERAND	0xC000008D
EXCEPTION_FLT_DIVIDE_BY_ZERO	STATUS_FLOAT_DIVIDE_BY_ZERO	0xC000008E
EXCEPTION_FLT_INEXACT_RESULT	STATUS_FLOAT_INEXACT_RESULT	0xC000008F
EXCEPTION_FLT_INVALID_OPERATION	STATUS_FLOAT_INVALID_OPERATION	0xC0000090
EXCEPTION_FLT_OVERFLOW	STATUS_FLOAT_OVERFLOW	0xC0000091
EXCEPTION_FLT_STACK_CHECK	STATUS_FLOAT_STACK_CHECK	0xC0000092
EXCEPTION_FLT_UNDERFLOW	STATUS_FLOAT_UNDERFLOW	0xC0000093
EXCEPTION_INT_DIVIDE_BY_ZERO	STATUS_INTEGER_DIVIDE_BY_ZERO	0xC0000094
EXCEPTION_INT_OVERFLOW	STATUS_INTEGER_OVERFLOW	0xC0000095
EXCEPTION_PRIV_INSTRUCTION	STATUS_PRIVILEGED_INSTRUCTION	0xC0000096
EXCEPTION_IN_PAGE_ERROR	STATUS_IN_PAGE_ERROR	0xC0000006
EXCEPTION_ILLEGAL_INSTRUCTION	STATUS_ILLEGAL_INSTRUCTION	0xC000001D
EXCEPTION_NONCONTINUABLE_EXCEPTION	STATUS_NONCONTINUABLE_EXCEPTION	0xC0000025
EXCEPTION_STACK_OVERFLOW	STATUS_STACK_OVERFLOW	0xC00000FD
EXCEPTION_INVALID_DISPOSITION	STATUS_INVALID_DISPOSITION	0xC0000026
EXCEPTION_GUARD_PAGE	STATUS_GUARD_PAGE_VIOLATION	0x80000001
EXCEPTION_INVALID_HANDLE	STATUS_INVALID_HANDLE	0xC0000008
EXCEPTION_POSSIBLE_DEADLOCK	STATUS_POSSIBLE_DEADLOCK	0xC0000194
CONTROL_C_EXIT	STATUS_CONTROL_C_EXIT	0xC000013A

Le code de chaque exception se décompose comme suit:



[48] MSDN

S est un code status de base: 11—erreur; 10—warning; 01—information; 00—succès. U—lorsqu'il s'agit d'un code utilisateur.

Voici pourquoi nous choisissons le code 0xE1223344—E_{16} (1110_2) 0xE (1110b) qui signifie 1) qu'il s'agit d'une exception utilisateur; 2) qu'il s'agit d'une erreur.

Pour être honnête, l'exemple fonctionne aussi bien sans ces bits de poids fort.

Tentons maintenant de lire la valeur à l'adresse mémoire 0.

Bien entendu, dans win32 il n'existe rien à cette adresse, ce qui déclenche une exception.

Le premier gestionnaire à être invoqué est le vôtre. Il reconnaît l'exception car il compare le code avec celui de la constante EXCEPTION_ACCESS_VIOLATION.

Le code qui lit la mémoire à l'adresse 0 ressemble à ceci:

Listing 4.23: MSVC 2010

```
        ...
        xor     eax, eax
        mov     eax, DWORD PTR [eax]  ; exception will occur here
        push    eax
        push    OFFSET msg
        call    _printf
        add     esp, 8
        ...
```

Serait-il possible de corriger cette erreur « on the fly » afin de continuer l'exécution du programme?

Notre gestionnaire d'exception peut modifier la valeur du registre EAX puis laisser l'OS exécuter de nouveau l'instruction fautive. C'est ce que nous faisons et la raison pour laquelle `printf()` affiche 1234. Lorsque notre gestionnaire a fini son travail, la valeur de EAX n'est plus 0 mais l'adresse de la variable globale `new_value`. L'exécution du programme se poursuit donc.

Voici ce qui se passe: le gestionnaire mémoire de la CPU signale une erreur, la CPU suspend le thread, trouve le gestionnaire d'exception dans le noyau Windows, lequel à son tour appelle les gestionnaires de la chaîne SEH un par un.

Nous utilisons ici le compilateur MSVC 2010. Bien entendu, il n'y a aucune garantie que celui-ci décide d'utiliser le registre EAX pour conserver la valeur du pointeur.

Le truc du remplacement du contenu du registre n'est qu'une illustration de ce que peut être le fonctionnement interne des SEH. En pratique, il est très rare qu'il soit utilisé pour corriger « on-the-fly » une erreur.

Pourquoi les enregistrements SEH sont-ils conservés directement sur la pile et non pas à un autre endroit?

L'explication la plus plausible est que l'OS n'a ainsi pas besoin de libérer l'espace qu'ils utilisent. Ces enregistrements sont automatiquement supprimés lorsque la fonction se termine. C'est un peu comme la fonction alloca(): (1.7.2 on page 35).

Retour à MSVC

Microsoft a ajouté un mécanisme non standard de gestion d'exceptions à MSVC[49] essentiellement à l'usage des programmeurs C. Ce mécanisme est totalement distinct de celui définit par le standard ISO du langage C++.

```
__try
{
    ...
}
__except(filter code)
{
    handler code
}
```

A la place du gestionnaire d'exception, on peut trouver un block « Finally » :

[49] MSDN

```
__try
{
    ...
}
__finally
{
    ...
}
```

Le code de filtrage est une expression. L'évaluation de celle-ci permet de définir si le gestionnaire reconnaît l'exception qui a été déclenchée.

Si votre filtre est trop complexe pour tenir dans une seule expression, une fonction de filtrage séparée peut être définie.

Il existe de nombreuses constructions de ce type dans le noyau Windows. En voici quelques exemples (WRK[50]):

Listing 4.24: WRK-v1.2/base/ntos/ob/obwait.c

```
try {

    KeReleaseMutant( (PKMUTANT)SignalObject,
                     MUTANT_INCREMENT,
                     FALSE,
                     TRUE );

} except((GetExceptionCode () == STATUS_ABANDONED ||
          GetExceptionCode () == STATUS_MUTANT_NOT_OWNED)?
             EXCEPTION_EXECUTE_HANDLER  :
             EXCEPTION_CONTINUE_SEARCH) {
    Status = GetExceptionCode();

    goto WaitExit;
}
```

Listing 4.25: WRK-v1.2/base/ntos/cache/cachesub.c

```
try {

    RtlCopyBytes( (PVOID)((PCHAR)CacheBuffer + PageOffset),
                  UserBuffer,
                  MorePages ?
                    (PAGE_SIZE - PageOffset)  :
                    (ReceivedLength - PageOffset) );

} except( CcCopyReadExceptionFilter( GetExceptionInformation(),
                                                          &Status ) ) {
```

Voici aussi un exemple de code de filtrage:

Listing 4.26: WRK-v1.2/base/ntos/cache/copysup.c

```
LONG
CcCopyReadExceptionFilter(
    IN PEXCEPTION_POINTERS ExceptionPointer,
    IN PNTSTATUS ExceptionCode
    )

/*++

Routine Description :

    This routine serves as an exception filter and has the special job of
    extracting the "real" I/O error when Mm raises STATUS_IN_PAGE_ERROR
    beneath us.

Arguments :
```

[50] Windows Research Kernel

```
    ExceptionPointer - A pointer to the exception record that contains
                       the real Io Status.

    ExceptionCode - A pointer to an NTSTATUS that is to receive the real
                    status.

Return Value :

    EXCEPTION_EXECUTE_HANDLER

--*/

{
    *ExceptionCode = ExceptionPointer->ExceptionRecord->ExceptionCode;

    if ( (*ExceptionCode == STATUS_IN_PAGE_ERROR) &&
         (ExceptionPointer->ExceptionRecord->NumberParameters >= 3) ) {

        *ExceptionCode = (NTSTATUS) ExceptionPointer->ExceptionRecord->ExceptionInformation[2];
    }

    ASSERT( !NT_SUCCESS(*ExceptionCode) );

    return EXCEPTION_EXECUTE_HANDLER;
}
```

En interne, SEH est une extension du mécanisme de gestion des exceptions implémenté par l'OS. La fonction de gestion d'exceptions est `_except_handler3` (pour SEH3) ou `_except_handler4` (pour SEH4).

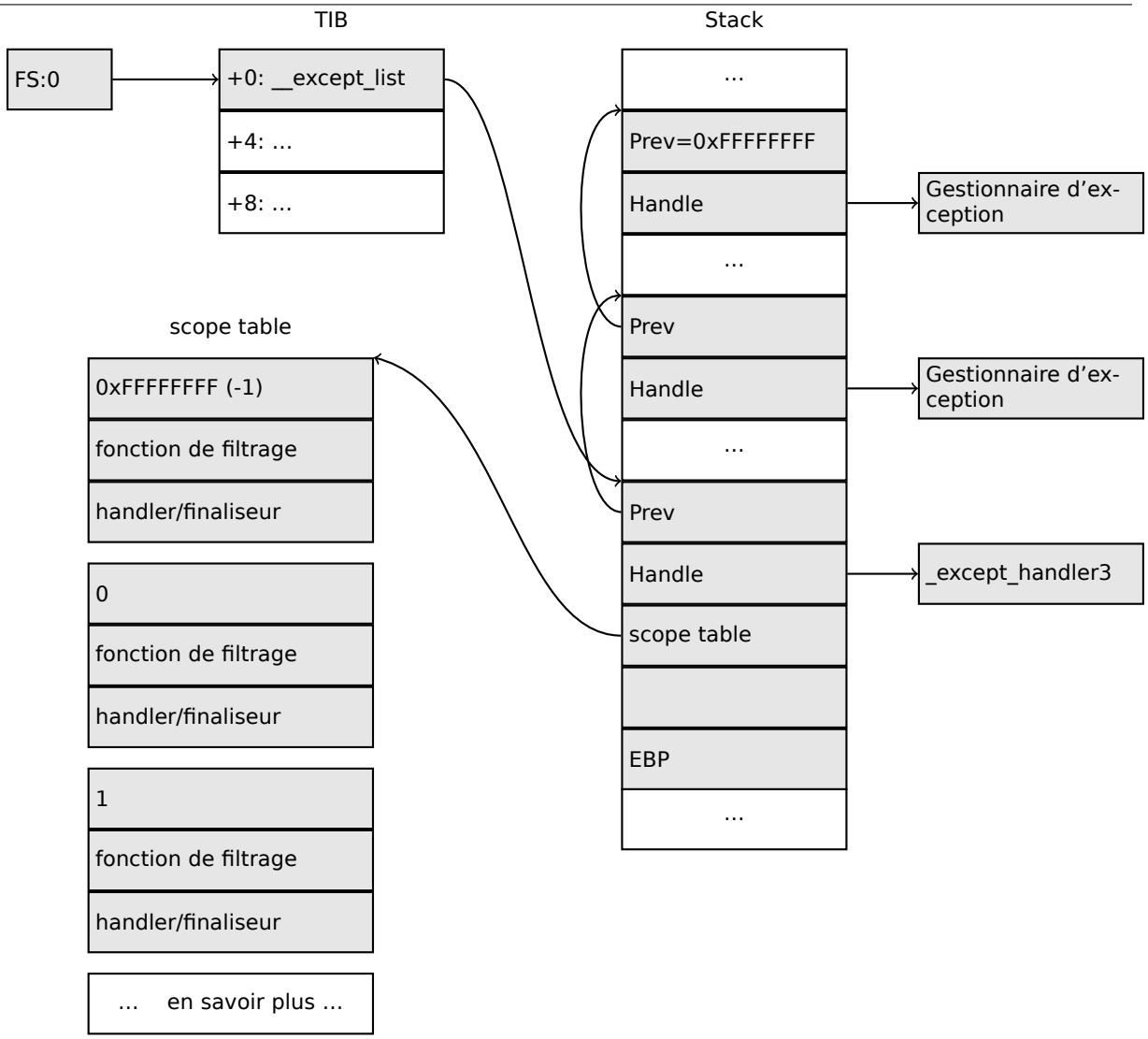
Le code de ce gestionnaire est propre à MSVC et est situé dans ses librairies, ou dans msvcr*.dll. Il est essentiel de comprendre que SEH est purement lié à MSVC.

D'autres compilateurs win32 peuvent choisir un modèle totalement différent.

SEH3

SEH3 est géré par la fonction `_except_handler3`. Il ajoute à la structure `_EXCEPTION_REGISTRATION` un pointeur vers une *scope table* et une variable *previous try level*. SEH4 de son côté ajoute 4 valeurs à la structure *scope table* pour la gestion des dépassements de buffer.

La structure *scope table* est un ensemble de pointeurs vers les blocs de code du filtre et du gestionnaire de chaque niveau *try/except* imbriqué.



Il est essentiel de comprendre que l'OS ne se préoccupe que des champs *prev/handle* et de rien d'autre. Les autres champs sont exploités par la fonction `_except_handler3`, de même que le contenu de la structure *scope table* afin de décider quel gestionnaire exécuter et quand.

Le code source de la fonction `_except_handler3` n'est pas public.

Cependant, le système d'exploitation Sanos, possède un mode de compatibilité win32. Celui-ci réimplémente les mêmes fonctions d'une manière quasi équivalente à celle de Windows [51]. On trouve une autre réimplementation dans Wine[52] ainsi que dans ReactOS[53].

Lorsque le champ *filter* est un pointeur NULL, le champ *handler* est un pointeur vers un bloc de code *finally*.

Au cours de l'exécution, la valeur du champ *previous try level* change. Ceci permet à la fonction `_except_handler` de connaître le niveau d'imbrication et donc de savoir quelle entrée de la table *scope table* utiliser en cas d'exception.

SEH3: exemple de bloc try/except

```
#include <stdio.h>
#include <windows.h>
#include <excpt.h>

int main()
{
```

[51] http://go.yurichev.com/17058
[52] GitHub
[53] http://go.yurichev.com/17060

```
    int* p = NULL;
    __try
    {
        printf("hello #1!\n");
        *p = 13;    // causes an access violation exception;
        printf("hello #2!\n");
    }
    __except(GetExceptionCode()==EXCEPTION_ACCESS_VIOLATION ?
            EXCEPTION_EXECUTE_HANDLER  : EXCEPTION_CONTINUE_SEARCH)
    {
        printf("access violation, can't recover\n");
    }
}
```

Listing 4.27: MSVC 2003

```
$SG74605 DB    'hello #1!', 0aH, 00H
$SG74606 DB    'hello #2!', 0aH, 00H
$SG74608 DB    'access violation, can''t recover', 0aH, 00H
_DATA    ENDS

; scope table :
CONST     SEGMENT
$T74622   DD    0ffffffffH    ; previous try level
          DD    FLAT :$L74617  ; filter
          DD    FLAT :$L74618  ; handler

CONST    ENDS
_TEXT    SEGMENT
$T74621 = -32 ; size = 4
_p$ = -28     ; size = 4
__$SEHRec$ = -24 ; size = 24
_main    PROC NEAR
    push   ebp
    mov    ebp, esp
    push   -1                                ; previous try level
    push   OFFSET FLAT :$T74622              ; scope table
    push   OFFSET FLAT :__except_handler3    ; handler
    mov    eax, DWORD PTR fs :__except_list
    push   eax                               ; prev
    mov    DWORD PTR fs :__except_list, esp
    add    esp, -16
; 3 registers to be saved :
    push   ebx
    push   esi
    push   edi
    mov    DWORD PTR __$SEHRec$[ebp], esp
    mov    DWORD PTR _p$[ebp], 0
    mov    DWORD PTR __$SEHRec$[ebp+20], 0  ; previous try level
    push   OFFSET FLAT :$SG74605 ; 'hello #1!'
    call   _printf
    add    esp, 4
    mov    eax, DWORD PTR _p$[ebp]
    mov    DWORD PTR [eax], 13
    push   OFFSET FLAT :$SG74606 ; 'hello #2!'
    call   _printf
    add    esp, 4
    mov    DWORD PTR __$SEHRec$[ebp+20], -1  ; previous try level
    jmp    SHORT $L74616

    ; filter code :
$L74617 :
$L74627 :
    mov    ecx, DWORD PTR __$SEHRec$[ebp+4]
    mov    edx, DWORD PTR [ecx]
    mov    eax, DWORD PTR [edx]
    mov    DWORD PTR $T74621[ebp], eax
    mov    eax, DWORD PTR $T74621[ebp]
    sub    eax, -1073741819; c0000005H
    neg    eax
```

```
    sbb    eax, eax
    inc    eax
$L74619 :
$L74626 :
    ret    0

    ; handler code :
$L74618 :
    mov    esp, DWORD PTR __$SEHRec$[ebp]
    push   OFFSET FLAT :$SG74608 ; 'access violation, can''t recover'
    call   _printf
    add    esp, 4
    mov    DWORD PTR __$SEHRec$[ebp+20], -1 ; setting previous try level back to -1
$L74616 :
    xor    eax, eax
    mov    ecx, DWORD PTR __$SEHRec$[ebp+8]
    mov    DWORD PTR fs :__except_list, ecx
    pop    edi
    pop    esi
    pop    ebx
    mov    esp, ebp
    pop    ebp
    ret    0
_main    ENDP
_TEXT    ENDS
END
```

Nous voyons ici la manière dont le bloc SEH est construit sur la pile. La structure *scope table* est présente dans le segment CONST du programme— ce qui est normal puisque son contenu n'a jamais besoin d'être changé. Un point intéressant est la manière dont la valeur de la variable *previous try level* évolue. Sa valeur initiale est 0xFFFFFFFF (−1). L'entrée dans le bloc `try` débute par l'écriture de la valeur 0 dans la variable. La sortie du bloc `try` est marquée par la restauration de la valeur −1. Nous voyons également l'adresse du bloc de filtrage et de celui du gestionnaire.

Nous pouvons donc observer facilement la présence de blocs *try/except* dans la fonction.

Le code d'initialisation des structures SEH dans le prologue de la fonction peut être partagé par de nombreuses fonctions. Le compilateur choisi donc parfois d'insérer dans le prologue d'une fonction un appel à la fonction `SEH_prolog()` qui assure cette initialisation.

Le code de nettoyage des structures SEH se trouve quant à lui dans la fonction `SEH_epilog()`.

Tentons d'exécuter cet exemple dans tracer :

```
tracer.exe -l :2.exe --dump-seh
```

Listing 4.28: tracer.exe output

```
EXCEPTION_ACCESS_VIOLATION at 2.exe!main+0x44 (0x401054) ExceptionInformation[0]=1
EAX=0x00000000 EBX=0x7efde000 ECX=0x0040cbc8 EDX=0x0008e3c8
ESI=0x00001db1 EDI=0x00000000 EBP=0x0018feac ESP=0x0018fe80
EIP=0x00401054
FLAGS=AF IF RF
* SEH frame at 0x18fe9c prev=0x18ff78 handler=0x401204 (2.exe!_except_handler3)
SEH3 frame. previous trylevel=0
scopetable entry[0]. previous try level=-1, filter=0x401070 (2.exe!main+0x60) handler=0x401088 ⬋
    ↳ (2.exe!main+0x78)
* SEH frame at 0x18ff78 prev=0x18ffc4 handler=0x401204 (2.exe!_except_handler3)
SEH3 frame. previous trylevel=0
scopetable entry[0]. previous try level=-1, filter=0x401531 (2.exe!mainCRTStartup+0x18d) ⬋
    ↳ handler=0x401545 (2.exe!mainCRTStartup+0x1a1)
* SEH frame at 0x18ffc4 prev=0x18ffe4 handler=0x771f71f5 (ntdll.dll!__except_handler4)
SEH4 frame. previous trylevel=0
SEH4 header :    GSCookieOffset=0xfffffffe GSCookieXOROffset=0x0
                 EHCookieOffset=0xffffffcc EHCookieXOROffset=0x0
scopetable entry[0]. previous try level=-2, filter=0x771f74d0 (ntdll.dll!⬋
    ↳ ___safe_se_handler_table+0x20) handler=0x771f90eb (ntdll.dll!_TppTerminateProcess@4+0x43)
* SEH frame at 0x18ffe4 prev=0xffffffff handler=0x77247428 (ntdll.dll!_FinalExceptionHandler@16⬋
    ↳ )
```

Nous constatons que la chaîne SEH est constituée de 4 gestionnaires.

Le deux premiers sont situés dans le code de notre exemple. Deux? Mais nous n'en avons défini qu'un! Effectivement, mais un second a été initialisé dans la fonction `_mainCRTStartup()` du CRT. Il semble que celui-ci gère au moins les exceptions FPU. Son code source figure dans le fichier `crt/src/winxfltr.c` fournit avec l'installation de MSVC.

Le troisième est le gestionnaire SEH4 dans ntdll.dll. Le quatrième n'est pas lié à MSVC et se situe dans ntdll.dll. Son nom suffit à en décrire l'utilité.

Comme vous le constatez, nous avons 3 types de gestionnaire dans la même chaîne:

L'un n'a rien à voir avec MSVC (le dernier) et deux autres sont liés à MSVC: SEH3 et SEH4.

SEH3: exemple de deux blocs try/except

```
#include <stdio.h>
#include <windows.h>
#include <excpt.h>

int filter_user_exceptions (unsigned int code, struct _EXCEPTION_POINTERS *ep)
{
    printf("in filter. code=0x%08X\n", code);
    if (code == 0x112233)
    {
        printf("yes, that is our exception\n");
        return EXCEPTION_EXECUTE_HANDLER;
    }
    else
    {
        printf("not our exception\n");
        return EXCEPTION_CONTINUE_SEARCH;
    };
}
int main()
{
    int* p = NULL;
    __try
    {
        __try
        {
            printf ("hello!\n");
            RaiseException (0x112233, 0, 0, NULL);
            printf ("0x112233 raised. now let's crash\n");
            *p = 13;    // causes an access violation exception;
        }
        __except(GetExceptionCode()==EXCEPTION_ACCESS_VIOLATION ?
                EXCEPTION_EXECUTE_HANDLER  : EXCEPTION_CONTINUE_SEARCH)
        {
            printf("access violation, can't recover\n");
        }
    }
    __except(filter_user_exceptions(GetExceptionCode(), GetExceptionInformation()))
    {
        // the filter_user_exceptions() function answering to the question
        // "is this exception belongs to this block?"
        // if yes, do the follow :
        printf("user exception caught\n");
    }
}
```

Nous avons maintenant deux blocs `try`. La structure *scope table* possède donc deux entrées, une pour chaque bloc. La valeur de *Previous try level* change selon que l'exécution entre ou sort des blocs `try`.

Listing 4.29: MSVC 2003

```
$SG74606 DB    'in filter. code=0x%08X', 0aH, 00H
$SG74608 DB    'yes, that is our exception', 0aH, 00H
```

```
$SG74610 DB    'not our exception', 0aH, 00H
$SG74617 DB    'hello!', 0aH, 00H
$SG74619 DB    '0x112233 raised. now let''s crash', 0aH, 00H
$SG74621 DB    'access violation, can''t recover', 0aH, 00H
$SG74623 DB    'user exception caught', 0aH, 00H

_code$ = 8   ; size = 4
_ep$ = 12    ; size = 4
_filter_user_exceptions PROC NEAR
    push   ebp
    mov    ebp, esp
    mov    eax, DWORD PTR _code$[ebp]
    push   eax
    push   OFFSET FLAT :$SG74606 ; 'in filter. code=0x%08X'
    call   _printf
    add    esp, 8
    cmp    DWORD PTR _code$[ebp], 1122867; 00112233H
    jne    SHORT $L74607
    push   OFFSET FLAT :$SG74608 ; 'yes, that is our exception'
    call   _printf
    add    esp, 4
    mov    eax, 1
    jmp    SHORT $L74605
$L74607 :
    push   OFFSET FLAT :$SG74610 ; 'not our exception'
    call   _printf
    add    esp, 4
    xor    eax, eax
$L74605 :
    pop    ebp
    ret    0
_filter_user_exceptions ENDP

; scope table :
CONST    SEGMENT
$T74644   DD    0ffffffffH   ; previous try level for outer block
          DD    FLAT :$L74634 ; outer block filter
          DD    FLAT :$L74635 ; outer block handler
          DD    00H          ; previous try level for inner block
          DD    FLAT :$L74638 ; inner block filter
          DD    FLAT :$L74639 ; inner block handler
CONST    ENDS

$T74643 = -36     ; size = 4
$T74642 = -32     ; size = 4
_p$ = -28         ; size = 4
__$SEHRec$ = -24  ; size = 24
_main    PROC NEAR
    push   ebp
    mov    ebp, esp
    push   -1 ; previous try level
    push   OFFSET FLAT :$T74644
    push   OFFSET FLAT :__except_handler3
    mov    eax, DWORD PTR fs :__except_list
    push   eax
    mov    DWORD PTR fs :__except_list, esp
    add    esp, -20
    push   ebx
    push   esi
    push   edi
    mov    DWORD PTR __$SEHRec$[ebp], esp
    mov    DWORD PTR _p$[ebp], 0
    mov    DWORD PTR __$SEHRec$[ebp+20], 0 ; outer try block entered. set previous try level to↙
    ↳ 0
    mov    DWORD PTR __$SEHRec$[ebp+20], 1 ; inner try block entered. set previous try level to↙
    ↳ 1
    push   OFFSET FLAT :$SG74617 ; 'hello!'
    call   _printf
    add    esp, 4
    push   0
```

```
    push   0
    push   0
    push   1122867    ; 00112233H
    call   DWORD PTR __imp__RaiseException@16
    push   OFFSET FLAT :$SG74619 ; '0x112233 raised. now let''s crash'
    call   _printf
    add    esp, 4
    mov    eax, DWORD PTR _p$[ebp]
    mov    DWORD PTR [eax], 13
    mov    DWORD PTR __$SEHRec$[ebp+20], 0 ; inner try block exited. set previous try level ⬋
    ↳ back to 0
    jmp    SHORT $L74615

; inner block filter :
$L74638 :
$L74650 :
    mov    ecx, DWORD PTR __$SEHRec$[ebp+4]
    mov    edx, DWORD PTR [ecx]
    mov    eax, DWORD PTR [edx]
    mov    DWORD PTR $T74643[ebp], eax
    mov    eax, DWORD PTR $T74643[ebp]
    sub    eax, -1073741819; c0000005H
    neg    eax
    sbb    eax, eax
    inc    eax
$L74640 :
$L74648 :
    ret    0

; inner block handler :
$L74639 :
    mov    esp, DWORD PTR __$SEHRec$[ebp]
    push   OFFSET FLAT :$SG74621 ; 'access violation, can''t recover'
    call   _printf
    add    esp, 4
    mov    DWORD PTR __$SEHRec$[ebp+20], 0 ; inner try block exited. set previous try level ⬋
    ↳ back to 0

$L74615 :
    mov    DWORD PTR __$SEHRec$[ebp+20], -1 ; outer try block exited, set previous try level ⬋
    ↳ back to -1
    jmp    SHORT $L74633

; outer block filter :
$L74634 :
$L74651 :
    mov    ecx, DWORD PTR __$SEHRec$[ebp+4]
    mov    edx, DWORD PTR [ecx]
    mov    eax, DWORD PTR [edx]
    mov    DWORD PTR $T74642[ebp], eax
    mov    ecx, DWORD PTR __$SEHRec$[ebp+4]
    push   ecx
    mov    edx, DWORD PTR $T74642[ebp]
    push   edx
    call   _filter_user_exceptions
    add    esp, 8
$L74636 :
$L74649 :
    ret    0

; outer block handler :
$L74635 :
    mov    esp, DWORD PTR __$SEHRec$[ebp]
    push   OFFSET FLAT :$SG74623 ; 'user exception caught'
    call   _printf
    add    esp, 4
    mov    DWORD PTR __$SEHRec$[ebp+20], -1 ; both try blocks exited. set previous try level ⬋
    ↳ back to -1
$L74633 :
    xor    eax, eax
```

```
    mov    ecx, DWORD PTR __$SEHRec$[ebp+8]
    mov    DWORD PTR fs :__except_list, ecx
    pop    edi
    pop    esi
    pop    ebx
    mov    esp, ebp
    pop    ebp
    ret    0
_main    ENDP
```

Si nous positionnons un point d'arrêt sur la fonction `printf()` qui est appelée par le gestionnaire, nous pouvons constater comment un nouveau gestionnaire SEH est ajouté.

Il s'agit peut-être d'un autre mécanisme interne de la gestion SEH. Nous constatons aussi que notre structure *scope table* contient 2 entrées.

```
tracer.exe -l :3.exe bpx=3.exe!printf --dump-seh
```

Listing 4.30: tracer.exe output

```
(0) 3.exe!printf
EAX=0x0000001b EBX=0x00000000 ECX=0x0040cc58 EDX=0x0008e3c8
ESI=0x00000000 EDI=0x00000000 EBP=0x0018f840 ESP=0x0018f838
EIP=0x004011b6
FLAGS=PF ZF IF
* SEH frame at 0x18f88c prev=0x18fe9c handler=0x771db4ad (ntdll.dll!ExecuteHandler2@20+0x3a)
* SEH frame at 0x18fe9c prev=0x18ff78 handler=0x4012e0 (3.exe!_except_handler3)
SEH3 frame. previous trylevel=1
scopetable entry[0]. previous try level=-1, filter=0x401120 (3.exe!main+0xb0) handler=0x40113b ⬐
    ↳ (3.exe!main+0xcb)
scopetable entry[1]. previous try level=0, filter=0x4010e8 (3.exe!main+0x78) handler=0x401100 ⬐
    ↳ (3.exe!main+0x90)
* SEH frame at 0x18ff78 prev=0x18ffc4 handler=0x4012e0 (3.exe!_except_handler3)
SEH3 frame. previous trylevel=0
scopetable entry[0]. previous try level=-1, filter=0x40160d (3.exe!mainCRTStartup+0x18d) ⬐
    ↳ handler=0x401621 (3.exe!mainCRTStartup+0x1a1)
* SEH frame at 0x18ffc4 prev=0x18ffe4 handler=0x771f71f5 (ntdll.dll!__except_handler4)
SEH4 frame. previous trylevel=0
SEH4 header :    GSCookieOffset=0xfffffffe GSCookieXOROffset=0x0
                EHCookieOffset=0xfffffffcc EHCookieXOROffset=0x0
scopetable entry[0]. previous try level=-2, filter=0x771f74d0 (ntdll.dll!⬐
    ↳ ___safe_se_handler_table+0x20) handler=0x771f90eb (ntdll.dll!_TppTerminateProcess@4+0x43)
* SEH frame at 0x18ffe4 prev=0xffffffff handler=0x77247428 (ntdll.dll!_FinalExceptionHandler@16⬐
    ↳ )
```

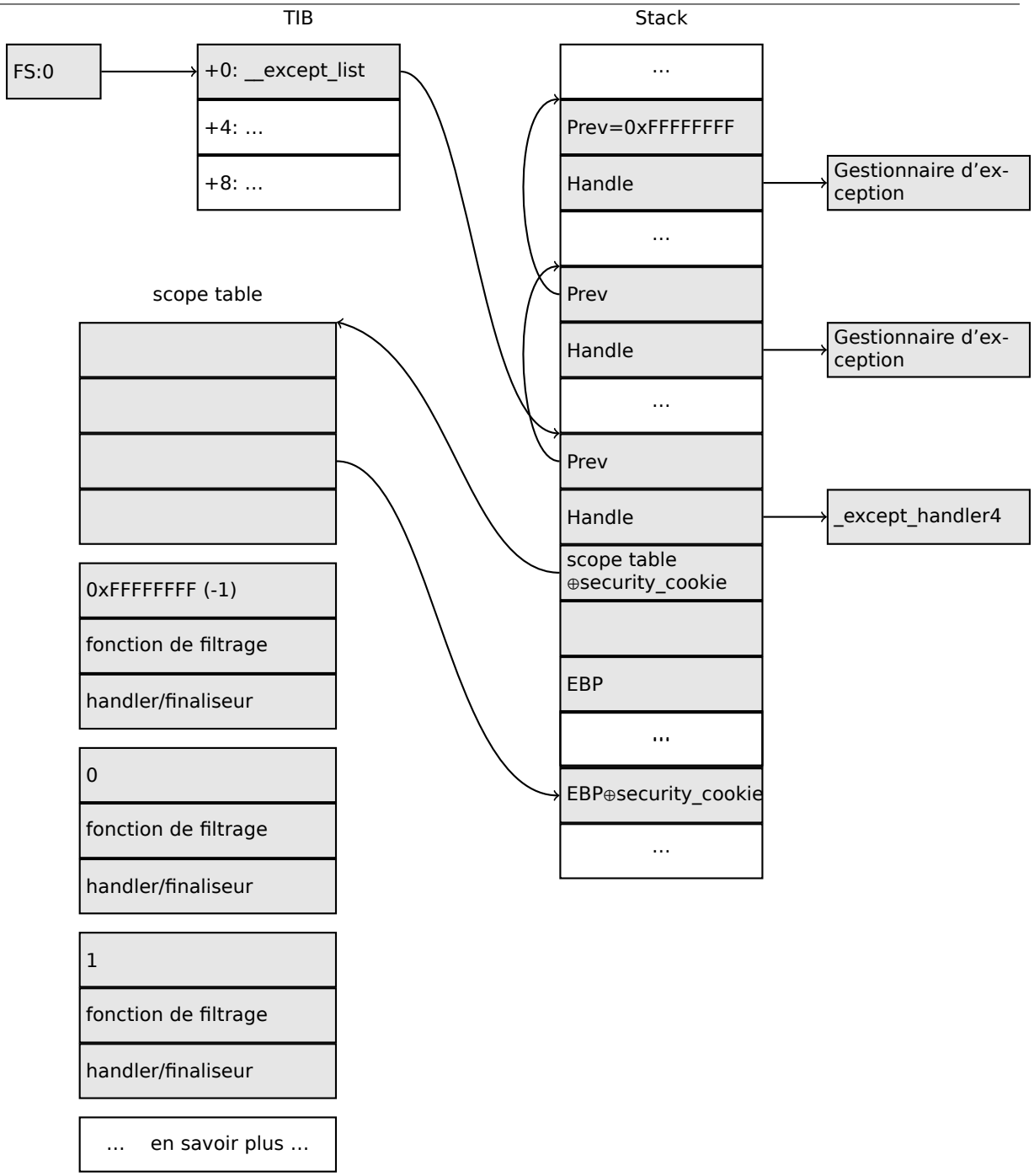
SEH4

Lors d'une attaque par dépassement de buffer (1.20.2 on page 276), l'adresse de la *scope table* peut être modifiée. C'est pourquoi à partir de MSVC 2005 SEH3 a été amélioré vers SEH4 pour ajouter une protection contre ce type d'attaque. Le pointeur vers la structure *scope table* est désormais xored avec la valeur d'un security cookie. Par ailleurs, la structure *scope table* a été étendue avec une en-tête contenant deux pointeurs vers des *security cookies*.

Chaque élément contient un offset dans la pile d'une valeur correspondant à: adresse du stack frame (EBP) xored avec la valeur du `security_cookie` lui aussi situé sur la pile.

Durant la gestion d'exception, l'intégrité de cette valeur est vérifiée. La valeur de chaque *security cookie* situé sur la pile est aléatoire. Une attaque à distance ne pourra donc pas la deviner.

Avec SEH4, la valeur initiale de *previous try level* est de -2 et non de -1.



Voici deux exemples compilés avec MSVC 2012 et SEH4:

Listing 4.31: MSVC 2012: exemple bloc try unique

```
$SG85485 DB    'hello #1!', 0aH, 00H
$SG85486 DB    'hello #2!', 0aH, 00H
$SG85488 DB    'access violation, can''t recover', 0aH, 00H

; scope table :
xdata$x          SEGMENT
__sehtable$_main DD 0ffffffeH   ; GS Cookie Offset
    DD           00H            ; GS Cookie XOR Offset
    DD           0ffffffccH     ; EH Cookie Offset
    DD           00H            ; EH Cookie XOR Offset
    DD           0fffffffeH     ; previous try level
    DD           FLAT :$LN12@main  ; filter
    DD           FLAT :$LN8@main   ; handler
xdata$x          ENDS

$T2 = -36        ; size = 4
_p$ = -32        ; size = 4
tv68 = -28       ; size = 4
__$SEHRec$ = -24 ; size = 24
```

```
_main    PROC
    push   ebp
    mov    ebp, esp
    push   -2
    push   OFFSET __sehtable$_main
    push   OFFSET __except_handler4
    mov    eax, DWORD PTR fs :0
    push   eax
    add    esp, -20
    push   ebx
    push   esi
    push   edi
    mov    eax, DWORD PTR ___security_cookie
    xor    DWORD PTR __$SEHRec$[ebp+16], eax ; xored pointer to scope table
    xor    eax, ebp
    push   eax                               ; ebp ^ security_cookie
    lea    eax, DWORD PTR __$SEHRec$[ebp+8] ; pointer to VC_EXCEPTION_REGISTRATION_RECORD
    mov    DWORD PTR fs :0, eax
    mov    DWORD PTR __$SEHRec$[ebp], esp
    mov    DWORD PTR _p$[ebp], 0
    mov    DWORD PTR __$SEHRec$[ebp+20], 0 ; previous try level
    push   OFFSET $SG85485 ; 'hello #1!'
    call   _printf
    add    esp, 4
    mov    eax, DWORD PTR _p$[ebp]
    mov    DWORD PTR [eax], 13
    push   OFFSET $SG85486 ; 'hello #2!'
    call   _printf
    add    esp, 4
    mov    DWORD PTR __$SEHRec$[ebp+20], -2 ; previous try level
    jmp    SHORT $LN6@main

; filter :
$LN7@main :
$LN12@main :
    mov    ecx, DWORD PTR __$SEHRec$[ebp+4]
    mov    edx, DWORD PTR [ecx]
    mov    eax, DWORD PTR [edx]
    mov    DWORD PTR $T2[ebp], eax
    cmp    DWORD PTR $T2[ebp], -1073741819 ; c0000005H
    jne    SHORT $LN4@main
    mov    DWORD PTR tv68[ebp], 1
    jmp    SHORT $LN5@main
$LN4@main :
    mov    DWORD PTR tv68[ebp], 0
$LN5@main :
    mov    eax, DWORD PTR tv68[ebp]
$LN9@main :
$LN11@main :
    ret    0

; handler :
$LN8@main :
    mov    esp, DWORD PTR __$SEHRec$[ebp]
    push   OFFSET $SG85488 ; 'access violation, can''t recover'
    call   _printf
    add    esp, 4
    mov    DWORD PTR __$SEHRec$[ebp+20], -2 ; previous try level
$LN6@main :
    xor    eax, eax
    mov    ecx, DWORD PTR __$SEHRec$[ebp+8]
    mov    DWORD PTR fs :0, ecx
    pop    ecx
    pop    edi
    pop    esi
    pop    ebx
    mov    esp, ebp
    pop    ebp
    ret    0
_main    ENDP
```

Listing 4.32: MSVC 2012: exemple de deux blocs try

```
$SG85486 DB    'in filter. code=0x%08X', 0aH, 00H
$SG85488 DB    'yes, that is our exception', 0aH, 00H
$SG85490 DB    'not our exception', 0aH, 00H
$SG85497 DB    'hello!', 0aH, 00H
$SG85499 DB    '0x112233 raised. now let''s crash', 0aH, 00H
$SG85501 DB    'access violation, can''t recover', 0aH, 00H
$SG85503 DB    'user exception caught', 0aH, 00H

xdata$x    SEGMENT
__sehtable$_main DD 0fffffffeH         ; GS Cookie Offset
                 DD    00H             ; GS Cookie XOR Offset
                 DD    0ffffffc8H      ; EH Cookie Offset
                 DD    00H             ; EH Cookie Offset
                 DD    0fffffffeH      ; previous try level for outer block
                 DD    FLAT :$LN19@main ; outer block filter
                 DD    FLAT :$LN9@main  ; outer block handler
                 DD    00H             ; previous try level for inner block
                 DD    FLAT :$LN18@main ; inner block filter
                 DD    FLAT :$LN13@main ; inner block handler
xdata$x    ENDS

$T2 = -40        ; size = 4
$T3 = -36        ; size = 4
_p$ = -32        ; size = 4
tv72 = -28       ; size = 4
__$SEHRec$ = -24 ; size = 24
_main    PROC
    push   ebp
    mov    ebp, esp
    push   -2  ; initial previous try level
    push   OFFSET __sehtable$_main
    push   OFFSET __except_handler4
    mov    eax, DWORD PTR fs :0
    push   eax ; prev
    add    esp, -24
    push   ebx
    push   esi
    push   edi
    mov    eax, DWORD PTR ___security_cookie
    xor    DWORD PTR __$SEHRec$[ebp+16], eax       ; xored pointer to scope table
    xor    eax, ebp                                ; ebp ^ security_cookie
    push   eax
    lea    eax, DWORD PTR __$SEHRec$[ebp+8]        ; pointer to ⬋
    ↳ VC_EXCEPTION_REGISTRATION_RECORD
    mov    DWORD PTR fs :0, eax
    mov    DWORD PTR __$SEHRec$[ebp], esp
    mov    DWORD PTR _p$[ebp], 0
    mov    DWORD PTR __$SEHRec$[ebp+20], 0 ; entering outer try block, setting previous try ⬋
    ↳ level=0
    mov    DWORD PTR __$SEHRec$[ebp+20], 1 ; entering inner try block, setting previous try ⬋
    ↳ level=1
    push   OFFSET $SG85497 ; 'hello!'
    call   _printf
    add    esp, 4
    push   0
    push   0
    push   0
    push   1122867 ; 00112233H
    call   DWORD PTR __imp__RaiseException@16
    push   OFFSET $SG85499 ; '0x112233 raised. now let''s crash'
    call   _printf
    add    esp, 4
    mov    eax, DWORD PTR _p$[ebp]
    mov    DWORD PTR [eax], 13
    mov    DWORD PTR __$SEHRec$[ebp+20], 0 ; exiting inner try block, set previous try level ⬋
    ↳ back to 0
    jmp    SHORT $LN2@main

; inner block filter :
```

```
$LN12@main :
$LN18@main :
    mov    ecx, DWORD PTR __$SEHRec$[ebp+4]
    mov    edx, DWORD PTR [ecx]
    mov    eax, DWORD PTR [edx]
    mov    DWORD PTR $T3[ebp], eax
    cmp    DWORD PTR $T3[ebp], -1073741819 ; c0000005H
    jne    SHORT $LN5@main
    mov    DWORD PTR tv72[ebp], 1
    jmp    SHORT $LN6@main
$LN5@main :
    mov    DWORD PTR tv72[ebp], 0
$LN6@main :
    mov    eax, DWORD PTR tv72[ebp]
$LN14@main :
$LN16@main :
    ret    0

; inner block handler :
$LN13@main :
    mov    esp, DWORD PTR __$SEHRec$[ebp]
    push   OFFSET $SG85501 ; 'access violation, can''t recover'
    call   _printf
    add    esp, 4
    mov    DWORD PTR __$SEHRec$[ebp+20], 0 ; exiting inner try block, setting previous try ⬋
    ↳ level back to 0
$LN2@main :
    mov    DWORD PTR __$SEHRec$[ebp+20], -2 ; exiting both blocks, setting previous try level ⬋
    ↳ back to -2
    jmp    SHORT $LN7@main

; outer block filter :
$LN8@main :
$LN19@main :
    mov    ecx, DWORD PTR __$SEHRec$[ebp+4]
    mov    edx, DWORD PTR [ecx]
    mov    eax, DWORD PTR [edx]
    mov    DWORD PTR $T2[ebp], eax
    mov    ecx, DWORD PTR __$SEHRec$[ebp+4]
    push   ecx
    mov    edx, DWORD PTR $T2[ebp]
    push   edx
    call   _filter_user_exceptions
    add    esp, 8
$LN10@main :
$LN17@main :
    ret    0

; outer block handler :
$LN9@main :
    mov    esp, DWORD PTR __$SEHRec$[ebp]
    push   OFFSET $SG85503 ; 'user exception caught'
    call   _printf
    add    esp, 4
    mov    DWORD PTR __$SEHRec$[ebp+20], -2 ; exiting both blocks, setting previous try level ⬋
    ↳ back to -2
$LN7@main :
    xor    eax, eax
    mov    ecx, DWORD PTR __$SEHRec$[ebp+8]
    mov    DWORD PTR fs :0, ecx
    pop    ecx
    pop    edi
    pop    esi
    pop    ebx
    mov    esp, ebp
    pop    ebp
    ret    0
_main    ENDP

_code$ = 8  ; size = 4
```

```
_ep$ = 12   ; size = 4
_filter_user_exceptions PROC
    push   ebp
    mov    ebp, esp
    mov    eax, DWORD PTR _code$[ebp]
    push   eax
    push   OFFSET $SG85486 ; 'in filter. code=0x%08X'
    call   _printf
    add    esp, 8
    cmp    DWORD PTR _code$[ebp], 1122867 ; 00112233H
    jne    SHORT $LN2@filter_use
    push   OFFSET $SG85488 ; 'yes, that is our exception'
    call   _printf
    add    esp, 4
    mov    eax, 1
    jmp    SHORT $LN3@filter_use
    jmp    SHORT $LN3@filter_use
$LN2@filter_use :
    push   OFFSET $SG85490 ; 'not our exception'
    call   _printf
    add    esp, 4
    xor    eax, eax
$LN3@filter_use :
    pop    ebp
    ret    0
_filter_user_exceptions ENDP
```

La signification de *cookies* est la suivante: `Cookie Offset` est la différence entre l'adresse dans la pile de la dernière valeur sauvegardée du registre EBP et de l'adresse dans la pile du résultat de l'addition $EBP \oplus security\_cookie$. `Cookie XOR Offset` est quant à lui la différence entre $EBP \oplus security\_cookie$ et la valeur conservée sur la pile.

Si le prédicat ci-dessous n'est pas respecté, le processus est arrêté du fait d'une corruption de la pile:

$$security\_cookie \oplus (CookieXOROffset + address\_of\_saved\_EBP) == stack[address\_of\_saved\_EBP + CookieOffset]$$

Lorsque `Cookie Offset` vaut -2, ceci indique qu'il n'est pas présent.

La vérification des *Cookies* est aussi implémentée dans mon tracer, voir GitHub pour les détails.

Pour les versions à partir de MSVC 2005, il est toujours possible de revenir à la version SEH3 en utilisant l'option `/GS-`. Toutefois, le CRT continue à utiliser SEH4.

Windows x64

Vous imaginez bien qu'il n'est pas très performant de construire le contexte SEH dans le prologue de chaque fonction. S'y ajoute les nombreux changements de la valeur de *previous try level* durant l'exécution de la fonction.

C'est pourquoi avec x64, la manière de faire a complèetement changé. Tous les pointeurs vers les blocs `try`, les filtres et les gestionnaires sont désormais stockés dans une nouveau segment PE: `.pdata` à partir duquel les gestionnaires d'exception de l'OS récupèreront les informations.

Voici deux exemples tirés de la section précédente et compilés pour x64:

Listing 4.33: MSVC 2012

```
$SG86276 DB     'hello #1!', 0aH, 00H
$SG86277 DB     'hello #2!', 0aH, 00H
$SG86279 DB     'access violation, can''t recover', 0aH, 00H

pdata   SEGMENT
$pdata$main DD  imagerel $LN9
        DD      imagerel $LN9+61
        DD      imagerel $unwind$main
pdata   ENDS
pdata   SEGMENT
$pdata$main$filt$0 DD imagerel main$filt$0
        DD      imagerel main$filt$0+32
        DD      imagerel $unwind$main$filt$0
```

```
pdata   ENDS
xdata   SEGMENT
$unwind$main DD 020609H
        DD      030023206H
        DD      imagerel __C_specific_handler
        DD      01H
        DD      imagerel $LN9+8
        DD      imagerel $LN9+40
        DD      imagerel main$filt$0
        DD      imagerel $LN9+40
$unwind$main$filt$0 DD 020601H
        DD      050023206H
xdata   ENDS

_TEXT   SEGMENT
main    PROC
$LN9 :
        push    rbx
        sub     rsp, 32
        xor     ebx, ebx
        lea     rcx, OFFSET FLAT :$SG86276 ; 'hello #1!'
        call    printf
        mov     DWORD PTR [rbx], 13
        lea     rcx, OFFSET FLAT :$SG86277 ; 'hello #2!'
        call    printf
        jmp     SHORT $LN8@main
$LN6@main :
        lea     rcx, OFFSET FLAT :$SG86279 ; 'access violation, can''t recover'
        call    printf
        npad    1 ; align next label
$LN8@main :
        xor     eax, eax
        add     rsp, 32
        pop     rbx
        ret     0
main    ENDP
_TEXT   ENDS

text$x  SEGMENT
main$filt$0 PROC
        push    rbp
        sub     rsp, 32
        mov     rbp, rdx
$LN5@main$filt$ :
        mov     rax, QWORD PTR [rcx]
        xor     ecx, ecx
        cmp     DWORD PTR [rax], -1073741819; c0000005H
        sete    cl
        mov     eax, ecx
$LN7@main$filt$ :
        add     rsp, 32
        pop     rbp
        ret     0
        int     3
main$filt$0 ENDP
text$x  ENDS
```

Listing 4.34: MSVC 2012

```
$SG86277 DB     'in filter. code=0x%08X', 0aH, 00H
$SG86279 DB     'yes, that is our exception', 0aH, 00H
$SG86281 DB     'not our exception', 0aH, 00H
$SG86288 DB     'hello!', 0aH, 00H
$SG86290 DB     '0x112233 raised. now let''s crash', 0aH, 00H
$SG86292 DB     'access violation, can''t recover', 0aH, 00H
$SG86294 DB     'user exception caught', 0aH, 00H

pdata   SEGMENT
$pdata$filter_user_exceptions DD imagerel $LN6
        DD      imagerel $LN6+73
```

```
        DD      imagerel $unwind$filter_user_exceptions
$pdata$main DD  imagerel $LN14
        DD      imagerel $LN14+95
        DD      imagerel $unwind$main
pdata   ENDS
pdata   SEGMENT
$pdata$main$filt$0 DD imagerel main$filt$0
        DD      imagerel main$filt$0+32
        DD      imagerel $unwind$main$filt$0
$pdata$main$filt$1 DD imagerel main$filt$1
        DD      imagerel main$filt$1+30
        DD      imagerel $unwind$main$filt$1
pdata   ENDS

xdata   SEGMENT
$unwind$filter_user_exceptions DD 020601H
        DD      030023206H
$unwind$main DD 020609H
        DD      030023206H
        DD      imagerel __C_specific_handler
        DD      02H
        DD      imagerel $LN14+8
        DD      imagerel $LN14+59
        DD      imagerel main$filt$0
        DD      imagerel $LN14+59
        DD      imagerel $LN14+8
        DD      imagerel $LN14+74
        DD      imagerel main$filt$1
        DD      imagerel $LN14+74
$unwind$main$filt$0 DD 020601H
        DD      050023206H
$unwind$main$filt$1 DD 020601H
        DD      050023206H
xdata   ENDS

_TEXT   SEGMENT
main    PROC
$LN14 :
        push    rbx
        sub     rsp, 32
        xor     ebx, ebx
        lea     rcx, OFFSET FLAT :$SG86288 ; 'hello!'
        call    printf
        xor     r9d, r9d
        xor     r8d, r8d
        xor     edx, edx
        mov     ecx, 1122867 ; 00112233H
        call    QWORD PTR __imp_RaiseException
        lea     rcx, OFFSET FLAT :$SG86290 ; '0x112233 raised. now let''s crash'
        call    printf
        mov     DWORD PTR [rbx], 13
        jmp     SHORT $LN13@main
$LN11@main :
        lea     rcx, OFFSET FLAT :$SG86292 ; 'access violation, can''t recover'
        call    printf
        npad    1 ; align next label
$LN13@main :
        jmp     SHORT $LN9@main
$LN7@main :
        lea     rcx, OFFSET FLAT :$SG86294 ; 'user exception caught'
        call    printf
        npad    1 ; align next label
$LN9@main :
        xor     eax, eax
        add     rsp, 32
        pop     rbx
        ret     0
main    ENDP

text$x  SEGMENT
```

```
main$filt$0 PROC
        push    rbp
        sub     rsp, 32
        mov     rbp, rdx
$LN10@main$filt$ :
        mov     rax, QWORD PTR [rcx]
        xor     ecx, ecx
        cmp     DWORD PTR [rax], -1073741819; c0000005H
        sete    cl
        mov     eax, ecx
$LN12@main$filt$ :
        add     rsp, 32
        pop     rbp
        ret     0
        int     3
main$filt$0 ENDP

main$filt$1 PROC
        push    rbp
        sub     rsp, 32
        mov     rbp, rdx
$LN6@main$filt$ :
        mov     rax, QWORD PTR [rcx]
        mov     rdx, rcx
        mov     ecx, DWORD PTR [rax]
        call    filter_user_exceptions
        npad    1 ; align next label
$LN8@main$filt$ :
        add     rsp, 32
        pop     rbp
        ret     0
        int     3
main$filt$1 ENDP
text$x  ENDS

_TEXT   SEGMENT
code$ = 48
ep$ = 56
filter_user_exceptions PROC
$LN6 :
        push    rbx
        sub     rsp, 32
        mov     ebx, ecx
        mov     edx, ecx
        lea     rcx, OFFSET FLAT :$SG86277 ; 'in filter. code=0x%08X'
        call    printf
        cmp     ebx, 1122867; 00112233H
        jne     SHORT $LN2@filter_use
        lea     rcx, OFFSET FLAT :$SG86279 ; 'yes, that is our exception'
        call    printf
        mov     eax, 1
        add     rsp, 32
        pop     rbx
        ret     0
$LN2@filter_use :
        lea     rcx, OFFSET FLAT :$SG86281 ; 'not our exception'
        call    printf
        xor     eax, eax
        add     rsp, 32
        pop     rbx
        ret     0
filter_user_exceptions ENDP
_TEXT   ENDS
```

Pour plus d'informations sur le sujet, lisez [Igor Skochinsky, *Compiler Internals: Exceptions and RTTI*, (2012)] [54].

Hormis les informations d'exception, `.pdata` est aussi une section qui contient les adresses de début et de

[54] Aussi disponible en http://go.yurichev.com/17294

fin de toutes les fonctions. Elle revêt donc un intérêt particulier dans le cadre d'une analyse automatique d'un programme.

En lire plus sur SEH

[Matt Pietrek, *A Crash Course on the Depths of Win32™ Structured Exception Handling*, (1997)][55], [Igor Skochinsky, *Compiler Internals: Exceptions and RTTI*, (2012)] [56].

4.5.4 Windows NT: Section critique

Dans tout OS, les sections critiques sont très importantes dans un système multithreadé, principalement pour donner la garantie qu'un seul thread peut accéder à certaines données à un instant précis, en bloquant les autres threads et les interruptions.

Voilà comment une structure CRITICAL_SECTION est déclarée dans la série des OS Windows NT :

Listing 4.35: (Windows Research Kernel v1.2) public/sdk/inc/nturtl.h

```
typedef struct _RTL_CRITICAL_SECTION {
    PRTL_CRITICAL_SECTION_DEBUG DebugInfo;

    //
    //  The following three fields control entering and exiting the critical
    //  section for the resource
    //

    LONG LockCount;
    LONG RecursionCount;
    HANDLE OwningThread;        // from the thread's ClientId->UniqueThread
    HANDLE LockSemaphore;
    ULONG_PTR SpinCount;        // force size on 64-bit systems when packed
} RTL_CRITICAL_SECTION, *PRTL_CRITICAL_SECTION;
```

Voilà comment la fonction EnterCriticalSection() fonctionne:

Listing 4.36: Windows 2008/ntdll.dll/x86 (begin)

```
_RtlEnterCriticalSection@4

var_C           = dword ptr -0Ch
var_8           = dword ptr -8
var_4           = dword ptr -4
arg_0           = dword ptr  8

                mov     edi, edi
                push    ebp
                mov     ebp, esp
                sub     esp, 0Ch
                push    esi
                push    edi
                mov     edi, [ebp+arg_0]
                lea     esi, [edi+4] ; LockCount
                mov     eax, esi
                lock btr dword ptr [eax], 0
                jnb     wait ; jump if CF=0

loc_7DE922DD :
                mov     eax, large fs :18h
                mov     ecx, [eax+24h]
                mov     [edi+0Ch], ecx
                mov     dword ptr [edi+8], 1
                pop     edi
                xor     eax, eax
                pop     esi
                mov     esp, ebp
```

[55] Aussi disponible en http://go.yurichev.com/17293
[56] Aussi disponible en http://go.yurichev.com/17294

```
                pop     ebp
                retn    4

... skipped
```

L'instruction la plus importante dans ce morceau de code est BTR (préfixée avec LOCK):

Le bit d'index zéro est stocké dans le flag CF et est éffacé en mémoire. Ceci est une opération atomique, bloquant tous les autres accès du CPU à cette zone de mémoire (regardez le préfixe LOCK se trouvant avant l'instruction BTR). Si le bit en LockCount est 1, bien, remise à zéro et retour de la fonction: nous sommes dans une section critique.

Si non—la section critique est déjà occupée par un autre thread, donc attendre. L'attente est effectuée en utilisant WaitForSingleObject().

Et voici comment la fonction LeaveCriticalSection() fonctionne:

Listing 4.37: Windows 2008/ntdll.dll/x86 (begin)

```
_RtlLeaveCriticalSection@4 proc near

arg_0           = dword ptr  8

                mov     edi, edi
                push    ebp
                mov     ebp, esp
                push    esi
                mov     esi, [ebp+arg_0]
                add     dword ptr [esi+8], 0FFFFFFFFh ; RecursionCount
                jnz     short loc_7DE922B2
                push    ebx
                push    edi
                lea     edi, [esi+4]    ; LockCount
                mov     dword ptr [esi+0Ch], 0
                mov     ebx, 1
                mov     eax, edi
                lock xadd [eax], ebx
                inc     ebx
                cmp     ebx, 0FFFFFFFFh
                jnz     loc_7DEA8EB7

loc_7DE922B0 :
                pop     edi
                pop     ebx

loc_7DE922B2 :
                xor     eax, eax
                pop     esi
                pop     ebp
                retn    4

... skipped
```

XADD signifie « exchange and add » (échanger et ajouter).

Dans ce cas, elle ajoute 1 à LockCount, en même temps qu'elle sauve la valeur initiale de LockCount dans le registre EBX. Toutefois, la valeur dans EBX est incrémentée avec l'aide du INC EBX subséquent, et il sera ainsi égal à la valeur modifiée de LockCount.

Cette opération est atomique puisqu'elle est préfixée par LOCK, signifiant que tous les autres CPUs ou cœurs de CPU dans le système ne peuvent pas accéder à cette zone de la mémoire.

Le préfixe LOCK est très important:

sans lui deux threads, travaillant chacune sur un CPU ou un cœur de CPU séparé pourraient essayer d'entrer dans la section critique et de modifier la valeur en mémoire, ce qui résulterait en un comportement non déterministe.

Chapitre 5

Outils

> Maintenant que Dennis Yurichev a réalisé ce livre gratuit, il s'agit d'une contribution au monde de la connaissance et de l'éducation gratuite. Cependant, pour l'amour de la liberté, nous avons besoin d'outils de rétro-ingénierie (libres) afin de remplacer les outils propriétaires mentionnés dans ce livre.
>
> Richard M. Stallman

5.1 Analyse statique

Outils à utiliser lorsqu'aucun processus n'est en cours d'exécution.

- (Gratuit, open-source) *ent*[1] : outil d'analyse d'entropie. En savoir plus sur l'entropie : **??** on page ??.
- *Hiew*[2] : pour de petites modifications de code dans les fichiers binaires. Inclut un assembleur/désassembleur.
- Libre, open-source *GHex*[3] : éditeur hexadécimal simple pour Linux.
- (Libre, open-source) *xxd* et *od* : utilitaires standards UNIX pour réaliser un dump.
- (Libre, open-source) *strings* : outil *NIX pour rechercher des chaînes ASCII dans des fichiers binaires, fichiers exécutables inclus. Sysinternals ont une alternative [4] qui supporte les larges chaînes de caractères (UTF-16, très utilisé dans Windows).
- (Libre, open-source) *Binwalk*[5] : analyser les images firmware.
- (Libre, open-source) *binary grep* : un petit utilitaire pour rechercher une séquence d'octets dans un paquet de fichiers, incluant ceux non exécutables : GitHub. Il y a aussi rafind2 dans rada.re pour le même usage.

5.1.1 Désassembleurs

- *IDA*. Une ancienne version Freeware est disponible via téléchargement [6]. Anti-sèche des touches de raccourci: **??** on page ??
- *Binary Ninja*[7]
- (Gratuit, open-source) *zynamics BinNavi*[8]

[1] http://www.fourmilab.ch/random/
[2] hiew.ru
[3] https://wiki.gnome.org/Apps/Ghex
[4] https://technet.microsoft.com/en-us/sysinternals/strings
[5] http://binwalk.org/
[6] hex-rays.com/products/ida/support/download_Freeware.shtml
[7] http://binary.ninja/
[8] https://www.zynamics.com/binnavi.html

- (Gratuit, open-source) *objdump* : simple utilitaire en ligne de commandes pour désassembler et réaliser des dumps.
- (Gratuit, open-source) *readelf*[9] : réaliser des dumps d'informations sur des fichiers ELF.

5.1.2 Décompilateurs

Il n'existe qu'un seul décompilateur connu en C, d'excellente qualité et disponible au public *Hex-Rays* : hex-rays.com/products/decompiler/

Pour en savoir plus: **??** on page ??.

5.1.3 Comparaison de versions

Vous pouvez éventuellement les utiliser lorsque vous comparez la version originale d'un exécutable et une version remaniée, pour déterminer ce qui a été corrigé et en déterminer la raison.

- (Gratuit) *zynamics BinDiff*[10]
- (Gratuit, open-source) *Diaphora*[11]

5.2 Analyse dynamique

Outils à utiliser lorsque que le système est en cours d'exploitation ou lorsqu'un processus est en cours d'exécution.

5.2.1 Débogueurs

- (Gratuit) *OllyDbg*. Débogueur Win32 très populaire [12]. Anti-sèche des touches de raccourci: **??** on page ??
- (Gratuit, open-source) *GDB*. Débogueur peu populaire parmi les ingénieurs en rétro-ingénierie, car il est principalement destiné aux programmeurs. Quelques commandes : **??** on page ??. Il y a une interface graphique pour GDB, "GDB dashboard"[13].
- (Gratuit, open-source) *LLDB*[14].
- *WinDbg*[15] : débogueur pour le noyau Windows.
- (Gratuit, open-source) *Radare* AKA rada.re AKA r2[16]. Une interface graphique existe aussi : *ragui*[17].
- (Gratuit, open-source) *tracer*. L'auteur utilise souvent *tracer* [18] au lieu d'un débogueur.

  L'auteur de ces lignes a finalement arrêté d'utiliser un débogueur, depuis que tout ce dont il a besoin est de repérer les arguments d'une fonction lorsque cette dernière est exécutée, ou l'état des registres à un instant donné. Le temps de chargement d'un débogueur étant trop long, un petit utilitaire sous le nom de *tracer* a été conçu. Il fonctionne depuis la ligne de commandes, permettant d'intercepter l'exécution d'une fonction, en plaçant des breakpoints à des endroits définis, en lisant et en changeant l'état des registres, etc...

  N.B.: *tracer* n'évolue pas, parce qu'il a été développé en tant qu'outil de démonstration pour ce livre, et non pas comme un outil dont on se servirait au quotidien.

[9] https://sourceware.org/binutils/docs/binutils/readelf.html
[10] https://www.zynamics.com/software.html
[11] https://github.com/joxeankoret/diaphora
[12] ollydbg.de
[13] https://github.com/cyrus-and/gdb-dashboard
[14] http://lldb.llvm.org/
[15] https://developer.microsoft.com/en-us/windows/hardware/windows-driver-kit
[16] http://rada.re/r/
[17] http://radare.org/ragui/
[18] yurichev.com

5.2.2 Tracer les appels de librairies

ltrace[19].

5.2.3 Tracer les appels système

strace / dtruss

Montre les appels système (syscalls(4.3 on page 557)) effectués dans l'immédiat.

Par exemple:

```
# strace df -h

...

access("/etc/ld.so.nohwcap", F_OK)      = -1 ENOENT (No such file or directory)
open("/lib/i386-linux-gnu/libc.so.6", O_RDONLY|O_CLOEXEC) = 3
read(3, "\177ELF\1\1\1\0\0\0\0\0\0\0\0\0\3\0\3\0\1\0\0\0\220\232\1\0004\0\0\0"..., 512) = 512
fstat64(3, {st_mode=S_IFREG|0755, st_size=1770984, ...}) = 0
mmap2(NULL, 1780508, PROT_READ|PROT_EXEC, MAP_PRIVATE|MAP_DENYWRITE, 3, 0) = 0xb75b3000
```

Mac OS X a dtruss pour faire la même chose.

Cygwin a également strace, mais de ce que je sais, cela ne fonctionne que pour les fichiers .exe compilés pour l'environnement Cygwin lui-même.

5.2.4 Sniffer le réseau

Sniffer signifie intercepter des informations qui peuvent vous intéresser.

(Gratuit, open-source) *Wireshark*[20] pour sniffer le réseau. Peut également sniffer les protocoles USB [21].

Wireshark a un petit (ou vieux) frère *tcpdump*[22], outil simple en ligne de commandes.

5.2.5 Sysinternals

(Gratuit) Sysinternals (développé par Mark Russinovich) [23]. Ces outils sont importants et valent la peine d'être étudiés : Process Explorer, Handle, VMMap, TCPView, Process Monitor.

5.2.6 Valgrind

(Gratuit, open-source) un puissant outil pour détecter les fuites mémoire : http://valgrind.org/. Grâce à ses puissants mécanismes JIT[24] ("Just In Time"), Valgrind est utilisé comme un framework pour d'autres outils.

5.2.7 Emulateurs

- (Gratuit, open-source) *QEMU*[25] : émulateur pour différents CPUs et architectures.
- (Gratuit, open-source) *DosBox*[26] : émulateur MS-DOS, principalement utilisé pour le rétro-gaming.
- (Gratuit, open-source) *SimH*[27] : émulateur d'anciens ordinateurs, unités centrales, etc...

[19] http://www.ltrace.org/
[20] https://www.wireshark.org/
[21] https://wiki.wireshark.org/CaptureSetup/USB
[22] http://www.tcpdump.org/
[23] https://technet.microsoft.com/en-us/sysinternals/bb842062
[24] Just-In-Time compilation
[25] http://qemu.org
[26] https://www.dosbox.com/
[27] http://simh.trailing-edge.com/

5.3 Autres outils

Microsoft Visual Studio Express [28] : Version gratuite simplifiée de Visual Studio, pratique pour des études de cas simples.

Quelques options utiles : **??** on page ??.

Il y a un site web appelé “Compiler Explorer”, permettant de compiler des petits morceaux de code et de voir le résultat avec des versions variées de GCC et d’architectures (au moins x86, ARM, MIPS): http://godbolt.org/—Je l’aurais utilisé pour le livre si je l’avais connu!

5.3.1 Calculatrices

Une bonne calculatrice pour les besoins des rétro-ingénieurs doit au moins supporter les bases décimale, hexadécimale et binaire, ainsi que plusieurs opérations importantes comme XOR et les décalages.

- IDA possède une calculatrice intégrée (“?”).
- rada.re a *rax2*.
- https://github.com/DennisYurichev/progcalc
- En dernier recours, la calculatrice standard de Windows dispose d’un mode programmeur.

5.4 Un outil manquant ?

Si vous connaissez un bon outil non listé précédemment, n’hésitez pas à m’en faire la remarque : dennis@yurichev.com.

[28] visualstudio.com/en-US/products/visual-studio-express-vs

Chapitre 6

Exemples de Reverse Engineering de format de fichier propriétaire

6.1 Chiffrement primitif avec XOR

6.1.1 Chiffrement XOR le plus simple

J'ai vu une fois un logiciel où tous les messages de débogage étaient chiffrés en utilisant XOR avec une valeur de 3. Autrement dit, les deux bits les plus bas de chaque caractères étaient inversés.

"Hello, world" devenait "Kfool/#tlqog":

```
#!/usr/bin/python

msg="Hello, world!"

print "".join(map(lambda x : chr(ord(x)^3), msg))
```

Ceci est un chiffrement assez intéressant (ou plutôt une offuscation), car il possède deux propriétés importantes: 1) fonction unique pour le chiffrement/déchiffrement, il suffit de l'appliquer à nouveau; 2) les caractères résultants sont aussi imprimable, donc la chaîne complète peut être utilisée dans du code source, sans caractères d'échappement.

La seconde propriété exploite le fait que tous les caractères imprimables sont organisés en lignes: 0x2x-0x7x, et lorsque vous inversez les deux bits de poids faible, le caractère est *déplacé* de 1 ou 3 caractères à droite ou à gauche, mais n'est jamais *déplacé* sur une autre ligne (peut-être non imprimable):

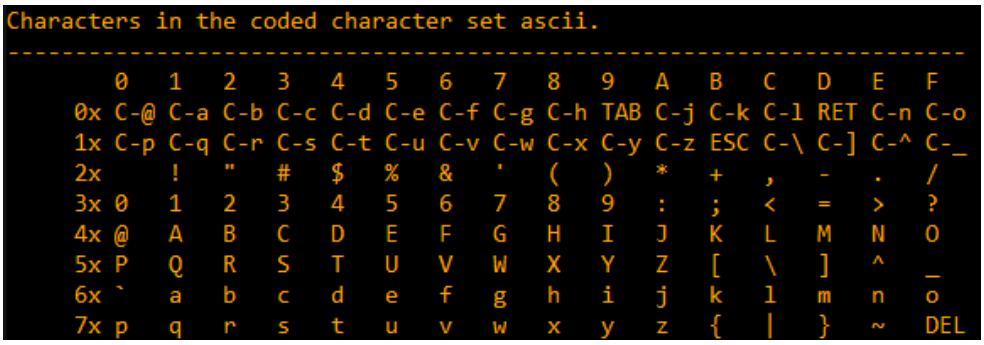


Fig. 6.1: Table ASCII 7-bit dans Emacs

...avec la seule exception du caractère 0x7F.

Par exemple, *chiffrons* les caractères de l'intervalle A-Z:

```
#!/usr/bin/python

msg="@ABCDEFGHIJKLMNO"

print "".join(map(lambda x : chr(ord(x)^3), msg))
```

Résultat:

```
CBA@GFEDKJIHONML
```

C’est comme si les caractères “@” et “C” avaient été échangés, ainsi que “B” et “a”.

Encore une fois, ceci est un exemple intéressant de l’exploitation des propriétés de XOR plutôt qu’un chiffrement: le même effet de *préservation de l’imprimabilité* peut être obtenu en échangeant chacun des 4 bits de poids faible, avec n’importe quelle combinaison.

6.1.2 Norton Guide: chiffrement XOR à 1 octet le plus simple possible

Norton Guide[1] était très populaire à l'époque de MS-DOS, c'était un programme résident qui fonctionnait comme un manuel de référence hypertexte.

Les bases de données de Norton Guide étaient des fichiers avec l'extension .ng, dont le contenu avait l'air chiffré:

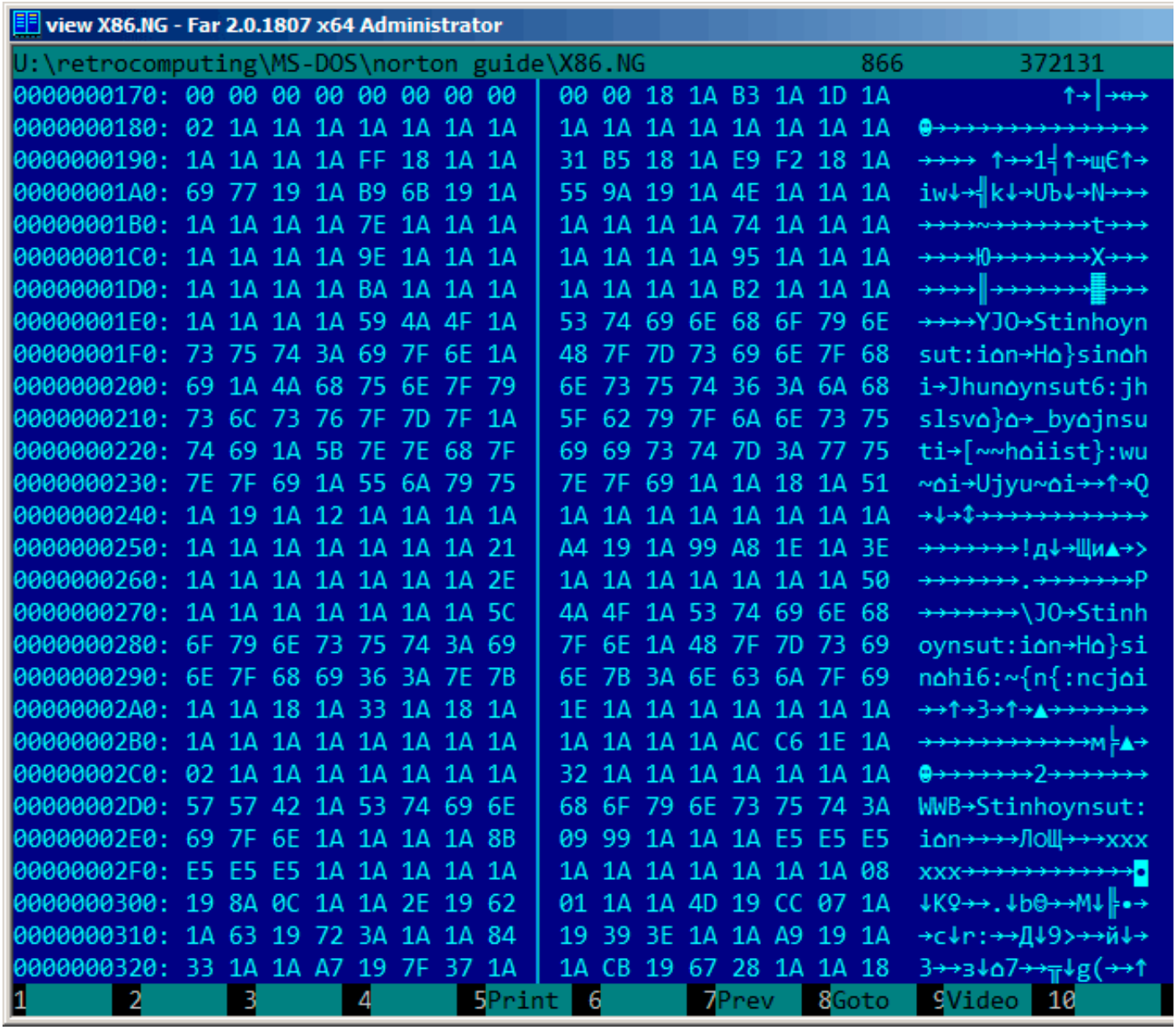


Fig. 6.2: Aspect très typique

Pourquoi pensons-nous qu'il est chiffré mais pas compressé?

Nous voyons que l'octet 0x1A (ressemblant à « → ») est très fréquent, ça ne serait pas possible dans un fichier compressé.

Nous voyons aussi de longues parties constituées seulement de lettres latines, et qui ressemble à des chaînes de caractères dans un langage inconnu.

[1] wikipédia

Puisque l'octet 0x1A revient si souvent, nous pouvons essayer de décrypter le fichier, en supposant qu'il est chiffré avec le chiffrement XOR le plus simple.

Si nous appliquons un XOR avec la constante 0x1A à chaque octet dans Hiew, nous voyons des chaînes de texte familières en anglais:

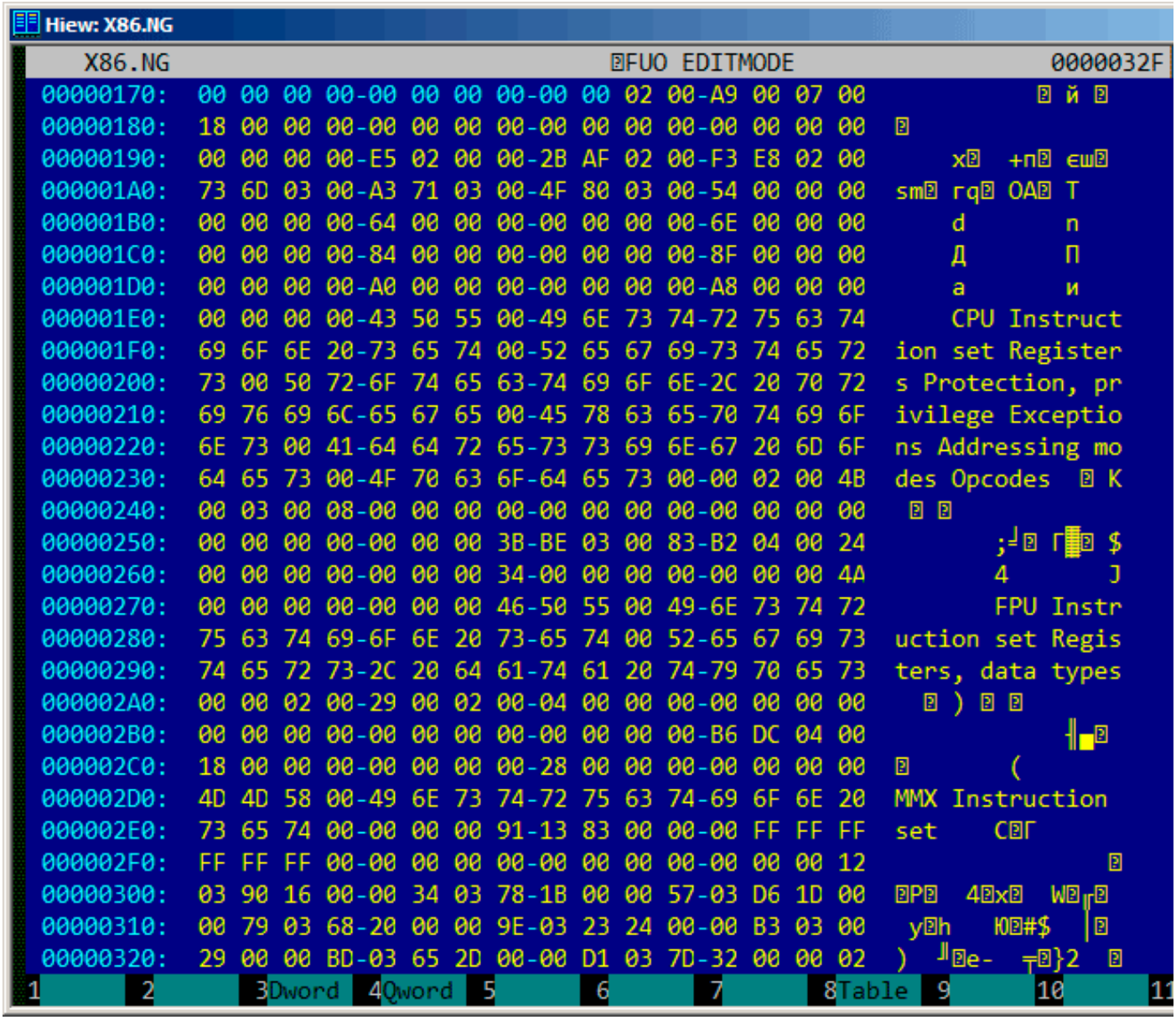


Fig. 6.3: XOR dans Hiew avec 0x1A

Le chiffrement XOR avec un seul octet constant est la méthode de chiffrement la plus simple, que l'on rencontre néanmoins parfois.

Maintenant, nous comprenons pourquoi l'octet 0x1A revenait si souvent: parce qu'il y a beaucoup d'octets à zéro et qu'ils sont remplacés par 0x1A dans la forme chiffrée.

Mais la constante pourrait être différente. Dans ce cas, nous pourrions essayer chaque constante dans l'intervalle 0..255 et chercher quelque chose de familier dans le fichier déchiffré. 256 n'est pas si grand.

Plus d'informations sur le format de fichier de Norton Guide: http://go.yurichev.com/17317.

Entropie

Une propriété très importante de tels systèmes de chiffrement est que l'entropie des blocs chiffrés/déchiffrés est la même.

Voici mon analyse faite dans Wolfram Mathematica 10.

Listing 6.1: Wolfram Mathematica 10

```
In[1]:= input = BinaryReadList["X86.NG"];

In[2]:= Entropy[2, input] // N
Out[2]= 5.62724

In[3]:= decrypted = Map[BitXor[#, 16^^1A] &, input];

In[4]:= Export["X86_decrypted.NG", decrypted, "Binary"];

In[5]:= Entropy[2, decrypted] // N
Out[5]= 5.62724

In[6]:= Entropy[2, ExampleData[{"Text", "ShakespearesSonnets"}]] // N
Out[6]= 4.42366
```

Ici, nous chargeons le fichier, obtenons son entropie, le déchiffrons, le sauvons et obtenons à nouveau son entropie (la même!).

Mathematica fourni également quelques textes en langue anglaise bien connus pour analyse.

Nous obtenons ainsi l'entropie de sonnets de Shakespeare, et elle est proche de l'entropie du fichier que nous venons d'analyser.

Le fichier que nous avons analysé consiste en des phrases en langue anglaise, qui sont proches du langage de Shakespeare.

Et le texte en langue anglaise XOR-é possède la même entropie.

Toutefois, ceci n'est pas vrai lorsque le fichier est XOR-é avec un pattern de plus d'un octet.

Le fichier qui vient d'être analysé peut être téléchargé ici: http://go.yurichev.com/17350.

Encore un mot sur la base de l'entropie

Wolfram Mathematica calcule l'entropie avec une base e (base des logarithmes naturels), et l'utilitaire[2] UNIX *ent* utilise une base 2.

Donc, nous avons mis explicitement une base 2 dans la commande `Entropy`, donc Mathematica nous donne le même résultat que l'utilitaire *ent*.

[2] http://www.fourmilab.ch/random/

6.1.3 Chiffrement le plus simple possible avec un XOR de 4-octets

Si un pattern plus long était utilisé, comme un pattern de 4 octets, ça serait facile à repérer.

Par exemple, voici le début du fichier kernel32.dll (version 32-bit de Windows Server 2008):

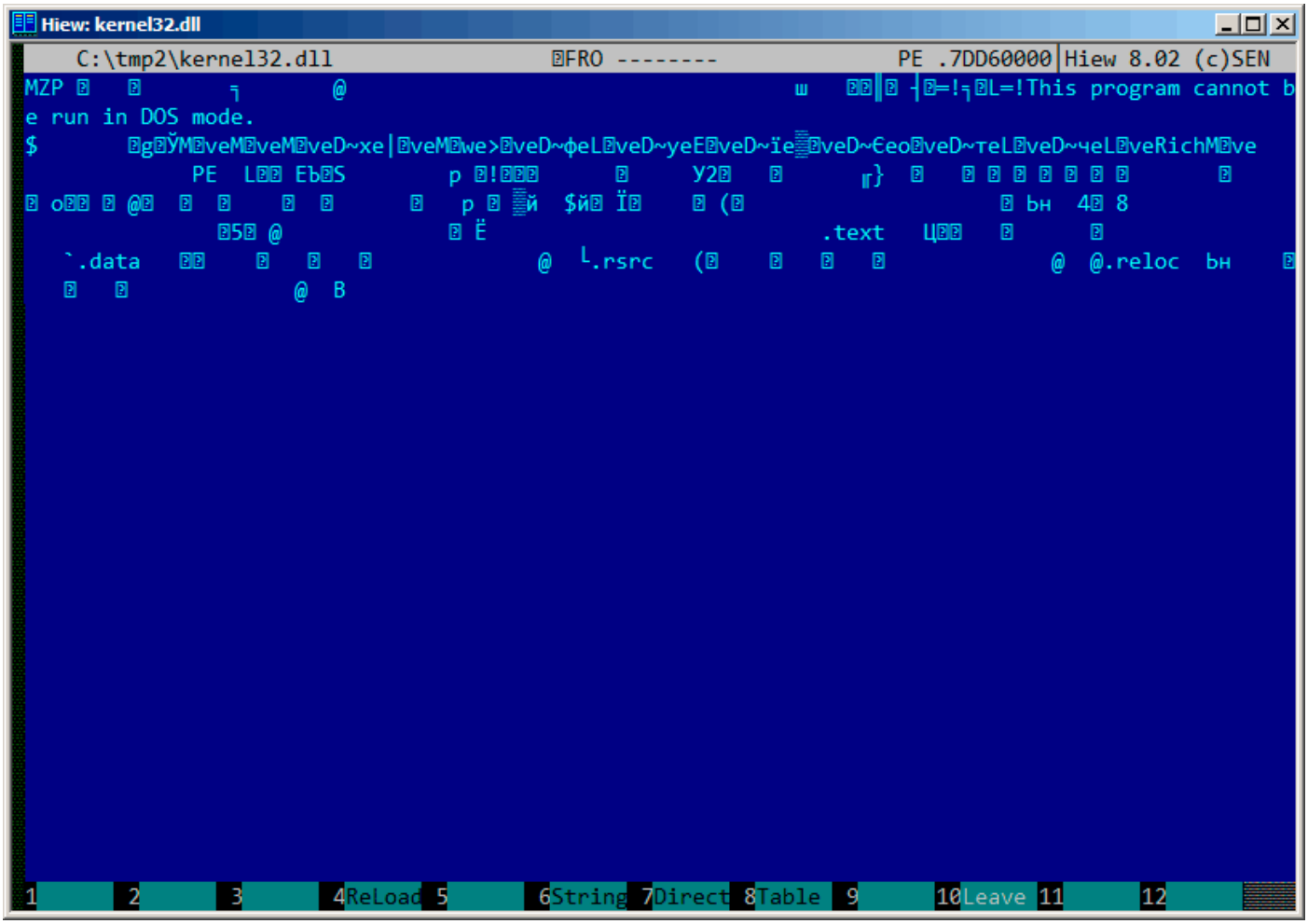


Fig. 6.4: Fichier original

Ici, il est « chiffré » avec une clef de 4-octet:

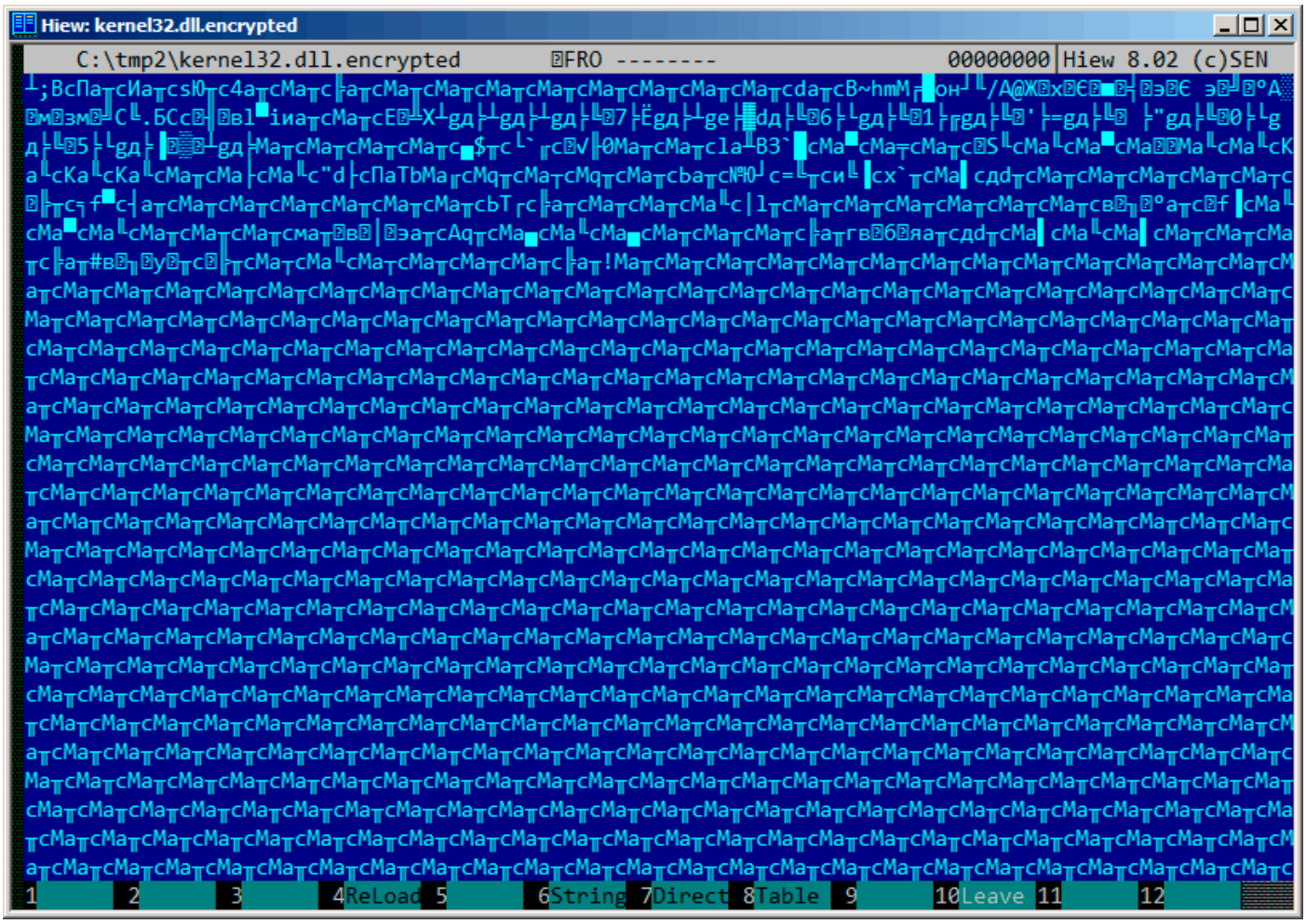


Fig. 6.5: Fichier « chiffré »

Il est facile de repérer les 4 symboles récurrents.

En effet, l'entête d'un fichier PE comporte de longues zones de zéro, ce qui explique que la clef devient visible.

Voici le début d'un entête PE au format hexadécimal:

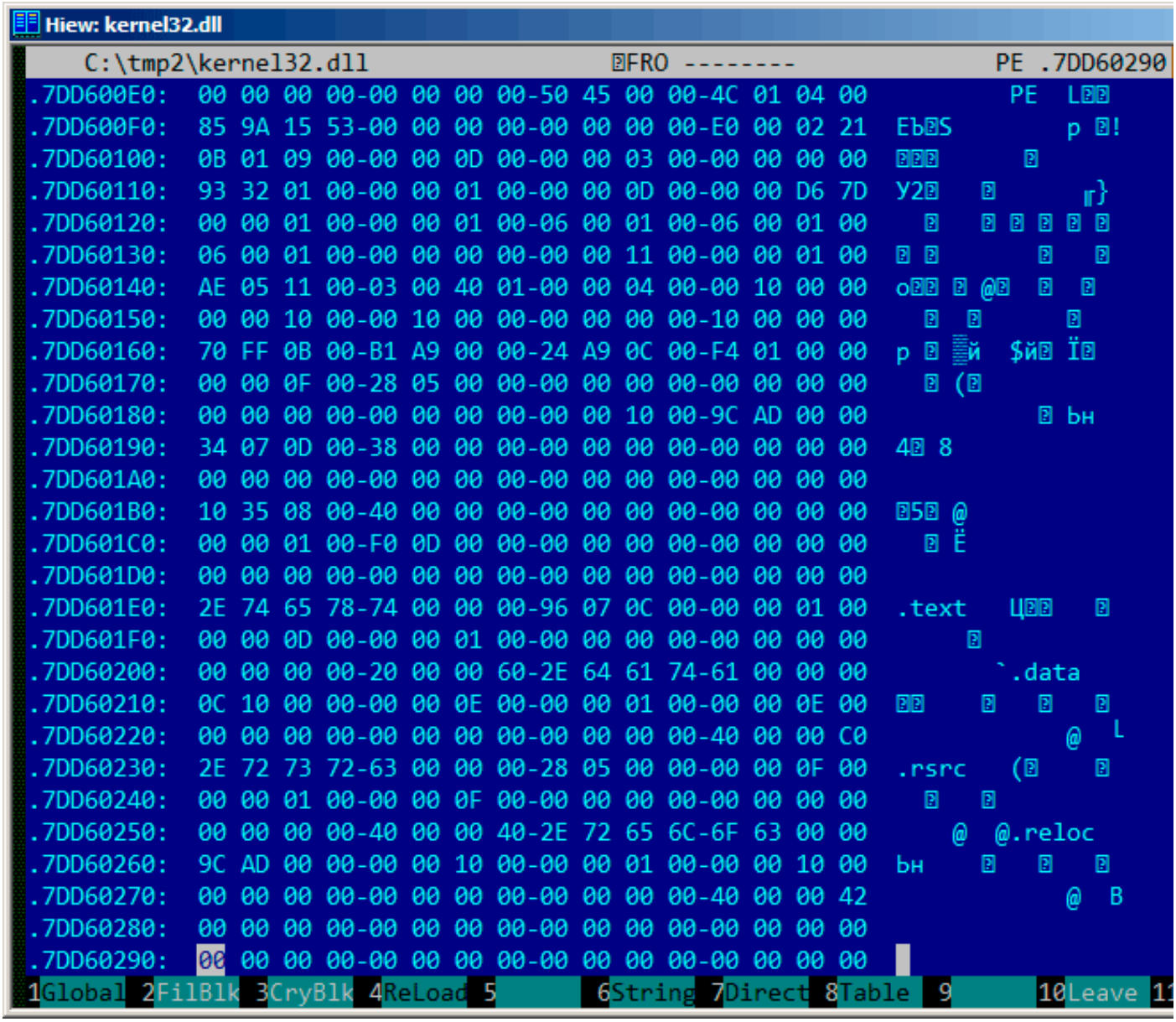


Fig. 6.6: Entête PE

Le voici « chiffré » :

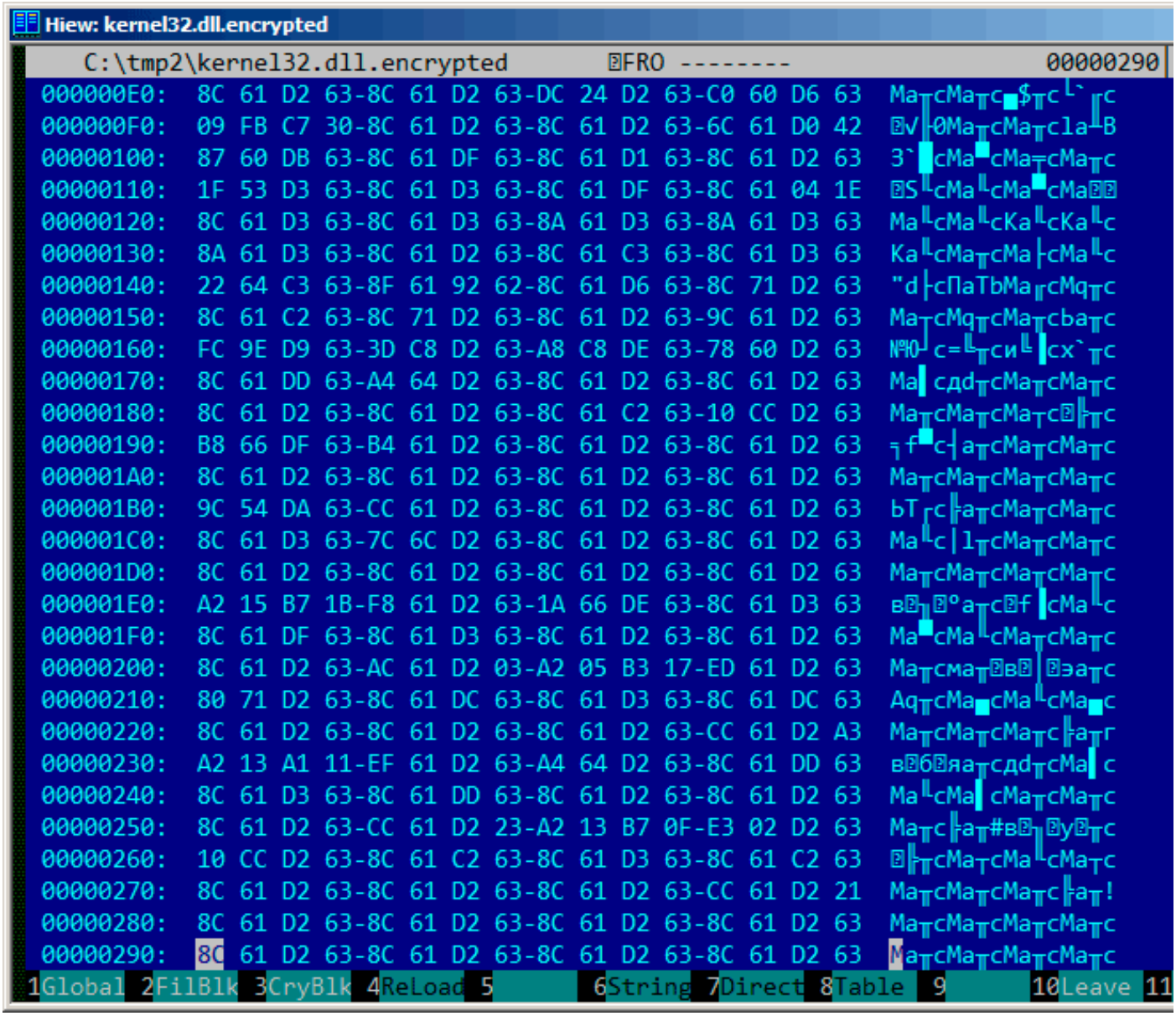


Fig. 6.7: Entête PE « chiffré »

Il est facile de repérer que la clef est la séquence de 4 octets suivant: `8C 61 D2 63`.

Avec cette information, c'est facile de déchiffrer le fichier entier.

Il est important de garder à l'esprit ces propriétés importantes des fichiers PE: 1) l'entête PE comporte de nombreuses zones remplies de zéro; 2) toutes les sections PE sont complétées avec des zéros jusqu'à une limite de page (4096 octets), donc il y a d'habitude de longues zones à zéro après chaque section.

Quelques autres formats de fichier contiennent de longues zones de zéro.

C'est typique des fichiers utilisés par les scientifiques et les ingénieurs logiciels.

Pour ceux qui veulent inspecter ces fichiers eux-même, ils sont téléchargeables ici: http://go.yurichev.com/17352.

Exercice

- http://challenges.re/50

6.1.4 Chiffrement simple utilisant un masque XOR

J'ai trouvé un vieux jeu de fiction interactif en plongeant profondément dans *if-archive*[3] :

```
The New Castle v3.5 - Text/Adventure Game
in the style of the original Infocom (tm)
type games, Zork, Collosal Cave (Adventure),
etc.  Can you solve the mystery of the
abandoned castle?
Shareware from Software Customization.
Software Customization [ASP] Version 3.5 Feb. 2000
```

Il est téléchargeable https://github.com/DennisYurichev/RE-for-beginners/blob/master/ff/XOR/mask_1/files/newcastle.tgzici.

Il y a un fichier à l'intérieur (appelé *castle.dbf*) qui est visiblement chiffré, mais pas avec un vrai algorithme de crypto, qui n'est pas non plus compressé, il s'agit plutôt de quelque chose de plus simple. Je ne vais même pas mesurer le niveau d'entropie (**??** on page ??) du fichier, car je suis sûr qu'il est bas. Voici à quoi il ressemble dans Midnight Commander:

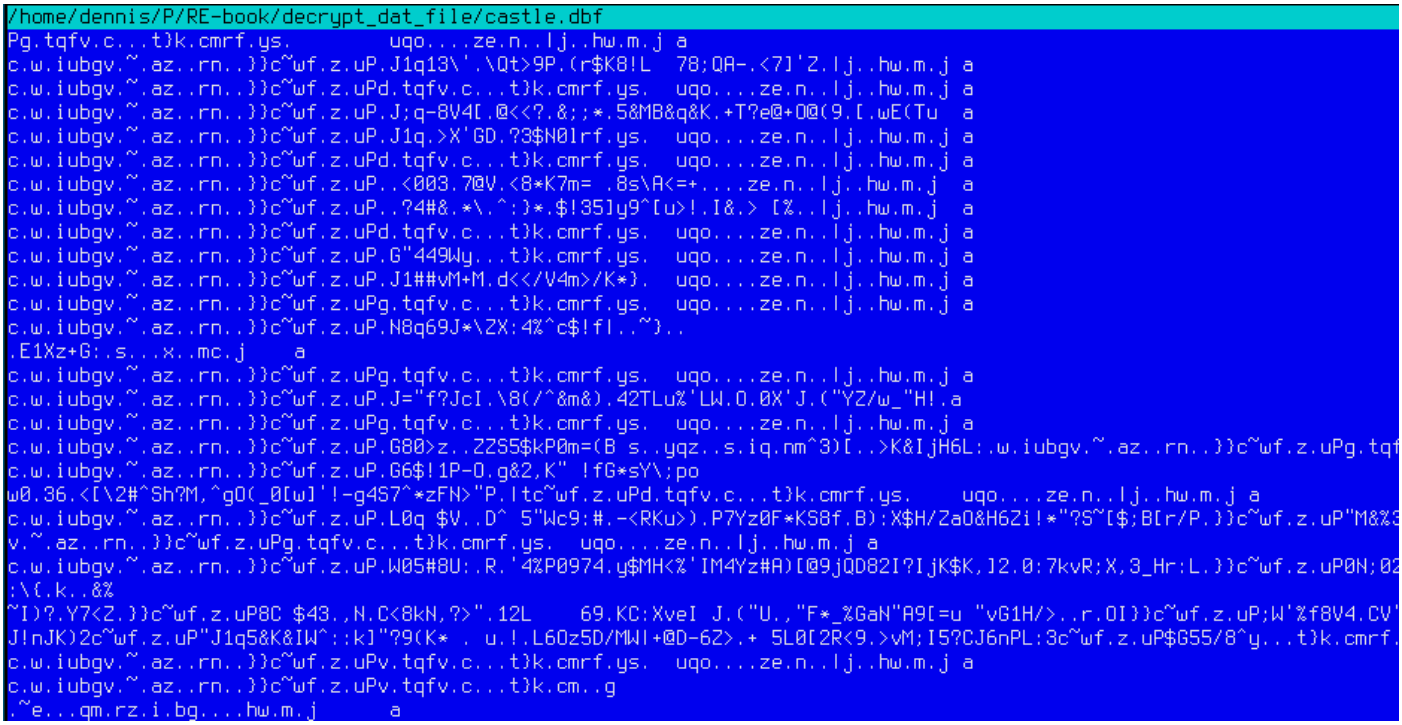


Fig. 6.8: Fichier chiffré dans Midnight Commander

Le fichier chiffré peut être téléchargé ici: https://github.com/DennisYurichev/RE-for-beginners/blob/master/ff/XOR/mask_1/files/castle.dbf.bz2.

Sera-t-il possible de le décrypter sans accéder au programme, en utilisant juste ce fichier?

Il y a clairement un pattern visible de chaînes répétées. Si un simple chiffrement avec un masque XOR a été appliqué, une répétition de telles chaînes en est une signature notable, car, il y avait probablement de longues suites (lacunes[4]) d'octets à zéro, qui, à tour de rôle, sont présentes dans de nombreux fichiers exécutables, tout comme dans des fichiers de données binaires.

Ici, je vais afficher le début du fichier en utilisant l'utilitaire UNIX *xxd* :

```
...

0000030: 09 61 0d 63 0f 77 14 69 75 62 67 76 01 7e 1d 61  .a.c.w.iubgv.~.a
0000040: 7a 11 0f 72 6e 03 05 7d 7d 63 7e 77 66 1e 7a 02  z..rn..}}c~wf.z.
0000050: 75 50 02 4a 31 71 31 33 5c 27 08 5c 51 74 3e 39  uP.J1q13\'.\Qt>9
0000060: 50 2e 28 72 24 4b 38 21 4c 09 37 38 3b 51 41 2d  P.(r$K8!L.78;QA-
```

[3] http://www.ifarchive.org/

[4] Comme dans https://en.wikipedia.org/wiki/Lacuna_(manuscripts)

```
0000070: 1c 3c 37 5d 27 5a 1c 7c 6a 10 14 68 77 08 6d 1a  .<7]'Z.|j..hw.m.

0000080: 6a 09 61 0d 63 0f 77 14 69 75 62 67 76 01 7e 1d  j.a.c.w.iubgv.~.
0000090: 61 7a 11 0f 72 6e 03 05 7d 7d 63 7e 77 66 1e 7a  az..rn..}}c~wf.z
00000a0 : 02 75 50 64 02 74 71 66 76 19 63 08 13 17 74 7d  .uPd.tqfv.c...t}
00000b0 : 6b 19 63 6d 72 66 0e 79 73 1f 09 75 71 6f 05 04  k.cmrf.ys..uqo..
00000c0 : 7f 1c 7a 65 08 6e 0e 12 7c 6a 10 14 68 77 08 6d  ..ze.n..|j..hw.m

00000d0 : 1a 6a 09 61 0d 63 0f 77 14 69 75 62 67 76 01 7e  .j.a.c.w.iubgv.~
00000e0 : 1d 61 7a 11 0f 72 6e 03 05 7d 7d 63 7e 77 66 1e  .az..rn..}}c~wf.
00000f0 : 7a 02 75 50 01 4a 3b 71 2d 38 56 34 5b 13 40 3c  z.uP.J;q-8V4[.@<
0000100: 3c 3f 19 26 3b 3b 2a 0e 35 26 4d 42 26 71 26 4b  <?.&;;*.5&MB&q&K
0000110: 04 2b 54 3f 65 40 2b 4f 40 28 39 10 5b 2e 77 45  .+T?e@+O@(9.[.wE

0000120: 28 54 75 09 61 0d 63 0f 77 14 69 75 62 67 76 01  (Tu.a.c.w.iubgv.
0000130: 7e 1d 61 7a 11 0f 72 6e 03 05 7d 7d 63 7e 77 66  ~.az..rn..}}c~wf
0000140: 1e 7a 02 75 50 02 4a 31 71 15 3e 58 27 47 44 17  .z.uP.J1q.>X'GD.
0000150: 3f 33 24 4e 30 6c 72 66 0e 79 73 1f 09 75 71 6f  ?3$N0lrf.ys..uqo
0000160: 05 04 7f 1c 7a 65 08 6e 0e 12 7c 6a 10 14 68 77  ....ze.n..|j..hw

...
```

Concentrons-nous sur la chaîne visible `iubgv` se répétant. En regardant ce dump, nous voyons clairement que la période de l'occurrence de la chaîne est 0x51 ou 81. La taille du fichier est 1658961, et est divisible par 81 (et il y a donc 20481 blocs).

Maintenant, je vais utiliser Mathematica pour l'analyse, y a-t-il des blocs de 81 octets se répètant dans le fichier? Je vais séparer le fichier d'entrée en blocs de 81 octets et ensuite utiliser la fonction *Tally[]*[5] qui compte simplement combien de fois un élément était présent dans la liste en entrée. La sortie de Tally n'est pas triée, donc je vais ajouter la fonction *Sort[]* pour trier le nombre d'occurrences par ordre décroissant.

```
input = BinaryReadList["/home/dennis/.../castle.dbf"];

blocks = Partition[input, 81];

stat = Sort[Tally[blocks], #1[[2]] > #2[[2]] &]
```

Et voici la sortie:

```
{{{80, 103, 2, 116, 113, 102, 118, 25, 99, 8, 19, 23, 116, 125, 107,
   25, 99, 109, 114, 102, 14, 121, 115, 31, 9, 117, 113, 111, 5, 4,
   127, 28, 122, 101, 8, 110, 14, 18, 124, 106, 16, 20, 104, 119, 8,
   109, 26, 106, 9, 97, 13, 99, 15, 119, 20, 105, 117, 98, 103, 118,
   1, 126, 29, 97, 122, 17, 15, 114, 110, 3, 5, 125, 125, 99, 126,
   119, 102, 30, 122, 2, 117}, 1739},
{{80, 100, 2, 116, 113, 102, 118, 25, 99, 8, 19, 23, 116,
   125, 107, 25, 99, 109, 114, 102, 14, 121, 115, 31, 9, 117, 113,
   111, 5, 4, 127, 28, 122, 101, 8, 110, 14, 18, 124, 106, 16, 20,
   104, 119, 8, 109, 26, 106, 9, 97, 13, 99, 15, 119, 20, 105, 117,
   98, 103, 118, 1, 126, 29, 97, 122, 17, 15, 114, 110, 3, 5, 125,
   125, 99, 126, 119, 102, 30, 122, 2, 117}, 1422},
{{80, 101, 2, 116, 113, 102, 118, 25, 99, 8, 19, 23, 116,
   125, 107, 25, 99, 109, 114, 102, 14, 121, 115, 31, 9, 117, 113,
   111, 5, 4, 127, 28, 122, 101, 8, 110, 14, 18, 124, 106, 16, 20,
   104, 119, 8, 109, 26, 106, 9, 97, 13, 99, 15, 119, 20, 105, 117,
   98, 103, 118, 1, 126, 29, 97, 122, 17, 15, 114, 110, 3, 5, 125,
   125, 99, 126, 119, 102, 30, 122, 2, 117}, 1012},
{{80, 120, 2, 116, 113, 102, 118, 25, 99, 8, 19, 23, 116,
   125, 107, 25, 99, 109, 114, 102, 14, 121, 115, 31, 9, 117, 113,
   111, 5, 4, 127, 28, 122, 101, 8, 110, 14, 18, 124, 106, 16, 20,
   104, 119, 8, 109, 26, 106, 9, 97, 13, 99, 15, 119, 20, 105, 117,
   98, 103, 118, 1, 126, 29, 97, 122, 17, 15, 114, 110, 3, 5, 125,
   125, 99, 126, 119, 102, 30, 122, 2, 117}, 377},

...

{{80, 2, 74, 49, 113, 21, 62, 88, 39, 71, 68, 23, 63, 51, 36, 78, 48,
```

[5] https://reference.wolfram.com/language/ref/Tally.html

```
   108, 114, 102, 14, 121, 115, 31, 9, 117, 113, 111, 5, 4, 127, 28,
   122, 101, 8, 110, 14, 18, 124, 106, 16, 20, 104, 119, 8, 109, 26,
   106, 9, 97, 13, 99, 15, 119, 20, 105, 117, 98, 103, 118, 1, 126,
   29, 97, 122, 17, 15, 114, 110, 3, 5, 125, 125, 99, 126, 119, 102,
   30, 122, 2, 117}, 1},
{{80, 1, 74, 59, 113, 45, 56, 86, 52, 91, 19, 64, 60, 60, 63,
   25, 38, 59, 59, 42, 14, 53, 38, 77, 66, 38, 113, 38, 75, 4, 43, 84,
    63, 101, 64, 43, 79, 64, 40, 57, 16, 91, 46, 119, 69, 40, 84, 117,
    9, 97, 13, 99, 15, 119, 20, 105, 117, 98, 103, 118, 1, 126, 29,
   97, 122, 17, 15, 114, 110, 3, 5, 125, 125, 99, 126, 119, 102, 30,
   122, 2, 117}, 1},
{{80, 2, 74, 49, 113, 49, 51, 92, 39, 8, 92, 81, 116, 62, 57,
   80, 46, 40, 114, 36, 75, 56, 33, 76, 9, 55, 56, 59, 81, 65, 45, 28,
    60, 55, 93, 39, 90, 28, 124, 106, 16, 20, 104, 119, 8, 109, 26,
   106, 9, 97, 13, 99, 15, 119, 20, 105, 117, 98, 103, 118, 1, 126,
   29, 97, 122, 17, 15, 114, 110, 3, 5, 125, 125, 99, 126, 119, 102,
   30, 122, 2, 117}, 1}}
```

La sortie de Tally est une liste de paires, chaque paire a un bloc de 81 octets et le nombre de fois qu'il apparaît dans le fichier. Nous voyons que le bloc le plus fréquent est le premier, il est apparu 1739 fois. Le second apparaît 1422 fois. Puis les autres: 1012 fois, 377 fois, etc. Les blocs de 81 octets qui ne sont apparus qu'une fois sont à la fin de la sortie.

Essayons de comparer ces blocs. Le premier et le second. Y a-t-il une fonction dans Mathematica qui compare les listes/tableaux? Certainement qu'il y en a une, mais dans un but didactique, je vais utiliser l'opération XOR pour la comparaison. En effet: si les octets dans deux tableaux d'entrée sont identiques, le résultat du XOR est 0. Si ils sont différents, le résultat sera différent de zéro.

Comparons le premier bloc (qui apparaît 1739 fois) et le second (qui apparaît 1422 fois):

```
In[]:= BitXor[stat[[1]][[1]], stat[[2]][[1]]]
Out[]= {0, 3, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, \
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, \
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, \
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0}
```

Ils ne diffèrent que par le second octet.

Comparons le second bloc (qui apparaît 1422 fois) et le troisième (qui apparaît 1012 fois):

```
In[]:= BitXor[stat[[2]][[1]], stat[[3]][[1]]]
Out[]= {0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, \
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, \
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, \
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0}
```

Ils ne diffèrent également que par le second octet.

Quoiqu'il en soit, essayons d'utiliser le bloc qui apparaît le plus comme une clef XOR et essayons de déchiffrer les quatre premiers blocs de 81 octets dans le fichier:

```
In[]:= key = stat[[1]][[1]]
Out[]= {80, 103, 2, 116, 113, 102, 118, 25, 99, 8, 19, 23, 116, \
125, 107, 25, 99, 109, 114, 102, 14, 121, 115, 31, 9, 117, 113, 111, \
5, 4, 127, 28, 122, 101, 8, 110, 14, 18, 124, 106, 16, 20, 104, 119, \
8, 109, 26, 106, 9, 97, 13, 99, 15, 119, 20, 105, 117, 98, 103, 118, \
1, 126, 29, 97, 122, 17, 15, 114, 110, 3, 5, 125, 125, 99, 126, 119, \
102, 30, 122, 2, 117}

In[]:= ToASCII[val_]  := If[val == 0, " ", FromCharacterCode[val, "PrintableASCII"]]

In[]:= DecryptBlockASCII[blk_]  := Map[ToASCII[#] &, BitXor[key, blk]]

In[]:= DecryptBlockASCII[blocks[[1]]]
Out[]= {" ", " ", " ", " ", " ", " ", " ", " ", " ", " ", " ", " ", " \
", " ", " ", " ", " ", " ", " ", " ", " ", " ", " ", " ", " ", " ", " ", " \
", " ", " ", " ", " ", " ", " ", " ", " ", " ", " ", " ", " ", " ", " ", " \
", " ", " ", " ", " ", " ", " ", " ", " ", " ", " ", " ", " ", " ", " ", " \
", " ", " ", " ", " ", " ", " ", " ", " ", " ", " ", " ", " ", " ", " ", " \
", " ", " ", " ", " ", " ", " ", " ", " ", " ", " ", " ", " ", " ", " ", " "}
```

```
In[]:= DecryptBlockASCII[blocks[[2]]]
Out[]= {" ", "e", "H", "E", " ", "W", "E", "E", "D", " ", "O", \
"F", " ", "C", "R", "I", "M", "E", " ", "B", "E", "A", "R", "S", " ", \
"B", "I", "T", "T", "E", "R", " ", "F", "R", "U", "I", "T", "?", \
" ", " ", " ", " ", " ", " ", " ", " ", " ", " ", " ", " ", " ", " ", \
" ", " ", " ", " ", " ", " ", " ", " ", " ", " ", " ", " ", " ", " ", \
" ", " ", " ", " ", " ", " ", " ", " ", " ", " ", " ", " ", " ", " ", \
" "}

In[]:= DecryptBlockASCII[blocks[[3]]]
Out[]= {" ", "?", " ", " ", " ", " ", " ", " ", " ", " ", " ", " \
", " ", " ", " ", " ", " ", " ", " ", " ", " ", " ", " ", " ", " ", " \
", " ", " ", " ", " ", " ", " ", " ", " ", " ", " ", " ", " ", " ", " \
", " ", " ", " ", " ", " ", " ", " ", " ", " ", " ", " ", " ", " ", " \
", " ", " ", " ", " ", " ", " ", " ", " ", " ", " ", " ", " ", " ", " \
", " ", " ", " ", " ", " ", " ", " ", " ", " ", " ", " ", " ", " ", " \
"}

In[]:= DecryptBlockASCII[blocks[[4]]]
Out[]= {" ", "f", "H", "O", " ", "K", "N", "O", "W", "S", " ", \
"W", "H", "A", "T", " ", "E", "V", "I", "L", " ", "L", "U", "R", "K", \
"S", " ", "I", "N", " ", "T", "H", "E", " ", "H", "E", "A", "R", "T", \
"S", " ", "O", "F", " ", "M", "E", "N", "?", " ", " ", " ", " ", " ", \
" ", " ", " ", " ", " ", " ", " ", " ", " ", " ", " ", " ", " ", " ", \
" ", " ", " ", " ", " ", " ", " ", " ", " ", " ", " ", " ", " ", " ", \
" "}
```

(J’ai remplacé les caractères non imprimables par « ? ».)

Donc nous voyons que le premier et le troisième blocs sont vides (ou presque vide), mais le second et le quatrième comportent clairement des mots/phrases en anglais. Ils semble que notre hypothèse à propos de la clef soit correct (au moins en partie). Cala signifie que le bloc de 81 octets qui apparaît le plus souvent dans le fichier peut être trouvé à des endroits comportant des séries d’octets à zéro ou quelque chose comme ça.

Essayons de déchiffrer le fichier entier:

```
DecryptBlock[blk_]  := BitXor[key, blk]

decrypted = Map[DecryptBlock[#] &, blocks];

BinaryWrite["/home/dennis/.../tmp", Flatten[decrypted]]

Close["/home/dennis/.../tmp"]
```

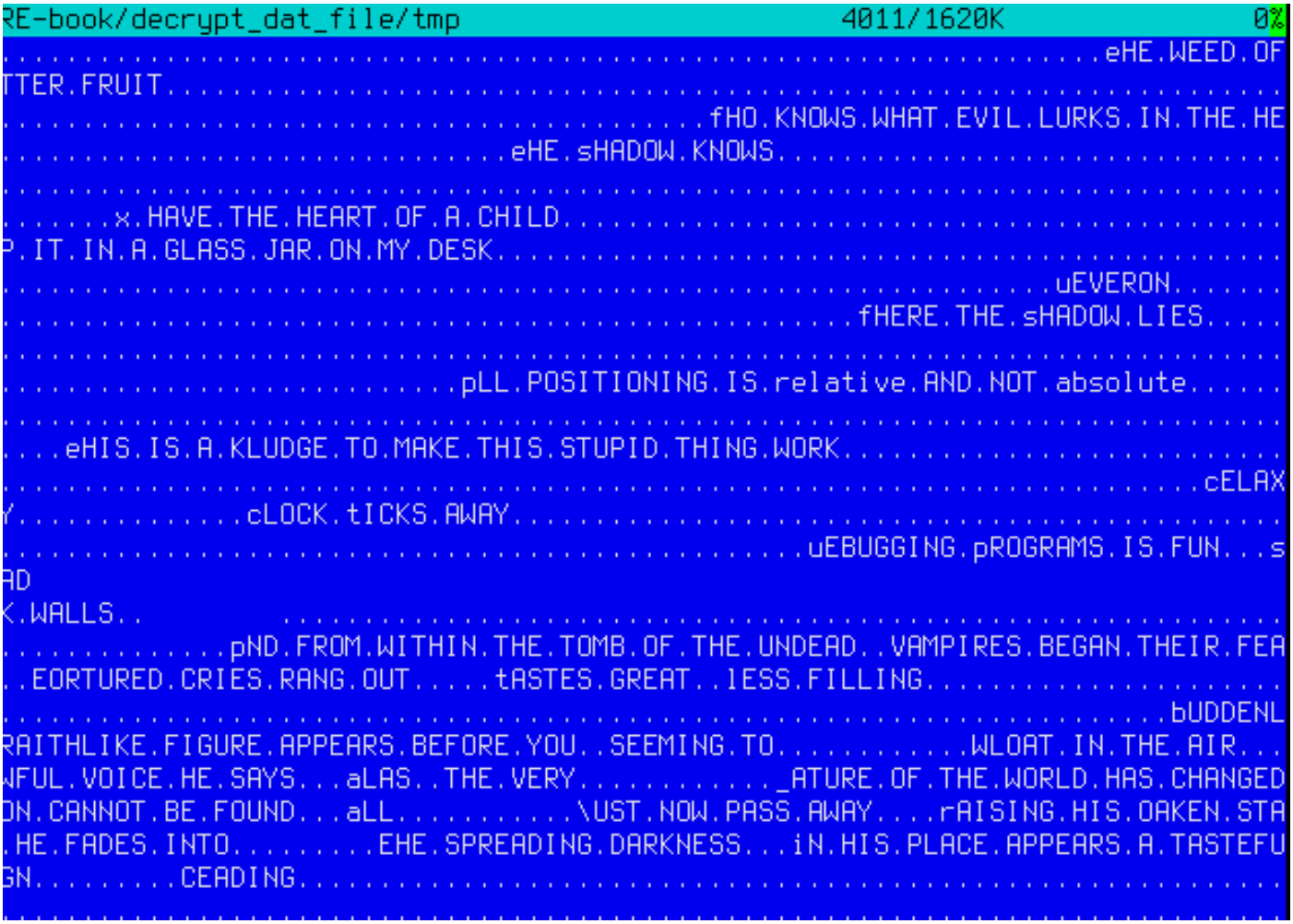


Fig. 6.9: Fichier déchiffré dans Midnight Commander, 1er essai

Ceci ressemble a des sortes de phrases en anglais d'un jeu, mais quelque chose ne va pas. Tout d'abord, la casse est inversée: les phrases et certains mots commence avec une minuscule, tandis que d'autres caractères sont en majuscule. De plus, certaines phrases commencent avec une mauvaise lettre. Regardez la toute première phrase: « eHE WEED OF CRIME BEARS BITTER FRUIT ». Que signifie « eHE » ? Ne devrait-on pas avoir « tHE » ici? Est-il possible que notre clef de déchiffrement ait un mauvais octet à cet endroit?

Regardons à nouveau le second bloc dans le fichier, la clef et le résultat décrypté:

```
In[]:= blocks[[2]]
Out[]= {80, 2, 74, 49, 113, 49, 51, 92, 39, 8, 92, 81, 116, 62, \
57, 80, 46, 40, 114, 36, 75, 56, 33, 76, 9, 55, 56, 59, 81, 65, 45, \
28, 60, 55, 93, 39, 90, 28, 124, 106, 16, 20, 104, 119, 8, 109, 26, \
106, 9, 97, 13, 99, 15, 119, 20, 105, 117, 98, 103, 118, 1, 126, 29, \
97, 122, 17, 15, 114, 110, 3, 5, 125, 125, 99, 126, 119, 102, 30, \
122, 2, 117}

In[]:= key
Out[]= {80, 103, 2, 116, 113, 102, 118, 25, 99, 8, 19, 23, 116, \
125, 107, 25, 99, 109, 114, 102, 14, 121, 115, 31, 9, 117, 113, 111, \
5, 4, 127, 28, 122, 101, 8, 110, 14, 18, 124, 106, 16, 20, 104, 119, \
8, 109, 26, 106, 9, 97, 13, 99, 15, 119, 20, 105, 117, 98, 103, 118, \
1, 126, 29, 97, 122, 17, 15, 114, 110, 3, 5, 125, 125, 99, 126, 119, \
102, 30, 122, 2, 117}

In[]:= BitXor[key, blocks[[2]]]
Out[]= {0, 101, 72, 69, 0, 87, 69, 69, 68, 0, 79, 70, 0, 67, 82, \
73, 77, 69, 0, 66, 69, 65, 82, 83, 0, 66, 73, 84, 84, 69, 82, 0, 70, \
82, 85, 73, 84, 14, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, \
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, \
0, 0, 0, 0}
```

L'octet chiffré est 2, l'octet de la clef est 103, $2 \oplus 103 = 101$ et 101 est le code ASCII du caractère « e ». A quoi devrait être égal l'octet de la clef, afin que le code ASCII résultant soit 116 (pour le caractère « t »)? $2 \oplus 116 = 118$, mettons 118 comme second octet de la clef ...

```
key = {80, 118, 2, 116, 113, 102, 118, 25, 99, 8, 19, 23, 116, 125,
  107, 25, 99, 109, 114, 102, 14, 121, 115, 31, 9, 117, 113, 111, 5,
  4, 127, 28, 122, 101, 8, 110, 14, 18, 124, 106, 16, 20, 104, 119, 8,
   109, 26, 106, 9, 97, 13, 99, 15, 119, 20, 105, 117, 98, 103, 118,
  1, 126, 29, 97, 122, 17, 15, 114, 110, 3, 5, 125, 125, 99, 126, 119,
   102, 30, 122, 2, 117}
```

...et déchiffrons le fichier à nouveau.

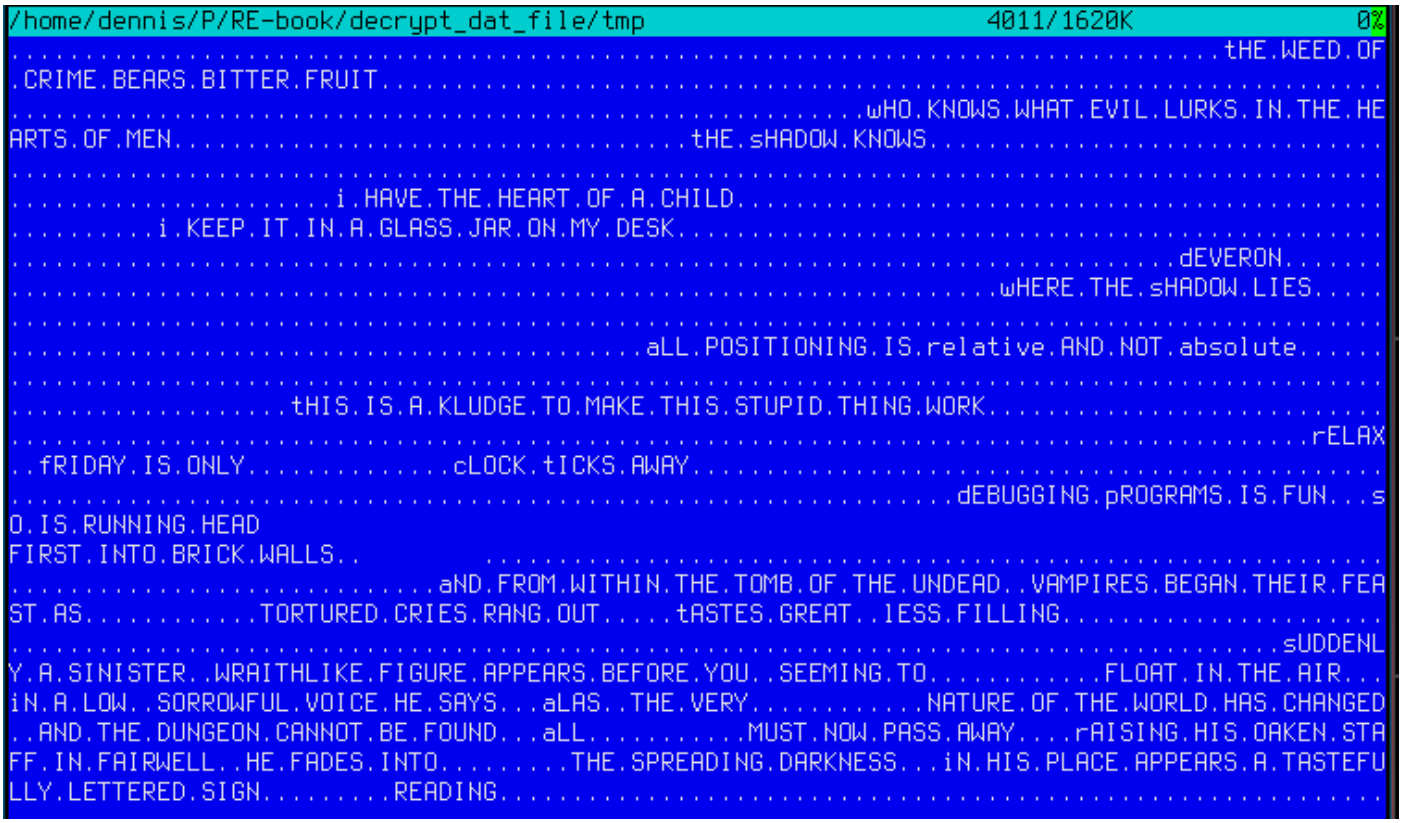


Fig. 6.10: Fichier déchiffré dans Midnight Commander, 2nd essai

Ouah, maintenant, la grammaire est correcte, toutes les phrases commencent avec une lettre correcte. Mais encore, l'inversion de la casse est suspecte. Pourquoi est-ce que le développeur les aurait écrites de cette façon? Peut-être que notre clef est toujours incorrecte?

En regardant la table ASCII, nous pouvons remarquer que les codes des lettres majuscules et des minuscules ne diffèrent que d'un bit (6ème bit en partant de 1, 0b100000):

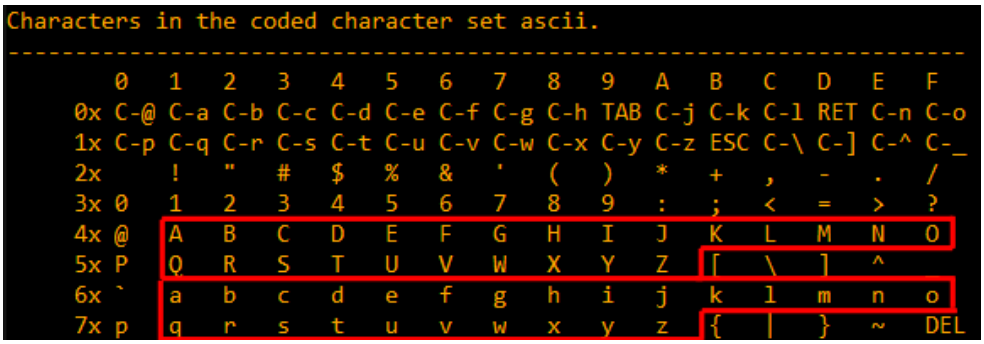


Fig. 6.11: table ASCII 7-bit dans Emacs

Cet octet avec seul le 6ème bit mis est 32 au format décimal. Mais 32 est le code ASCII de l'espace!

En effet, on peut changer la casse juste en XOR-ant le code ASCII d'un caractère avec 32 (plus à ce sujet: 3.16.3 on page 542).

Est-ce possible que les parties vides dans le fichier ne soient pas des octets à zéro, mais plutôt des espaces? Modifions notre clef XOR encore une fois (je vais appliquer Un XOR avec 32 à chaque octet de la clef):

```
(* "32" is scalar and "key" is vector, but that's OK *)

In[]:= key3 = BitXor[32, key]
Out[]= {112, 86, 34, 84, 81, 70, 86, 57, 67, 40, 51, 55, 84, 93, 75, \
57, 67, 77, 82, 70, 46, 89, 83, 63, 41, 85, 81, 79, 37, 36, 95, 60, \
90, 69, 40, 78, 46, 50, 92, 74, 48, 52, 72, 87, 40, 77, 58, 74, 41, \
65, 45, 67, 47, 87, 52, 73, 85, 66, 71, 86, 33, 94, 61, 65, 90, 49, \
47, 82, 78, 35, 37, 93, 93, 67, 94, 87, 70, 62, 90, 34, 85}

In[]:= DecryptBlock[blk_]  := BitXor[key3, blk]
```

Déchiffrons à nouveau le fichier d'entrée:

Fig. 6.12: Fichier déchiffré dans Midnight Commander, essai final

(Le fichier déchiffré est disponible https://github.com/DennisYurichev/RE-for-beginners/blob/master/ff/XOR/mask_1/files/decrypted.dat.bz2ici.)

Ceci est indiscutablement un fichier source correct. Oh, et nous voyons des nombres au début de chaque bloc. Ça doit être la source de notre clef XOR erronée. Il semble que le bloc de 81 octets le plus fréquent dans le fichier soit un bloc rempli avec des espaces et contenant le caractère « 1 » à la place du second octet. En effet, d'une façon ou d'une autre, de nombreux blocs sont entrelacés avec celui-ci.

Peut-être est-ce une sorte de remplissage pour les phrases/messages courts? D'autres blocs de 81 octets sont aussi remplis avec des blocs d'espaces, mais avec un chiffre différent, ainsi, ils ne diffèrent que du second octet.

C'est tout! Maintenant nous pouvons écrire un utilitaire pour chiffrer à nouveau le fichier, et peut-être le modifier avant.

Le fichier notebook de Mathematica est téléchargeable https://github.com/DennisYurichev/RE-for-beginne blob/master/ff/XOR/mask_1/files/XOR_mask_1.nbici.

Résumé: un tel chiffrement avec XOR n'est pas robuste du tout. Le développeur du jeu escomptait, probablement, empêcher les joueurs de chercher des informations sur le jeu, mais rien de plus sérieux. Néanmoins, un tel chiffrement est très populaire du fait de sa simplicité et de nombreux rétro-ingénieurs sont traditionnellement familier avec.

6.1.5 Chiffrement simple utilisant un masque XOR, cas II

J'ai un autre fichier chiffré, qui est clairement chiffré avec quelque chose de simple, comme un XOR:

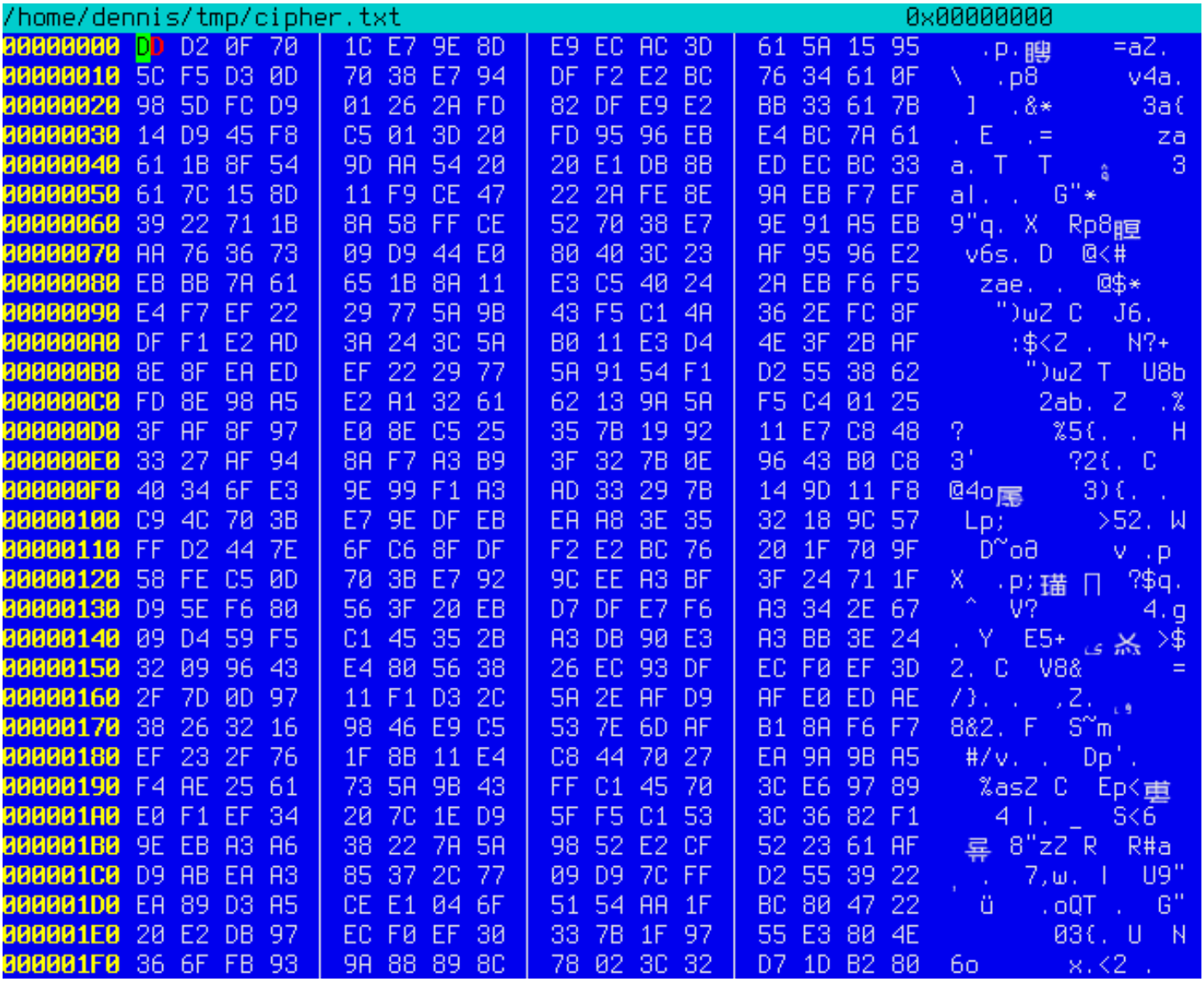


Fig. 6.13: Fichier chiffré dans Midnight Commander

Le fichier chiffré peut être téléchargé ici.

L'utilitaire Linux *ent* indique environ ~7.5 bits par octet, et ceci est un haut niveau d'entropie (**??** on page ??), proche de celui de fichiers compressés ou chiffrés correctement. Mais encore, nous distinguons clairement quelques patterns, il y a quelques blocs avec une taille de 17 octets, et nous pouvons voir des sortes d'échelles, se décalant d'un octet à chaque ligne de 16 octets.

On sait aussi que le texte clair est en anglais.

Maintenant, supposons que ce morceau de texte est chiffré par un simple XOR avec une clef de 17 octets.

J'ai essayé de repérer des blocs de 17 octets se répétant avec Mathematica, comme je l'ai fait dans l'exemple précédant (6.1.4 on page 613):

Listing 6.2: Mathematica

```
In[]:=input = BinaryReadList["/home/dennis/tmp/cipher.txt"];

In[]:=blocks = Partition[input, 17];

In[]:=Sort[Tally[blocks], #1[[2]] > #2[[2]] &]

Out[]:={{{248,128,88,63,58,175,159,154,232,226,161,50,97,127,3,217,80},1},
{{226,207,67,60,42,226,219,150,246,163,166,56,97,101,18,144,82},1},
{{228,128,79,49,59,250,137,154,165,236,169,118,53,122,31,217,65},1},
{{252,217,1,39,39,238,143,223,241,235,170,91,75,119,2,152,82},1},
{{244,204,88,112,59,234,151,147,165,238,170,118,49,126,27,144,95},1},
{{241,196,78,112,54,224,142,223,242,236,186,58,37,50,17,144,95},1},
{{176,201,71,112,56,230,143,151,234,246,187,118,44,125,8,156,17},1},
...
{{255,206,82,112,56,231,158,145,165,235,170,118,54,115,9,217,68},1},
{{249,206,71,34,42,254,142,154,235,247,239,57,34,113,27,138,88},1},
{{157,170,84,32,32,225,219,139,237,236,188,51,97,124,21,141,17},1},
{{248,197,1,61,32,253,149,150,235,228,188,122,97,97,27,143,84},1},
{{252,217,1,38,42,253,130,223,233,226,187,51,97,123,20,217,69},1},
{{245,211,13,112,56,231,148,223,242,226,188,118,52,97,15,152,93},1},
{{221,210,15,112,28,231,158,141,233,236,172,61,97,90,21,149,92},1}}
```

Pas de chance, chaque bloc de 17 octets est unique dans le fichier, et n'apparaît donc qu'une fois. Peut-être n'y a-t-il pas de zone de 17 octets à zéro, ou de zone contenant seulement des espaces. C'est possible toutefois: de telles séries d'espace peuvent être absentes dans des textes composés rigoureusement.

La première idée est d'essayer toutes les clefs de 17 octets possible et trouver celles qui donnent un résultat lisible après déchiffrement. La force brute n'est pas une option, car il y a 256^{17} clefs possible ($\sim 10^{40}$), c'est beaucoup trop. Mais il y a une bonne nouvelle: qui a dit que nous devons tester la clef de 17 octets en entier, pourquoi ne pas teste chaque octet séparémment? C'est possible en effet.

Maintenant, l'algorithme est:

- essayer chacun des 25 octets pour le premier octet de la clef;
- déchiffrer le 1er octet de chaque bloc de 17 octets du fichier;
- est-ce que tous les octets déchiffrés sont imprimable? garder un oeil dessus;
- faire de même pour chacun des 17 octets de la clef.

J'ai écrit le script Python suivant pour essayer cette idée:

Listing 6.3: Python script

```
each_Nth_byte=[""]*KEY_LEN

content=read_file(sys.argv[1])
# split input by 17-byte chunks :
all_chunks=chunks(content, KEY_LEN)
for c in all_chunks :
    for i in range(KEY_LEN) :
        each_Nth_byte[i]=each_Nth_byte[i] + c[i]

# try each byte of key
for N in range(KEY_LEN) :
    print "N=", N
    possible_keys=[]
    for i in range(256) :
        tmp_key=chr(i)*len(each_Nth_byte[N])
        tmp=xor_strings(tmp_key,each_Nth_byte[N])
        # are all characters in tmp[] are printable?
        if is_string_printable(tmp)==False :
            continue
        possible_keys.append(i)
    print possible_keys, "len=", len(possible_keys)
```

(La version complète du code source est ici.)

Voici sa sortie:

```
N= 0
[144, 145, 151] len= 3
N= 1
[160, 161] len= 2
N= 2
[32, 33, 38] len= 3
N= 3
[80, 81, 87] len= 3
N= 4
[78, 79] len= 2
N= 5
[142, 143] len= 2
N= 6
[250, 251] len= 2
N= 7
[254, 255] len= 2
N= 8
[130, 132, 133] len= 3
N= 9
[130, 131] len= 2
N= 10
[206, 207] len= 2
N= 11
[81, 86, 87] len= 3
N= 12
[64, 65] len= 2
N= 13
[18, 19] len= 2
N= 14
[122, 123] len= 2
N= 15
[248, 249] len= 2
N= 16
[48, 49] len= 2
```

Donc, il y a 2 ou 3 octets possible pour chaque octet de l clef de 17 octets. C'est mieux que 256 octets pour chaque octet, mais encore beaucoup trop. Il y a environ 1 million de clefs possible:

Listing 6.4: Mathematica

```
In[]:= 3*2*3*3*2*2*2*2*2*3*2*2*3*2*2*2*2*2
Out[]= 995328
```

Il est possible de les vérifier toutes, mais alors nous devons vérifier visuellement si le texte déchiffré à l'air d'un texte en anglais.

Prenons en compte le fait que nous avons à faire avec 1) un langage naturel 2) de l'anglais. Les langages naturels ont quelques caractéristiques statistiques importantes. Tout d'abord, le ponctuation et la longueur des mots. Quelle est la longueur moyenne des mots en anglais? Comptons les espaces dans quelques textes bien connus en anglais avec Mathematica.

Voici le fichier texte de « The Complete Works of William Shakespeare » provenant de la bibliothèque Gutenberg.

Listing 6.5: Mathematica

```
In[]:= input = BinaryReadList["/home/dennis/tmp/pg100.txt"];

In[]:= Tally[input]
Out[]= {{239, 1}, {187, 1}, {191, 1}, {84, 39878}, {104,
  218875}, {101, 406157}, {32, 1285884}, {80, 12038}, {114,
  209907}, {111, 282560}, {106, 2788}, {99, 67194}, {116,
  291243}, {71, 11261}, {117, 115225}, {110, 216805}, {98,
  46768}, {103, 57328}, {69, 42703}, {66, 15450}, {107, 29345}, {102,
  69103}, {67, 21526}, {109, 95890}, {112, 46849}, {108, 146532}, {87,
   16508}, {115, 215605}, {105, 199130}, {97, 245509}, {83,
  34082}, {44, 83315}, {121, 85549}, {13, 124787}, {10, 124787}, {119,
   73155}, {100, 134216}, {118, 34077}, {46, 78216}, {89, 9128}, {45,
  8150}, {76, 23919}, {42, 73}, {79, 33268}, {82, 29040}, {73,
```

```
  55893}, {72, 18486}, {68, 15726}, {58, 1843}, {65, 44560}, {49,
  982}, {50, 373}, {48, 325}, {91, 2076}, {35, 3}, {93, 2068}, {74,
  2071}, {57, 966}, {52, 107}, {70, 11770}, {85, 14169}, {78,
  27393}, {75, 6206}, {77, 15887}, {120, 4681}, {33, 8840}, {60,
  468}, {86, 3587}, {51, 343}, {88, 608}, {40, 643}, {41, 644}, {62,
  440}, {39, 31077}, {34, 488}, {59, 17199}, {126, 1}, {95, 71}, {113,
   2414}, {81, 1179}, {63, 10476}, {47, 48}, {55, 45}, {54, 73}, {64,
  3}, {53, 94}, {56, 47}, {122, 1098}, {90, 532}, {124, 33}, {38,
  21}, {96, 1}, {125, 2}, {37, 1}, {36, 2}}

In[]:= Length[input]/1285884 // N
Out[]= 4.34712
```

Il y a 1285884 espaces dans l'ensemble du fichier, et la fréquence de l'occurrence des espaces est de 1 par ~ 4.3 caractères.

Maintenant voici Alice's Adventures in Wonderland, par Lewis Carroll de la même bibliothèque:

Listing 6.6: Mathematica

```
In[]:= input = BinaryReadList["/home/dennis/tmp/pg11.txt"];

In[]:= Tally[input]
Out[]= {{239, 1}, {187, 1}, {191, 1}, {80, 172}, {114, 6398}, {111,
  9243}, {106, 222}, {101, 15082}, {99, 2815}, {116, 11629}, {32,
  27964}, {71, 193}, {117, 3867}, {110, 7869}, {98, 1621}, {103,
  2750}, {39, 2885}, {115, 6980}, {65, 721}, {108, 5053}, {105,
  7802}, {100, 5227}, {118, 911}, {87, 256}, {97, 9081}, {44,
  2566}, {121, 2442}, {76, 158}, {119, 2696}, {67, 185}, {13,
  3735}, {10, 3735}, {84, 571}, {104, 7580}, {66, 125}, {107,
  1202}, {102, 2248}, {109, 2245}, {46, 1206}, {89, 142}, {112,
  1796}, {45, 744}, {58, 255}, {68, 242}, {74, 13}, {50, 12}, {53,
  13}, {48, 22}, {56, 10}, {91, 4}, {69, 313}, {35, 1}, {49, 68}, {93,
   4}, {82, 212}, {77, 222}, {57, 11}, {52, 10}, {42, 88}, {83,
  288}, {79, 234}, {70, 134}, {72, 309}, {73, 831}, {85, 111}, {78,
  182}, {75, 88}, {86, 52}, {51, 13}, {63, 202}, {40, 76}, {41,
  76}, {59, 194}, {33, 451}, {113, 135}, {120, 170}, {90, 1}, {122,
  79}, {34, 135}, {95, 4}, {81, 85}, {88, 6}, {47, 24}, {55, 6}, {54,
  7}, {37, 1}, {64, 2}, {36, 2}}

In[]:= Length[input]/27964 // N
Out[]= 5.99049
```

Le résultat est différent, sans soute à cause d'un formatage des textes différents (indentation ou remplissage).

Ok, donc supposons que la fréquence moyenne de l'espace en anglais est de 1 espace tous les 4..7 caractères.

Maintenant, encore une bonne nouvelle: nous pouvons mesurer la fréquence des espaces au fur et à mesure du déchiffrement de notre fichier. Maintenant je compte les espaces dans chaque *slice* et jette les clefs de 1 octets qui produise un résultat avec un nombre d'espaces trop petit (ou trop grand, mais c'est presque impossible avec une si petite clef):

Listing 6.7: Python script

```
each_Nth_byte=[""]*KEY_LEN

content=read_file(sys.argv[1])
# split input by 17-byte chunks :
all_chunks=chunks(content, KEY_LEN)
for c in all_chunks :
    for i in range(KEY_LEN) :
        each_Nth_byte[i]=each_Nth_byte[i] + c[i]

# try each byte of key
for N in range(KEY_LEN) :
    print "N=", N
    possible_keys=[]
    for i in range(256) :
        tmp_key=chr(i)*len(each_Nth_byte[N])
```

```
        tmp=xor_strings(tmp_key,each_Nth_byte[N])
        # are all characters in tmp[] are printable?
        if is_string_printable(tmp)==False :
            continue
        # count spaces in decrypted buffer :
        spaces=tmp.count(' ')
        if spaces==0:
            continue
        spaces_ratio=len(tmp)/spaces
        if spaces_ratio<4:
            continue
        if spaces_ratio>7:
            continue
        possible_keys.append(i)
    print possible_keys, "len=", len(possible_keys)
```

(La version complète du code source se trouve ici.)

Ceci nous donne un seul octet possible pour chaque octet de la clef:

```
N= 0
[144] len= 1
N= 1
[160] len= 1
N= 2
[33] len= 1
N= 3
[80] len= 1
N= 4
[79] len= 1
N= 5
[143] len= 1
N= 6
[251] len= 1
N= 7
[255] len= 1
N= 8
[133] len= 1
N= 9
[131] len= 1
N= 10
[207] len= 1
N= 11
[86] len= 1
N= 12
[65] len= 1
N= 13
[18] len= 1
N= 14
[122] len= 1
N= 15
[249] len= 1
N= 16
[49] len= 1
```

Vérifions cette clef dans Mathematica:

Listing 6.8: Mathematica

```
In[]:= input = BinaryReadList["/home/dennis/tmp/cipher.txt"];

In[]:= blocks = Partition[input, 17];

In[]:= key = {144, 160, 33, 80, 79, 143, 251, 255, 133, 131, 207, 86, 65, 18, 122, 249, 49};

In[]:= EncryptBlock[blk_]  := BitXor[key, blk]

In[]:= encrypted = Map[EncryptBlock[#] &, blocks];

In[]:= BinaryWrite["/home/dennis/tmp/plain2.txt", Flatten[encrypted]]
```

```
In[]:= Close["/home/dennis/tmp/plain2.txt"]
```

Et le texte brut est:

```
Mr. Sherlock Holmes, who was usually very late in the mornings, save
upon those not infrequent occasions when he was up all night, was seated
at the breakfast table. I stood upon the hearth-rug and picked up the
stick which our visitor had left behind him the night before. It was a
fine, thick piece of wood, bulbous-headed, of the sort which is known as
a "Penang lawyer." Just under the head was a broad silver band nearly
an inch across. "To James Mortimer, M.R.C.S., from his friends of the
C.C.H.," was engraved upon it, with the date "1884." It was just such a
stick as the old-fashioned family practitioner used to carry--dignified,
solid, and reassuring.

"Well, Watson, what do you make of it?"

Holmes was sitting with his back to me, and I had given him no sign of
my occupation.

...
```

(La version complète de ce texte se trouve ici.)

Le texte semble correct. Oui, j'ai créé cet exemple de toutes pièces et j'ai choisi un texte très connu de Conan Doyle, mais c'est très proche de ce que j'ai eu à faire il y a quelques temps.

Autres idées à envisager

Si nous échouions avec le comptage des espaces, il y a d'autres idées à essayer:

- Prenons en considération le fait que les lettres minuscules sont plus fréquentes que celles en majuscule.
- Analyse des fréquences.
- Il y a aussi une bonne technique pour détecter le langage d'un texte: les trigrammes. Chaque langage possède des triplets de lettres fréquences, qui peuvent être « the » et « tha » en anglais. En lire plus à ce sujet: N-Gram-Based Text Categorization, http://code.activestate.com/recipes/326576/. Fait suffisamment intéressant, la détection des trigrammes peut être utilisée lorsque vous décryptez un texte chiffré progressivement, comme dans cet exemple (yous devez juste tester les 3 caractères décryptez adjacents).

  Pour les systèmes non-latin encodés en UTF-8, les choses peuvent être plus simples. Par exemple, les textes en russe encodés en UTF-8 ont chaque octet intercalé avec des octets 0xD0/0xD1. C'est parce que les caractères cyrilliques sont situés dans le 4ème bloc de la table Unicode. D'autres systèmes d'écriture on leurs propres blocs.

6.2 Fichier de sauvegarde du jeu Millenium

« Millenium Return to Earth » est un ancien jeu DOS (1991), qui vous permet d'extraire des ressources, de construire des vaisseaux, de les équiper et de les envoyer sur d'autres planètes, et ainsi de suite[6].

Comme beaucoup d'autres jeux, il vous permet de sauvegarder l'état du jeu dans un fichier.

Regardons si l'on peut y trouver quelque chose.

[6] Il peut être téléchargé librement ici

Donc, il y a des mines dans le jeu. Sur certaines planêtes, les mines rapportent plus vite, sur d'autres, moins vite. L'ensemble des ressources est aussi différent.

Ici, nous pouvons voir quelles ressources sont actuellement extraites.

Fig. 6.14: Mine: état 1

Sauvegardons l'état du jeu. C'est un fichier de 9538 octets.

Attendons quelques « jours » dans le jeu, et maintenant, nous avons plus de ressources extraites des mines.

Fig. 6.15: Mine: état 2

Sauvegardons à nouveau l'état du jeu.

Maintenant, essayons juste de comparer au niveau binaire les fichiers de sauvegarde en utilisant le simple utilitaire DOS/Windows FC:

```
...> FC /b 2200save.i.v1 2200SAVE.I.V2

Comparing files 2200save.i.v1 and 2200SAVE.I.V2
00000016: 0D 04
00000017: 03 04
0000001C : 1F 1E
00000146: 27 3B
00000BDA : 0E 16
00000BDC : 66 9B
00000BDE : 0E 16
00000BE0 : 0E 16
00000BE6 : DB 4C
00000BE7 : 00 01
00000BE8 : 99 E8
00000BEC : A1 F3
00000BEE : 83 C7
00000BFB : A8 28
00000BFD : 98 18
00000BFF : A8 28
00000C01 : A8 28
00000C07 : D8 58
00000C09 : E4 A4
00000C0D : 38 B8
00000C0F : E8 68
...
```

La sortie est incomplète ici, il y a plus de différences, mais j'ai tronqué le résultat pour montrer ce qu'il y a de plus intéressant.

Dans le premier état, nous avons 14 « unités » d'hydrogène et 102 « unités » d'oxygène.

Nous avons respectivement 22 et 155 « unités » dans le second état. Si ces valeurs sont sauvées dans le fichier de sauvegarde, nous devrions les voir dans la différence. Et en effet, nous les voyons. Il y a 0x0E (14) à la position 0xBDA et cette valeur est à 0x16 (22) dans la nouvelle version du fichier. Ceci est probablement l'hydrogène. Il y a 0x66 (102) à la position 0xBDC dans la vieille version et x9B (155) dans la nouvelle version du fichier. Il semble que ça soit l'oxygène.

Les deux fichiers sont disponibles sur le site web pour ceux qui veulent les inspecter (ou expérimenter) plus: beginners.re.

Voici la nouvelle version du fichier ouverte dans Hiew, j'ai marqué les valeurs relatives aux ressources extraites dans le jeu:

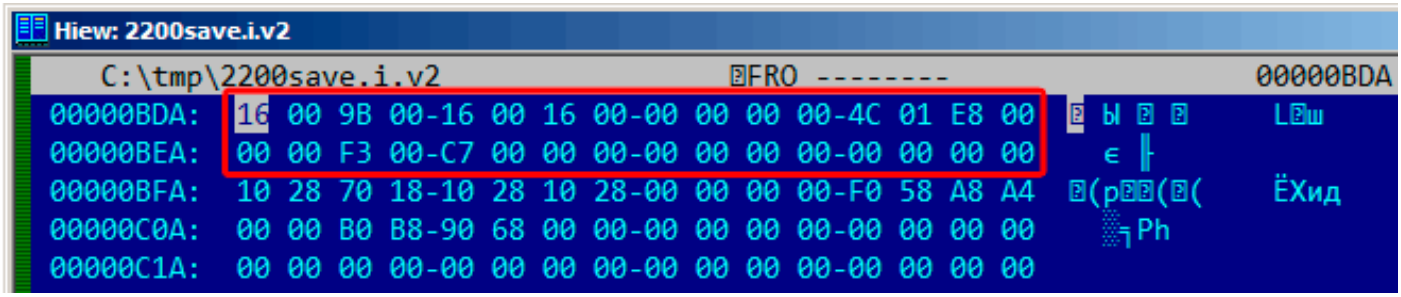


Fig. 6.16: Hiew: état 1

Vérifions chacune d'elles.

Ce sont clairement des valeurs 16-bits: ce n'est pas étonnant pour un logiciel DOS 16-bit où le type *int* fait 16-bit.

Vérifions nos hypothèses. Nous allons écrire la valeur 1234 (0x4D2) à la première position (ceci doit être l'hydrogène):

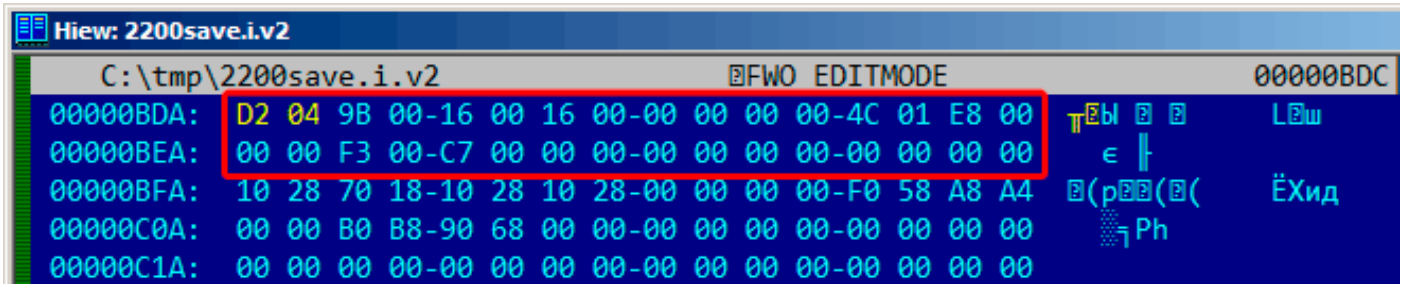


Fig. 6.17: Hiew: écrivons 1234 (0x4D2) ici

Puis nous chargeons le fichier modifié dans le jeu et jettons un coup d'oeil aux statistiques des mines:

Fig. 6.18: Vérifions la valeur pour l'hydrogène

Donc oui, c'est bien ça.

Maintenant essayons de finir le jeu le plus vite possible, mettons les valeurs maximales partout:

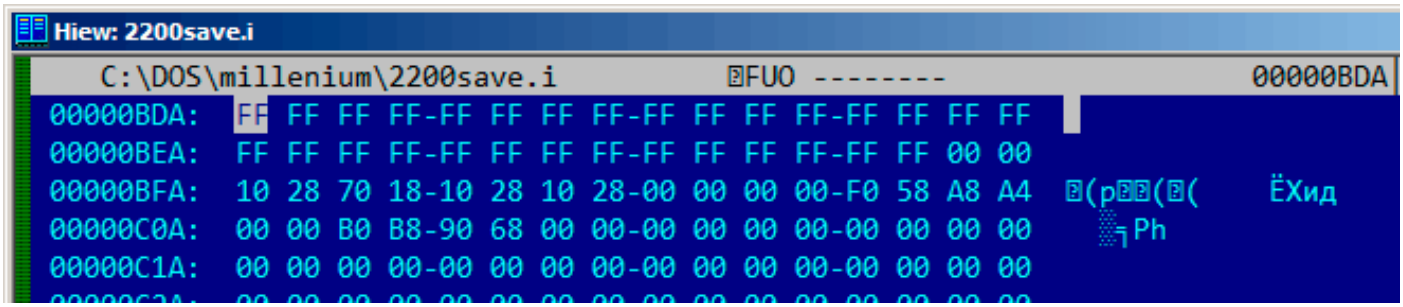


Fig. 6.19: Hiew: mettons les valeurs maximales

0xFFFF représente 65535, donc oui, nous avons maintenant beaucoup de ressources:

Fig. 6.20: Toutes les ressources sont en effet à 65535 (0xFFFF)

Laissons passer quelques « jours » dans le jeu et oups! Nous avons un niveau plus bas pour quelques ressources:

Fig. 6.21: Dépassement des variables de ressource

C'est juste un dépassement.

Le développeur du jeu n'a probablement pas pensé à un niveau aussi élevé de ressources, donc il n'a pas dû mettre des tests de dépassement, mais les mines « travaillent » dans le jeu, des ressources sont extraites, c'est pourquoi il y a des dépassements. Apparemment, c'est une mauvaise idée d'être aussi avide.

Il y a sans doute beaucoup plus de valeurs sauvées dans ce fichier.

Ceci est donc une méthode très simple de tricher dans les jeux. Les fichiers des meilleurs scores peuvent souvent être modifiés comme ceci.

Plus d'informations sur la comparaison des fichiers et des snapshots de mémoire: **??** on page ??.

6.3 *fortune* programme d'indexation de fichier

(Cette partie est tout d'abord apparue sur mon blog le 25 avril 2015.)

fortune est un programme UNIX bien connu qui affiche une phrase aléatoire depuis une collection. Certains geeks ont souvent configuré leur système de telle manière que *fortune* puisse être appelé après la connexion. *fortune* prend les phrases depuis des fichiers texte se trouvant dans */usr/share/games/fortunes* (sur Ubuntu Linux). Voici un exemple (fichier texte « fortunes »):

```
A day for firm decisions!!!!!  Or is it?
%
A few hours grace before the madness begins again.
%
A gift of a flower will soon be made to you.
%
```

```
A long-forgotten loved one will appear soon.

Buy the negatives at any price.
%
A tall, dark stranger will have more fun than you.
%
...
```

Donc, il s'agit juste de phrases, parfois sur plusieurs lignes, séparées par le signe pourcent. La tâche du programme *fortune* est de trouver une phrase aléatoire et de l'afficher. Afin de réaliser ceci, il doit scanner le fichier texte complet, compter les phrases, en choisir une aléatoirement et l'afficher. Mais le fichier texte peut être très gros, et même sur les ordinateurs modernes, cet algorithme naïf est du gaspillage de ressource. La façon simple de procéder est de garder un fichier index binaire contenant l'offset de chaque phrase du fichier texte. Avec le fichier index, le programme *fortune* peut fonctionner beaucoup plus vite: il suffit de choisir un index aléatoirement, prendre son offset, se déplacer à cet offset dans le fichier texte et d'y lire la phrase. C'est ce qui est effectivement fait dans le programme *fortune*. Examinons ce qu'il y a dans ce fichier index (ce sont les fichiers .dat dans le même répertoire) dans un éditeur hexadécimal. Ce programme est open-source bien sûr, mais intentionnellement, je ne vais pas jeter un coup d'œil dans le code source.

```
% od -t x1 --address-radix=x fortunes.dat
000000 00 00 00 02 00 00 01 af 00 00 00 bb 00 00 00 0f
000010 00 00 00 00 25 00 00 00 00 00 00 00 00 00 00 2b
000020 00 00 00 60 00 00 00 8f 00 00 00 df 00 00 01 14
000030 00 00 01 48 00 00 01 7c 00 00 01 ab 00 00 01 e6
000040 00 00 02 20 00 00 02 3b 00 00 02 7a 00 00 02 c5
000050 00 00 03 04 00 00 03 3d 00 00 03 68 00 00 03 a7
000060 00 00 03 e1 00 00 04 19 00 00 04 2d 00 00 04 7f
000070 00 00 04 ad 00 00 04 d5 00 00 05 05 00 00 05 3b
000080 00 00 05 64 00 00 05 82 00 00 05 ad 00 00 05 ce
000090 00 00 05 f7 00 00 06 1c 00 00 06 61 00 00 06 7a
0000a0 00 00 06 d1 00 00 07 0a 00 00 07 53 00 00 07 9a
0000b0 00 00 07 f8 00 00 08 27 00 00 08 59 00 00 08 8b
0000c0 00 00 08 a0 00 00 08 c4 00 00 08 e1 00 00 08 f9
0000d0 00 00 09 27 00 00 09 43 00 00 09 79 00 00 09 a3
0000e0 00 00 09 e3 00 00 0a 15 00 00 0a 4d 00 00 0a 5e
0000f0 00 00 0a 8a 00 00 0a a6 00 00 0a bf 00 00 0a ef
000100 00 00 0b 18 00 00 0b 43 00 00 0b 61 00 00 0b 8e
000110 00 00 0b cf 00 00 0b fa 00 00 0c 3b 00 00 0c 66
000120 00 00 0c 85 00 00 0c b9 00 00 0c d2 00 00 0d 02
000130 00 00 0d 3b 00 00 0d 67 00 00 0d ac 00 00 0d e0
000140 00 00 0e 1e 00 00 0e 67 00 00 0e a5 00 00 0e da
000150 00 00 0e ff 00 00 0f 43 00 00 0f 8a 00 00 0f bc
000160 00 00 0f e5 00 00 10 1e 00 00 10 63 00 00 10 9d
000170 00 00 10 e3 00 00 11 10 00 00 11 46 00 00 11 6c
000180 00 00 11 99 00 00 11 cb 00 00 11 f5 00 00 12 32
000190 00 00 12 61 00 00 12 8c 00 00 12 ca 00 00 13 87
0001a0 00 00 13 c4 00 00 13 fc 00 00 14 1a 00 00 14 6f
0001b0 00 00 14 ae 00 00 14 de 00 00 15 1b 00 00 15 55
0001c0 00 00 15 a6 00 00 15 d8 00 00 16 0f 00 00 16 4e
...
```

Sans aide particulière, nous pouvons voir qu'il y a quatre éléments de 4 octets sur chaque ligne de 16 octets. Peut-être que c'est notre tableau d'index. J'essaye de charger tout le fichier comme un tableau d'entier 32-bit dans Wolfram Mathematica:

```
In[]:= BinaryReadList["c :/tmp1/fortunes.dat", "UnsignedInteger32"]

Out[]= {33554432, 2936078336, 3137339392, 251658240, 0, 37, 0, \
721420288, 1610612736, 2399141888, 3741319168, 335609856, 1208025088, \
2080440320, 2868969472, 3858825216, 537001984, 989986816, 2046951424, \
3305242624, 67305472, 1023606784, 1745027072, 2801991680, 3775070208, \
419692544, 755236864, 2130968576, 2902720512, 3573809152, 84213760, \
990183424, 1678049280, 2181365760, 2902786048, 3456434176, \
4144300032, 470155264, 1627783168, 2047213568, 3506831360, 168230912, \
1392967680, 2584150016, 4161208320, 654835712, 1493696512, \
2332557312, 2684878848, 3288858624, 3775397888, 4178051072, \
...
```

Nope, quelque chose est faux, les nombres sont étrangement gros. Mais retournons à la sortie de *od* : chaque élément de 4 octets a deux octets nuls et deux octets non nuls. Donc les offsets (au moins au début du fichier) sont au maximum 16-bit. Peut-être qu'un endianness différent est utilisé dans le fichier? L'endianness par défaut dans Mathematica est little-endian, comme utilisé dans les CPUs Intel. Maintenant, je le change en big-endian:

```
In[]:= BinaryReadList["c :/tmp1/fortunes.dat", "UnsignedInteger32",
 ByteOrdering -> 1]

Out[]= {2, 431, 187, 15, 0, 620756992, 0, 43, 96, 143, 223, 276, \
328, 380, 427, 486, 544, 571, 634, 709, 772, 829, 872, 935, 993, \
1049, 1069, 1151, 1197, 1237, 1285, 1339, 1380, 1410, 1453, 1486, \
1527, 1564, 1633, 1658, 1745, 1802, 1875, 1946, 2040, 2087, 2137, \
2187, 2208, 2244, 2273, 2297, 2343, 2371, 2425, 2467, 2531, 2581, \
2637, 2654, 2698, 2726, 2751, 2799, 2840, 2883, 2913, 2958, 3023, \
3066, 3131, 3174, 3205, 3257, 3282, 3330, 3387, 3431, 3500, 3552, \
...
```

Oui, c'est quelque chose de lisible. Je choisi un élément au hasard (3066) qui s'écrit 0xBFA en format hexadécimal. J'ouvre le fichier texte 'fortunes' dans un éditeur hexadécimal, je met l'offset 0xBFA et je vois cette phrase:

```
% od -t x1 -c --skip-bytes=0xbfa --address-radix=x fortunes
000bfa  44  6f  20  77  68  61  74  20  63  6f  6d  65  73  20  6e  61
         D   o       w   h   a   t       c   o   m   e   s       n   a
000c0a  74  75  72  61  6c  6c  79  2e  20  20  53  65  65  74  68  65
         t   u   r   a   l   l   y   .           S   e   e   t   h   e
000c1a  20  61  6e  64  20  66  75  6d  65  20  61  6e  64  20  74  68
             a   n   d       f   u   m   e       a   n   d       t   h
....
```

Ou:

```
Do what comes naturally.  Seethe and fume and throw a tantrum.
%
```

D'autres offsets peuvent aussi être essayés, oui, ils sont valides.

Je peux aussi vérifier dans Mathematica que chaque élément consécutif est plus grand que le précédent. I.e., les éléments du tableau sont croissants. Dans le jargon mathématiques, ceci est appelé *fonction monotone strictement croissante*.

```
In[]:= Differences[input]

Out[]= {429, -244, -172, -15, 620756992, -620756992, 43, 53, 47, \
80, 53, 52, 52, 47, 59, 58, 27, 63, 75, 63, 57, 43, 63, 58, 56, 20, \
82, 46, 40, 48, 54, 41, 30, 43, 33, 41, 37, 69, 25, 87, 57, 73, 71, \
94, 47, 50, 50, 21, 36, 29, 24, 46, 28, 54, 42, 64, 50, 56, 17, 44, \
28, 25, 48, 41, 43, 30, 45, 65, 43, 65, 43, 31, 52, 25, 48, 57, 44, \
69, 52, 62, 73, 62, 53, 37, 68, 71, 50, 41, 57, 69, 58, 70, 45, 54, \
38, 45, 50, 42, 61, 47, 43, 62, 189, 61, 56, 30, 85, 63, 48, 61, 58, \
81, 50, 55, 63, 83, 80, 49, 42, 94, 54, 67, 81, 52, 57, 68, 43, 28, \
120, 64, 53, 81, 33, 82, 88, 29, 61, 32, 75, 63, 70, 47, 101, 60, 79, \
33, 48, 65, 35, 59, 47, 55, 22, 43, 35, 102, 53, 80, 65, 45, 31, 29, \
69, 32, 25, 38, 34, 35, 49, 59, 39, 41, 18, 43, 41, 83, 37, 31, 34, \
59, 72, 72, 81, 77, 53, 53, 50, 51, 45, 53, 39, 70, 54, 103, 33, 70, \
51, 95, 67, 54, 55, 65, 61, 54, 54, 53, 45, 100, 63, 48, 65, 71, 23, \
28, 43, 51, 61, 101, 65, 39, 78, 66, 43, 36, 56, 40, 67, 92, 65, 61, \
31, 45, 52, 94, 82, 82, 91, 46, 76, 55, 19, 58, 68, 41, 75, 30, 67, \
92, 54, 52, 108, 60, 56, 76, 41, 79, 54, 65, 74, 112, 76, 47, 53, 61, \
66, 53, 28, 41, 81, 75, 69, 89, 63, 60, 18, 18, 50, 79, 92, 37, 63, \
88, 52, 81, 60, 80, 26, 46, 80, 64, 78, 70, 75, 46, 91, 22, 63, 46, \
34, 81, 75, 59, 62, 66, 74, 76, 111, 55, 73, 40, 61, 55, 38, 56, 47, \
78, 81, 62, 37, 41, 60, 68, 40, 33, 54, 34, 41, 36, 49, 44, 68, 51, \
50, 52, 36, 53, 66, 46, 41, 45, 51, 44, 44, 33, 72, 40, 71, 57, 55, \
39, 66, 40, 56, 68, 43, 88, 78, 30, 54, 64, 36, 55, 35, 88, 45, 56, \
76, 61, 66, 29, 76, 53, 96, 36, 46, 54, 28, 51, 82, 53, 60, 77, 21, \
84, 53, 43, 104, 85, 50, 47, 39, 66, 78, 81, 94, 70, 49, 67, 61, 37, \
51, 91, 99, 58, 51, 49, 46, 68, 72, 40, 56, 63, 65, 41, 62, 47, 41, \
43, 30, 43, 67, 78, 80, 101, 61, 73, 70, 41, 82, 69, 45, 65, 38, 41, \
```

```
57, 82, 66}
```

Comme on le voit, excepté les 6 premières valeurs (qui appartiennent sans doute à l'entête du fichier d'index), tous les nombres sont en fait la longueur des phrases de texte (l'offset de la phrase suivante moins l'offset de la phrase courante est la longueur de la phrase courante).

Il est très important de garder à l'esprit que l'endianness peut être confondu avec un début de tableau incorrect. En effet, dans la sortie d'*od* nous voyons que chaque élément débutait par deux zéros. Mais lorsque nous décalons les des octets de chaque côté, nous pouvons interpréter ce tableau comme little-endian:

```
% od -t x1 --address-radix=x --skip-bytes=0x32 fortunes.dat
000032 01 48 00 00 01 7c 00 00 01 ab 00 00 01 e6 00 00
000042 02 20 00 00 02 3b 00 00 02 7a 00 00 02 c5 00 00
000052 03 04 00 00 03 3d 00 00 03 68 00 00 03 a7 00 00
000062 03 e1 00 00 04 19 00 00 04 2d 00 00 04 7f 00 00
000072 04 ad 00 00 04 d5 00 00 05 05 00 00 05 3b 00 00
000082 05 64 00 00 05 82 00 00 05 ad 00 00 05 ce 00 00
000092 05 f7 00 00 06 1c 00 00 06 61 00 00 06 7a 00 00
0000a2 06 d1 00 00 07 0a 00 00 07 53 00 00 07 9a 00 00
0000b2 07 f8 00 00 08 27 00 00 08 59 00 00 08 8b 00 00
0000c2 08 a0 00 00 08 c4 00 00 08 e1 00 00 08 f9 00 00
0000d2 09 27 00 00 09 43 00 00 09 79 00 00 09 a3 00 00
0000e2 09 e3 00 00 0a 15 00 00 0a 4d 00 00 0a 5e 00 00
...
```

Si nous interprétons ce tableau en little-endian, le premier élément est 0x4801, le second est 0x7C01, etc. La partie 8-bit haute de chacune de ces valeurs 16-bit nous semble être aléatoire, et la partie 8-bit basse semble être ascendante.

Mais je suis sûr que c'est un tableau en big-endian, car le tout dernier élément 32-bit du fichier est big-endian. (*00 00 5f c4* ici):

```
% od -t x1 --address-radix=x fortunes.dat
...
000660 00 00 59 0d 00 00 59 55 00 00 59 7d 00 00 59 b5
000670 00 00 59 f4 00 00 5a 35 00 00 5a 5e 00 00 5a 9c
000680 00 00 5a cb 00 00 5a f4 00 00 5b 1f 00 00 5b 3d
000690 00 00 5b 68 00 00 5b ab 00 00 5b f9 00 00 5c 49
0006a0 00 00 5c ae 00 00 5c eb 00 00 5d 34 00 00 5d 7a
0006b0 00 00 5d a3 00 00 5d f5 00 00 5e 3a 00 00 5e 67
0006c0 00 00 5e a8 00 00 5e ce 00 00 5e f7 00 00 5f 30
0006d0 00 00 5f 82 00 00 5f c4
0006d8
```

Peut-être que le développeur du programme *fortune* avait un ordinateur big-endian ou peut-être a-t-il été porté depuis quelque chose comme ça.

Ok, donc le tableau est big-endian, et, à en juger avec bon sens, la toute première phrase dans le fichier texte doit commencer à l'offset zéro. Donc la valeur zéro doit se trouver dans le tableau quelque part au tout début. Nous avons un couple d'élément à zéro au début. Mais le second est plus tentant: 43 se trouve juste après et 43 est un offset valide sur une phrase anglaise correcte dans le fichier texte.

Le dernier élément du tableau est 0x5FC4, et il n'y a pas de tel octet à cet offset dans le fichier texte. Donc le dernier élément du tableau pointe au delà de la fin du fichier. C'est probablement ainsi car la longueur de la phrase est calculée comme la différence entre l'offset de la phrase courante et l'offset de la phrase suivante. Ceci est plus rapide que de lire la chaîne à la recherche du caractère pourcent. Mais ceci ne fonctionne pas pour le dernier élément. Donc un élément *fictif* est aussi ajouté à la fin du tableau.

Donc les 6 première valeur entière 32-bit sont une sorte d'en-tête.

Oh, j'ai oublié de compter les phrases dans le fichier texte:

```
% cat fortunes | grep % | wc -l
432
```

Le nombre de phrases peut être présent dans dans l'index, mais peut-être pas. Dans le cas de fichiers d'index simples, le nombre d'éléments peut être facilement déduit de la taille du fichier d'index. Quoiqu'il en soit, il y a 432 phrases dans le fichier texte. Et nous voyons quelque chose de très familier dans le second élément (la valeur 431). J'ai vérifié dans d'autres fichiers (literature.dat et riddles.dat sur Ubuntu

Linux) et oui, le second élément 32-bit est bien le nombre de phrases moins 1. Pourquoi *moins 1* ? Peut-être que ceci n'est pas le nombre de phrases, mais plutôt le numéro de la dernière phrase (commençant à zéro)?

Et il y a d'autres éléments dans l'entête. Dans Mathematica, je charge chacun des trois fichiers disponible et je regarde leur en-tête:

```
In[14]:= input = BinaryReadList["c:/tmp1/fortunes.dat", "UnsignedInteger32",
            ByteOrdering → 1];

In[18]:= BaseForm[Take[input, {1, 6}], 16]
Out[18]//BaseForm=
         {2₁₆, 1af₁₆, bb₁₆, f₁₆, 0₁₆, 25000000₁₆}

In[19]:= input = BinaryReadList["c:/tmp1/literature.dat", "UnsignedInteger32",
            ByteOrdering → 1];

In[20]:= BaseForm[Take[input, {1, 6}], 16]
Out[20]//BaseForm=
         {2₁₆, 106₁₆, 983₁₆, 1a₁₆, 0₁₆, 25000000₁₆}

In[21]:= input = BinaryReadList["c:/tmp1/riddles.dat", "UnsignedInteger32", ByteOrdering → 1];

In[22]:= BaseForm[Take[input, {1, 6}], 16]
Out[22]//BaseForm=
         {2₁₆, 80₁₆, 7f2₁₆, 24₁₆, 0₁₆, 25000000₁₆}
```

Je n'ai aucune idée de la signification des autres valeurs, excepté la taille du fichier d'index. Certains champs sont les même pour tous les fichiers, d'autres non. D'après mon expérience, ils peuvent être:

- signature de fichier;
- version de fichier;
- checksum;
- des flags;
- peut-être même des identifiants de langages;
- timestamp du fichier texte, donc le programme *fortune* regénèrerait le fichier d'index isi l'utilisateur modifiait le fichier texte.

Par exemple, les fichiers Oracle .SYM (**??** on page ??) qui contiennent la table des symboles pour les fichiers DLL, contiennent aussi un timestamp correspondant au fichier DLL, afin d'être sûr qu'il est toujours valide.

D'un autre côté, les timestamps des fichiers textes et des fichiers d'index peuvent être désynchronisés après archivage/désarchivage/installation/déploiement/etc.

Mais ce ne sont pas des timestamps, d'après moi. La manière la plus compacte de représenter la date et l'heure est la valeur UNIX, qui est un nombre 32-bit. Nous ne voyons rien de tel ici. D'autres façons de les représenter sont encore moins compactes.

Donc, voici supposément comment fonctionne l'algorithme de *fortune* :

- prendre le nombre du second élément de la dernière phrase;
- générer un nombre aléatoire dans l'intervalle 0..number_of_last_phrase;
- trouver l'élément correspondant dans le tableau des offsets, prendre aussi l'offset suivant;
- écrire sur *stdout* tous les caractères depuis le fichier texte en commençant à l'offset jusqu'à l'offset suivant moins 2 (afin d'ignorer le caractère pourcent terminal et le caractère de la phrase suivante).

6.3.1 Hacking

Effectuons quelques essais afin de vérifier nos hypothèses. Je vais créer ce fichier texte dans le chemin et le nom */usr/share/games/fortunes/fortunes* :

```
Phrase one.
%
Phrase two.
%
```

Puis, ce fichier fortunes.dat. Je prend l'entête du fichier original fortunes.dat, j'ai mis à zéro changé le second champ (nombre de phrases) et j'ai laissé deux éléments dans le tableau: 0 et 0x1c, car la longueur totale du fichier texte *fortunes* est 28 (0x1c) octets:

```
% od -t x1 --address-radix=x fortunes.dat
000000 00 00 00 02 00 00 00 00 00 00 00 bb 00 00 00 0f
000010 00 00 00 00 25 00 00 00 00 00 00 00 00 00 00 1c
```

Maintenant, je le lance:

```
% /usr/games/fortune
fortune : no fortune found
```

Quelque chose ne va pas. Mettons le second champ à 1:

```
% od -t x1 --address-radix=x fortunes.dat
000000 00 00 00 02 00 00 00 01 00 00 00 bb 00 00 00 0f
000010 00 00 00 00 25 00 00 00 00 00 00 00 00 00 00 1c
```

Maintenant, ça fonctionne. Il affiche seulement la première phrase:

```
% /usr/games/fortune
Phrase one.
```

Hmmm. Laissons seulement un élément dans le tableau (0) sans le terminer:

```
% od -t x1 --address-radix=x fortunes.dat
000000 00 00 00 02 00 00 00 01 00 00 00 bb 00 00 00 0f
000010 00 00 00 00 25 00 00 00 00 00 00 00
00001c
```

Le programme Fortune montre toujours seulement la première phrase.

De cet essai nous apprenons que la signe pourcent dans le fichier texte est analysé et la taille n'est pas calculée comme je l'avais déduit avant, peut-être que même l'élément terminal du tableau n'est pas utilisé. Toutefois, il peut toujours l'être. Et peut-être qu'il l'a été dans le passé?

6.3.2 Les fichiers

Pour les besoins de la démonstration, je ne regarde toujours pas dans le code source de *fortune*. Si vous soulez essayer de comprendre la signification des autres valeurs dans l'entête du fichier d'index, vous pouvez essayer de le faire sans regarder dans le code source. Les fichiers que j'ai utilisé sous Ubuntu Linux 14.04 sont ici: http://beginners.re/examples/fortune/, les fichiers bricolés le sont aussi.

Oh, j'ai pris les fichiers de la version x64 d'Ubuntu, mais les éléments du tableau ont toujours une taille de 32 bit. C'est parce que les fichiers de texte de *fortune* ne vont sans doute jamais dépasser une taille de 4GiB[7]. Mais s'ils le devaient, tous les éléments devraient avoir une taille de 64 bit afin de pouvoir stocker un offset de dans un fichier texte plus gros que 4GiB.

Pour les lecteurs impatients, le code source de *fortune* est ici: https://launchpad.net/ubuntu/+source/fortune-mod/1:1.99.1-3.1ubuntu4.

[7] Gibibyte

6.4 Exercices

Essayez de rétro-ingénieurer tous les fichiers binaires de votre jeu favori, fichier des meilleurs scores inclus, ressources, etc.

Il y a aussi des fichiers binaires avec une structure connue: les fichiers utmp/wtmp, essayer de comprendre leur structure sans documentation.

L'entête EXIF dans les fichiers JPEG est documenté, mais vous pouvez essayer de comprendre sa structure sans aide, simplement en prenant des photos à différentes heures/dates, lieux, et essayer de trouver la date/heure et position GPS dans les données EXIF. Essayez de modifier la position GPS, uploadez le fichier JPEG dans Facebook et regardez, comment il va mettre votre photo sur la carte.

Essayez de patcher toutes les informations dans un fichier MP3 et voyez comment réagit votre lecteur de MP3 favori.

6.5 Pour aller plus loin

Pierre Capillon – Black-box cryptanalysis of home-made encryption algorithms: a practical case study.

Chapitre 7

Autres sujets

7.1 Modification de fichier exécutable

7.1.1 Chaînes de caractères

Les chaînes de caractères du langage C sont les plus faciles à modifier (sauf si elles sont chiffrées) avec un simple éditeur hexadécimal. La technique peut être mise en oeuvre même par ceux qui ne connaissent ni le langage machine, ni le format des fichiers exécutables. Il faut toutefois que la nouvelle chaîne de caractères ne soit pas plus longue que l'ancienne, sinon il existe un risque d'écraser une autre valeur ou du code exécutable.

C'est en utilisant cette méthode que de nombreux programmes ont été traduits à l'époque de MS-DOS, du moins dans les années 80 et 90 dans l'ex-URSS. Elle aboutissait parfois à la présence de quelques abréviations folkloriques dans la traduction, faute de place pour des chaînes plus longues.

En ce qui concerne Delphi, la taille de la chaîne de caractères doit elle aussi être ajustée.

7.1.2 code x86

Les tâches de modification courantes sont:

- Une des tâches la plus fréquente est de désactiver une instruction en l'écrasant avec des octets `0x90` (NOP).
- Les branchements conditionnels qui utilisent un code instruction tel que `74 xx` (`JZ`), peuvent être réécrits avec deux instructions NOP.

  Une autre technique consiste à désactiver un branchement conditionnel en écrasant le second octet avec la valeur 0 (*jump offset*).
- Une autre tâche courante consiste à faire en sorte qu'un branchement conditionnel soit effectué systématiquement. On y parvient en remplaçant le code instruction par `0xEB` qui correspond à l'instruction `JMP`.
- L'exécution d'une fonction peut être désactivée en remplaçant le premier octet par `RETN` (0xC3). Les fonctions dont la convention d'appel est `stdcall` (4.1.2 on page 544) font exception. Pour les modifier, il faut déterminer le nombre d'arguments (par exemple en trouvant une instruction `RETN` au sein de la fonction), puis en utilisant l'instruction `RETN` accompagnée d'un argument sur deux octets (0xC2).
- Il arrive qu'une fonction que l'on a désactivée doive retourner une valeur 0 ou 1. Certes on peut utiliser `MOV EAX, 0` ou `MOV EAX, 1`, mais cela occupe un peu trop d'espace.
  Une meilleure approche consiste à utiliser `XOR EAX, EAX` (2 octets `0x31 0xC0`) ou `XOR EAX, EAX / INC EAX` (3 octets `0x31 0xC0 0x40`).

Un logiciel peut être protégé contre les modifications. Le plus souvent la protection consiste à lire le code du programme (en mémoire) et à en calculer une valeur de contrôle. Cette technique nécessite que la protection lise le code avant de pouvoir agir. Elle peut donc être détectée en positionnant un point d'arrêt déclenché par la lecture de la mémoire contenant le code.

tracer possède l'option BPM pour ce faire.

La partie du fichier au format PE qui contient les informations de relocation (4.5.2 on page 570) ne doivent pas être modifiées par les patchs car le chargeur Windows risquerait d'écraser les modifications apportées. (Ces parties sont présentées sous forme grisées dans Hiew, par exemple: fig.1.21).

En dernier ressort, il est possible d'effectuer des modifications qui contournent les relocations, ou de modifier directement la table des relocations.

7.2 Statistiques sur le nombre d'arguments d'une fonction

J'ai toujours été intéressé par la moyenne du nombre d'arguments d'une fonction.

J'ai donc analysé un bon nombre de DLLs 32 bits de Windows7 (crypt32.dll, mfc71.dll, msvcr100.dll, shell32.dll, user32.dll, d3d11.dll, mshtml.dll, msxml6.dll, sqlncli11.dll, wininet.dll, mfc120.dll, msvbvm60.dll, ole32.dll, themeui.dll, wmp.dll) (parce qu'elles utilise la convention d'appel *stdcall* et qu'il est donc facile de retrouver les instructions `RETN X` en utilisant la commande *grep* sur leur code une fois celui-ci désasssemblé).

- no arguments: $\approx 29\%$
- 1 argument: $\approx 23\%$
- 2 arguments: $\approx 20\%$
- 3 arguments: $\approx 11\%$
- 4 arguments: $\approx 7\%$
- 5 arguments: $\approx 3\%$
- 6 arguments: $\approx 2\%$
- 7 arguments: $\approx 1\%$

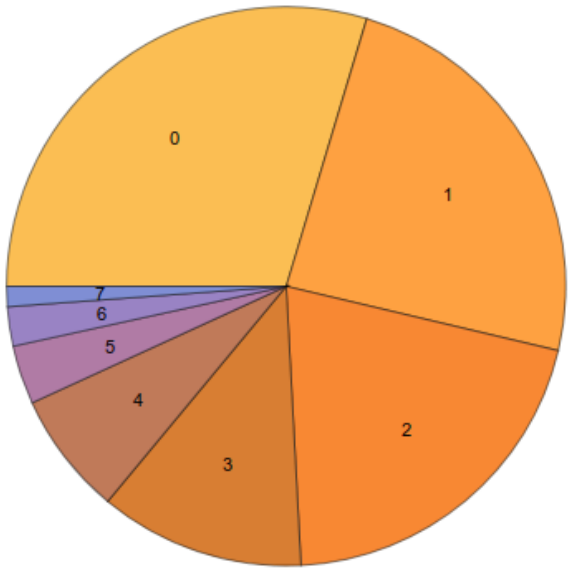


Fig. 7.1: Statistiques du nombre d'arguments moyen d'une fonction

Ces nombres dépendent beaucoup du style de programmation et peuvent s'avérer très différents pour d'autres produits logiciels.

7.3 Fonctions intrinsèques du compilateur

Les fonctions intrinsèques sont spécifiques à chaque compilateur. Ce ne sont pas des fonctions que vous pouvez retrouver dans une librairie. Le compilateur génère une séquence spécifique de code machine

lorsqu'il rencontre la fonction intrinsèque. Le plus souvent, il s'agit d'une pseudo fonction qui correspond à une instruction d'une CPU particulière.

Par exemple, il n'existe pas d'opérateur de décalage cyclique dans les langages C/C++. La plupart des CPUs supportent cependant des instructions de ce type. Pour faciliter la vie des programmeurs, le compilateur MSVC propose de telles pseudo fonctions *_rotl()* and *_rotr()*[1] qui sont directement traduites par le compilateur vers les instructions x86 ROL/ROR.

Les fonctions intrinsèques qui permettent de générer des instructions SSE sont un autre exemple.

La liste complète des fonctions intrinsèques exposées par le compilateur MSVC figurent dans le MSDN.

7.4 Anomalies des compilateurs

7.4.1 Oracle RDBMS 11.2 et Intel C++ 10.1

Le compilateur Intel C++ 10.1, qui fut utilisé pour la compilation de Oracle RDBMS 11.2 pour Linux86, émettait parfois deux instructions JZ successives, sans que la seconde instruction soit jamais référencée. Elle était donc inutile.

Listing 7.1: kdli.o from libserver11.a

```
.text :08114CF1                     loc_8114CF1 : ; CODE XREF : __PGOSF539_kdlimemSer+89A
.text :08114CF1                                   ; __PGOSF539_kdlimemSer+3994
.text :08114CF1 8B 45 08                mov     eax, [ebp+arg_0]
.text :08114CF4 0F B6 50 14             movzx   edx, byte ptr [eax+14h]
.text :08114CF8 F6 C2 01                test    dl, 1
.text :08114CFB 0F 85 17 08 00 00       jnz     loc_8115518
.text :08114D01 85 C9                   test    ecx, ecx
.text :08114D03 0F 84 8A 00 00 00       jz      loc_8114D93
.text :08114D09 0F 84 09 08 00 00       jz      loc_8115518
.text :08114D0F 8B 53 08                mov     edx, [ebx+8]
.text :08114D12 89 55 FC                mov     [ebp+var_4], edx
.text :08114D15 31 C0                   xor     eax, eax
.text :08114D17 89 45 F4                mov     [ebp+var_C], eax
.text :08114D1A 50                      push    eax
.text :08114D1B 52                      push    edx
.text :08114D1C E8 03 54 00 00          call    len2nbytes
.text :08114D21 83 C4 08                add     esp, 8
```

Listing 7.2: from the same code

```
.text :0811A2A5                     loc_811A2A5 : ; CODE XREF : kdliSerLengths+11C
.text :0811A2A5                                   ; kdliSerLengths+1C1
.text :0811A2A5 8B 7D 08                mov     edi, [ebp+arg_0]
.text :0811A2A8 8B 7F 10                mov     edi, [edi+10h]
.text :0811A2AB 0F B6 57 14             movzx   edx, byte ptr [edi+14h]
.text :0811A2AF F6 C2 01                test    dl, 1
.text :0811A2B2 75 3E                   jnz     short loc_811A2F2
.text :0811A2B4 83 E0 01                and     eax, 1
.text :0811A2B7 74 1F                   jz      short loc_811A2D8
.text :0811A2B9 74 37                   jz      short loc_811A2F2
.text :0811A2BB 6A 00                   push    0
.text :0811A2BD FF 71 08                push    dword ptr [ecx+8]
.text :0811A2C0 E8 5F FE FF FF          call    len2nbytes
```

Il s'agit probablement d'un bug du générateur de code du compilateur qui ne fut pas découvert durant les tests de celui-ci car le code produit fonctionnait conformément aux résultats attendus.

7.4.2 MSVC 6.0

Je viens juste de trouver celui-ci dans un vieux fragment de code :

[1] MSDN

```
                fabs
                fild    [esp+50h+var_34]
                fabs
                fxch    st(1) ; première instruction
                fxch    st(1) ; seconde instruction
                faddp   st(1), st
                fcomp   [esp+50h+var_3C]
                fnstsw  ax
                test    ah, 41h
                jz      short loc_100040B7
```

La première instruction FXCH interverti les valeurs de ST(0) et ST(1). La seconde effectue la même opération. Combinées, elles ne produisent donc aucun effet. Cet extrait provient d'un programme qui utilise la librairie MFC42.dll, il a donc du être compilé avec MSVC 6.0 ou 5.0 ou peut-être même MSVC 4.2 qui date des années 90.

Cette paire d'instruction ne produit aucun effet, ce qui expliquerait qu'elle n'ait pas été détectée lors des tests du compilateur MSVC. Ou bien j'ai loupé quelque chose ...

7.4.3 Résumé

Des anomalies constatées dans d'autres compilateurs figurent également dans ce livre: 1.22.2 on page 317, 3.7.3 on page 497, 3.15.7 on page 537, 1.20.7 on page 304, 1.14.4 on page 147, 1.22.5 on page 334.

Ces cas sont exposés dans ce livre afin de démontrer que ces compilateurs comportent leurs propres erreurs et qu'il convient de ne pas toujours se triturer le cerveau en tentant de comprendre pourquoi le compilateur a généré un code aussi étrange.

7.5 Itanium

Bien qu'elle n'ai pas réussi à percer, l'architecture Intel Itanium (IA64) est très intéressante.

Là ou les CPUs OOE réarrangent les instructions afin de les exécuter en parallèle, l'architecture EPIC[2] a constitué une tentative pour déléguer cette décision au compilateur.

Les compilateurs en question étaient évidemment particulièrement complexes.

Voici un exemple de code pour l'architecture IA64 qui implémente un algorithme de chiffrage simple du noyau Linux:

Listing 7.3: Linux kernel 3.2.0.4

```
#define TEA_ROUNDS              32
#define TEA_DELTA               0x9e3779b9

static void tea_encrypt(struct crypto_tfm *tfm, u8 *dst, const u8 *src)
{
        u32 y, z, n, sum = 0;
        u32 k0, k1, k2, k3;
        struct tea_ctx *ctx = crypto_tfm_ctx(tfm);
        const __le32 *in = (const __le32 *)src;
        __le32 *out = (__le32 *)dst;

        y = le32_to_cpu(in[0]);
        z = le32_to_cpu(in[1]);

        k0 = ctx->KEY[0];
        k1 = ctx->KEY[1];
        k2 = ctx->KEY[2];
        k3 = ctx->KEY[3];

        n = TEA_ROUNDS;

        while (n-- > 0) {
```

[2]Explicitly Parallel Instruction Computing

```
                sum += TEA_DELTA;
                y += ((z << 4) + k0) ^ (z + sum) ^ ((z >> 5) + k1);
                z += ((y << 4) + k2) ^ (y + sum) ^ ((y >> 5) + k3);
        }

        out[0] = cpu_to_le32(y);
        out[1] = cpu_to_le32(z);
}
```

Et voici maintenant le résultat de la compilation:

Listing 7.4: Linux Kernel 3.2.0.4 for Itanium 2 (McKinley)

```
0090|                    tea_encrypt :
0090|08 80 80 41 00 21      adds r16 = 96, r32              // ptr to ctx->KEY[2]
0096|80 C0 82 00 42 00      adds r8 = 88, r32               // ptr to ctx->KEY[0]
009C|00 00 04 00            nop.i 0
00A0|09 18 70 41 00 21      adds r3 = 92, r32               // ptr to ctx->KEY[1]
00A6|F0 20 88 20 28 00      ld4 r15 = [r34], 4              // load z
00AC|44 06 01 84            adds r32 = 100, r32;;           // ptr to ctx->KEY[3]
00B0|08 98 00 20 10 10      ld4 r19 = [r16]                 // r19=k2
00B6|00 01 00 00 42 40      mov r16 = r0                    // r0 always contain zero
00BC|00 08 CA 00            mov.i r2 = ar.lc                // save lc register
00C0|05 70 00 44 10 10
     9E FF FF FF 7F 20      ld4 r14 = [r34]                 // load y
00CC|92 F3 CE 6B            movl r17 = 0xFFFFFFFF9E3779B9;; // TEA_DELTA
00D0|08 00 00 00 01 00      nop.m 0
00D6|50 01 20 20 20 00      ld4 r21 = [r8]                  // r21=k0
00DC|F0 09 2A 00            mov.i ar.lc = 31                // TEA_ROUNDS is 32
00E0|0A A0 00 06 10 10      ld4 r20 = [r3];;                // r20=k1
00E6|20 01 80 20 20 00      ld4 r18 = [r32]                 // r18=k3
00EC|00 00 04 00            nop.i 0
00F0|
00F0|                                loc_F0 :
00F0|09 80 40 22 00 20      add r16 = r16, r17              // r16=sum, r17=TEA_DELTA
00F6|D0 71 54 26 40 80      shladd r29 = r14, 4, r21        // r14=y, r21=k0
00FC|A3 70 68 52            extr.u r28 = r14, 5, 27;;
0100|03 F0 40 1C 00 20      add r30 = r16, r14
0106|B0 E1 50 00 40 40      add r27 = r28, r20;;            // r20=k1
010C|D3 F1 3C 80            xor r26 = r29, r30;;
0110|0B C8 6C 34 0F 20      xor r25 = r27, r26;;
0116|F0 78 64 00 40 00      add r15 = r15, r25              // r15=z
011C|00 00 04 00            nop.i 0;;
0120|00 00 00 00 01 00      nop.m 0
0126|80 51 3C 34 29 60      extr.u r24 = r15, 5, 27
012C|F1 98 4C 80            shladd r11 = r15, 4, r19        // r19=k2
0130|0B B8 3C 20 00 20      add r23 = r15, r16;;
0136|A0 C0 48 00 40 00      add r10 = r24, r18              // r18=k3
013C|00 00 04 00            nop.i 0;;
0140|0B 48 28 16 0F 20      xor r9 = r10, r11;;
0146|60 B9 24 1E 40 00      xor r22 = r23, r9
014C|00 00 04 00            nop.i 0;;
0150|11 00 00 00 01 00      nop.m 0
0156|E0 70 58 00 40 A0      add r14 = r14, r22
015C|A0 FF FF 48            br.cloop.sptk.few loc_F0;;
0160|09 20 3C 42 90 15      st4 [r33] = r15, 4              // store z
0166|00 00 00 02 00 00      nop.m 0
016C|20 08 AA 00            mov.i ar.lc = r2;;              // restore lc register
0170|11 00 38 42 90 11      st4 [r33] = r14                 // store y
0176|00 00 00 02 00 80      nop.i 0
017C|08 00 84 00            br.ret.sptk.many b0;;
```

Nous constatons tout d'abord que toutes les instructions IA64 sont regroupées par 3.

Chaque groupe représente 16 octets (128 bits) et se décompose en une catégorie de code sur 5 bits puis 3 instructions de 41 bits chacune.

Dans IDA les groupes apparaissent sous la forme 6+6+4 octets —le motif est facilement reconnaissable.

En règle générale les trois instructions d'un groupe s'exécutent en parallèle, sauf si l'une d'elles est associée à un « stop bit ».

Il est probable que les ingénieurs d'Intel et de HP ont collecté des statistiques qui leur ont permis d'identifier les motifs les plus fréquents. Ils auraient alors décidé d'introduire une notion de type de groupe (AKA « templates »). Le type du groupe définit la catégorie des instructions qu'il contient. Ces catégories sont au nombre de 12.

Par exemple, un groupe de type 0 représente `MII`. Ceci signifie que la première instruction est une lecture ou écriture en mémoire (M), la seconde et la troisième sont des manipulations d'entiers (I).

Un autre exemple est le groupe de type 0x1d: `MFB`. La première instruction est la aussi de type mémoire (M), la seconde manipule un nombre flottant (F instruction FPU), et le dernier est un branchement (B).

Lorsque le compilateur ne parvient pas à sélectionner un instruction à inclure dans le groupe en cours de construction, il utilise une instruction de type NOP. Il existe donc des instructions `nop.i` pour remplacer ce qui devrait être une manipulation d'entier. De même un `nop.m` est utilisé pour combler un trou là où une instruction de type mémoire devrait se trouver.

Lorsque le programme est directement dédigé en assembleur, les instructions NOPs sont ajoutées de manière automatique.

Et ce n'est pas tout. Les groupes font eux-mêmes l'objet de regroupements.

Chaque instruction peut être marquée avec un « stop bit ». Le processeur exécute en parallèle, toutes les instructions jusqu'à ce qu'il rencontre un « stop bit ».

En pratique, les processeurs Itanium 2 peuvent exécuter jusqu'à deux groupes simultanément, soit un total de 6 instructions en parallèle.

Il faut évidemment que les instructions exécutées en parallèle, n'aient pas d'effet de bord entre elles. Dans le cas contraire, le résultat n'est pas défini. Le compilateur doit respecter cette contrainte ainsi que le nombre maximum de groupes simultanés du processeur cible en placant les « stop bit » au bon endroit.

En langage assembleur, les bits d'arrêt sont identifiés par deux point-virgule situé après l'instruction.

Ainsi dans notre exemple les instructions [90-ac] peuvent être exécutées simultanément. Le prochain groupe est [b0-cc].

Nous observons également un bit d'arrêt en 10c. L'instruction suivante comporte elle aussi un bit d'arrêt.

Ces deux instructions doivent donc être exécutées isolément des autres, (comme dans une architecture CISC).

En effet, l'instruction en 110 utilise le résultat produit par l'instruction précédente (valeur du registre r26). Les deux instructions ne peuvent s'exécuter simultanément.

Il semble que le compilateur n'ai pas trouvé de meilleure manière de paralléliser les instructions, ou en d'autres termes de plus charger la CPU. Les bits d'arrêt sont donc en trop grand nombre.

La rédaction manuelle de code en assembleur est une tâche pénible. Le programmeur doit effectuer lui-même les regroupements d'instructions.

Bien sûr, il peut ajouter un bit d'arrêt à chaque instruction mais cela dégrade les performances telles qu'elles ont été pensée pour l'Itanium.

Les codes source du noyau Linux contiennent un exemple intéressant d'un code assembleur produit manuellement pour IA64 :

http://go.yurichev.com/17322.

On trouvera une introduction à l'assembleur Itanium dans : [Mike Burrell, *Writing Efficient Itanium 2 Assembly Code* (2010)][3], [papasutra of haquebright, *WRITING SHELLCODE FOR IA-64* (2001)][4].

Deux autres caractéristiques très intéressantes d'Itanium feature sont la *speculative execution* et le bit NaT (« not a thing ») qui ressemble un peu aux nombres NaN :
MSDN.

7.6 Modèle de mémoire du 8086

Lorsque l'on a à faire avec des programmes 16-bit pour MS-DOS ou Win16 (**??** on page ?? ou **??** on page ??), nous voyons que les pointeurs consistent en deux valeurs 16-bit. Que signifient-elles? Eh oui, c'est encore un artéfact étrange de MS-DOS et du 8086.

[3] Aussi disponible en http://yurichev.com/mirrors/RE/itanium.pdf
[4] Aussi disponible en http://phrack.org/issues/57/5.html

Le 8086/8088 était un CPU 16-bit, mais était capable d'accèder à des adresses mémoire sur 20-bit (il était donc capable d'accèder 1MB de mémoire externe).

L'espace de la mémoire externe était divisé entre la RAM (max 640KB), la ROM, la fenêtre pour la mémoire vidéo, les cartes EMS, etc.

Rappelons que le 8086/8088 était en fait un descendant du CPU 8-bit 8080.

Le 8080 avait un espace mémoire de 16-bit, i.e., il pouvait seulement adresser 64KB.

Et probablement pour une raison de portage de vieux logiciels[5], le 8086 peut supporter plusieurs fenêtres de 64KB simultanément, situées dans l'espace d'adresse de 1MB.

C'est une sorte de virualisation de niveau jouet.

Tous les registres 8086 sont 16-bit, donc pour adresser plus, des registres spéciaux de segment (CS, DS, ES, SS) ont été ajoutés.

Chaque pointeur de 20-bit est calculé en utilisant les valeurs d'une paire de registres, de segment et d'adresse (p.e. DS:BX) comme suit:

$$real\_address = (segment\_register \ll 4) + address\_register$$

Par exemple, la fenêtre de RAM graphique (EGA[6], VGA[7]) sur les vieux compatibles IBM-PC a une taille de 64KB.

Pour y accèder, une valeur de 0xA000 doit être stockée dans l'un des registres de segments, p.e. dans DS.

Ainsi DS:0 adresse le premier octet de la RAM vidéo et DS:0xFFFF — le dernier octet de RAM.

L'adresse réelle sur le bus d'adresse de 20-bit, toutefois, sera dans l'intervalle 0xA0000 à 0xAFFFF.

Le programme peut contenir des adresses codées en dur comme 0x1234, mais l'OS peut avoir besoin de le charger à une adresse arbitraire, donc il recalcule les valeurs du registre de segment de façon à ce que le programme n'ait pas à se soucier de l'endroit oũ il est placé dans la RAM.

Donc, tout pointeur dans le vieil environnement MS-DOS consistait en fait en un segment d'adresse et une adresse dans ce segment, i.e., deux valeurs 16-bit. 20-bit étaient suffisants pour cela, cependant nous devions recalculer les adresses très souvent: passer plus d'information par la pile semblant un meilleur rapport espace/facilité.

À propos, à cause de tout cela, il n'était pas possible d'allouer un bloc de mémoire plus large que 64KB.

Les registres de segment furent réutilisés sur les 80286 comme sélecteurs, servant a une fonction différente.

Lorsque les CPU 80386 et les ordinateurs avec plus de RAM ont été introduits, MS-DOS était encore populaire, donc des extensions pour DOS ont émergés: ils étaient en fait une étape vers un OS « sérieux », basculant le CPU en mode protégé et fournissant des APIs mémoire bien meilleures pour les programmes qui devaient toujours fonctionner sous MS-DOS.

Des examples très populaires incluent DOS/4GW (le jeux vidéo DOOM a été compilé pour lui), Phar Lap, PMODE.

À propos, la même manière d'adresser la mémoire était utilisée dans la série 16-bit de Windows 3.x, avant Win32.

7.7 Réordonnancement des blocs élémentaires

7.7.1 Optimisation guidée par profil

Cette méthode d'optimisation déplace certains basic blocks vers d'autres sections du fichier binaire exécutable.

Il est évident que certaines parties d'une fonction sont exécutées plus fréquemment que d'autres (ex: le corps d'une boucle) et d'autres moins souvent (ex: gestionnaire d'erreur, gestionnaires d'exception).

[5] Je ne suis pas sûr à 100% de ceci
[6] Enhanced Graphics Adapter
[7] Video Graphics Array

Le compilateur ajoute dans le code exécutable des instructions d'instrumentation. Le développeur exécute ensuite un nombre important de tests, ce qui permet de collecter des statistiques.

A l'aider de ces dernières, le compilateur prépare le fichier exécutable final en déplacant les fragments de code les moins exécutés vers une autre section.

Tous les fragments de code les plus souvent exécutés sont ainsi regroupées, ce qui constitue un facteur important pour la rapidité d'exécution et l'utilisation du cache.

Voici un exemple de code Oracle RDBMS produit par le compilateur Intel C++:

Listing 7.5: orageneric11.dll (win32)

```
                public _skgfsync
_skgfsync       proc near

; address 0x6030D86A

                db      66h
                nop
                push    ebp
                mov     ebp, esp
                mov     edx, [ebp+0Ch]
                test    edx, edx
                jz      short loc_6030D884
                mov     eax, [edx+30h]
                test    eax, 400h
                jnz     __VInfreq__skgfsync  ; write to log
continue :
                mov     eax, [ebp+8]
                mov     edx, [ebp+10h]
                mov     dword ptr [eax], 0
                lea     eax, [edx+0Fh]
                and     eax, 0FFFFFFFCh
                mov     ecx, [eax]
                cmp     ecx, 45726963h
                jnz     error                ; exit with error
                mov     esp, ebp
                pop     ebp
                retn
_skgfsync       endp

...

; address 0x60B953F0

__VInfreq__skgfsync :
                mov     eax, [edx]
                test    eax, eax
                jz      continue
                mov     ecx, [ebp+10h]
                push    ecx
                mov     ecx, [ebp+8]
                push    edx
                push    ecx
                push    offset ... ; "skgfsync(se=0x%x, ctx=0x%x, iov=0x%x)\n"
                push    dword ptr [edx+4]
                call    dword ptr [eax] ; write to log
                add     esp, 14h
                jmp     continue

error :
                mov     edx, [ebp+8]
                mov     dword ptr [edx], 69AAh ; 27050 "function called with invalid FIB/IOV ⇙
    ↳ structure"
                mov     eax, [eax]
                mov     [edx+4], eax
                mov     dword ptr [edx+8], 0FA4h ; 4004
                mov     esp, ebp
                pop     ebp
                retn
; END OF FUNCTION CHUNK FOR _skgfsync
```

La distance entre ces deux fragments de code avoisine les 9 Mo.

Tous les fragments de code rarement exécutés sont regroupés à la fin de la section de code de la DLL.

La partie de la fonction qui a été déplacée était marquée par le compilateur Intel C++ avec le préfixe `VInfreq`.

Nous voyons donc qu'une partie de la fonction qui écrit dans un fichier journal (probablement à la suite d'une erreur ou d'un avertissement) n'a sans doute pas été exécuté très souvent durant les tests effectués par les développeurs Oracle lors de la collecte des statistiques. Il n'est même pas dit qu'elle ait jamais été exécutée.

Le bloc élémentaire qui écrit dans le journal s'achève par un retour à la partie « hot » de la fonction.

Un autre bloc élémentaire « infrequent » est celui qui retourne le code erreur 27050.

Pour ce qui est des fichiers Linux au format ELF, le compilateur Intel C++ déplace tous les fragments de code rarement exécutés vers une section séparée nommée `text.unlikely`. Les fragments les plus souvent exécutés sont quant à eux regroupés dans la section `text.hot`.

Cette information peut aider le rétro ingénieur à distinguer la partie principale d'une fonction des parties qui assurent la gestion d'erreurs.

Chapitre 8

Livres/blogs qui valent le détour

8.1 Livres et autres matériels

8.1.1 Rétro-ingénierie

- Eldad Eilam, *Reversing: Secrets of Reverse Engineering*, (2005)
- Bruce Dang, Alexandre Gazet, Elias Bachaalany, Sebastien Josse, *Practical Reverse Engineering: x86, x64, ARM, Windows Kernel, Reversing Tools, and Obfuscation*, (2014)
- Michael Sikorski, Andrew Honig, *Practical Malware Analysis: The Hands-On Guide to Dissecting Malicious Software*, (2012)
- Chris Eagle, *IDA Pro Book*, (2011)

Également, les livres de Kris Kaspersky.

8.1.2 Windows

- Mark Russinovich, *Microsoft Windows Internals*

Blogs:

- Microsoft: Raymond Chen
- nynaeve.net

8.1.3 C/C++

- Brian W. Kernighan, Dennis M. Ritchie, *The C Programming Language*, 2ed, (1988)
- *ISO/IEC 9899:TC3 (C C99 standard)*, (2007)[1]
- Bjarne Stroustrup, *The C++ Programming Language, 4th Edition*, (2013)
- C++11 standard[2]
- Agner Fog, *Optimizing software in C++* (2015)[3]
- Marshall Cline, *C++ FAQ*[4]
- Dennis Yurichev, *C/C++ programming language notes*[5]
- JPL Institutional Coding Standard for the C Programming Language[6]

[1] Aussi disponible en http://go.yurichev.com/17274
[2] Aussi disponible en http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2013/n3690.pdf.
[3] Aussi disponible en http://agner.org/optimize/optimizing_cpp.pdf.
[4] Aussi disponible en http://go.yurichev.com/17291
[5] Aussi disponible en http://yurichev.com/C-book.html
[6] Aussi disponible en https://yurichev.com/mirrors/C/JPL_Coding_Standard_C.pdf

8.1.4 Architecture x86 / x86-64

- Manuels Intel[7]
- Manuels AMD[8]
- Agner Fog, *The microarchitecture of Intel, AMD and VIA CPUs*, (2016)[9]
- Agner Fog, *Calling conventions* (2015)[10]
- [*Intel® 64 and IA-32 Architectures Optimization Reference Manual*, (2014)]
- [*Software Optimization Guide for AMD Family 16h Processors*, (2013)]

Quelque peu vieux, mais toujours intéressant à lire :

Michael Abrash, *Graphics Programming Black Book*, 1997[11] (il est connu pour son travail sur l'optimisation bas niveau pour des projets tels que Windows NT 3.1 et id Quake).

8.1.5 ARM

- Manuels ARM[12]
- *ARM(R) Architecture Reference Manual, ARMv7-A and ARMv7-R edition*, (2012)
- [*ARM Architecture Reference Manual, ARMv8, for ARMv8-A architecture profile*, (2013)][13]
- Advanced RISC Machines Ltd, *The ARM Cookbook*, (1994)[14]

8.1.6 Langage d'assemblage

Richard Blum — Professional Assembly Language.

8.1.7 Java

[Tim Lindholm, Frank Yellin, Gilad Bracha, Alex Buckley, *The Java(R) Virtual Machine Specification / Java SE 7 Edition*] [15].

8.1.8 UNIX

Eric S. Raymond, *The Art of UNIX Programming*, (2003)

8.1.9 Programmation en général

- Brian W. Kernighan, Rob Pike, *Practice of Programming*, (1999)
- Henry S. Warren, *Hacker's Delight*, (2002)Certaines personnes disent ques les trucs et astuces de ce livre ne sont plus pertinents aujourd'hui, car ils n'étaient valables que pour les CPUs RISC, où les instructions de branchement sont coûteuses. Néanmoins, ils peuvent énormément aider à comprendre l'algèbre booléenne et toutes les mathématiques associées.
- (Pour les passionnés avec des connaissances en informatique et mathématiques) Donald E. Knuth, *The Art of Computer Programming*.

[7] Aussi disponible en http://www.intel.com/content/www/us/en/processors/architectures-software-developer-manuals.html
[8] Aussi disponible en http://developer.amd.com/resources/developer-guides-manuals/
[9] Aussi disponible en http://agner.org/optimize/microarchitecture.pdf
[10] Aussi disponible en http://www.agner.org/optimize/calling_conventions.pdf
[11] Aussi disponible en https://github.com/jagregory/abrash-black-book
[12] Aussi disponible en http://infocenter.arm.com/help/index.jsp?topic=/com.arm.doc.subset.architecture.reference/index.html
[13] Aussi disponible en http://yurichev.com/mirrors/ARMv8-A_Architecture_Reference_Manual_(Issue_A.a).pdf
[14] Aussi disponible en http://go.yurichev.com/17273
[15] Aussi disponible en https://docs.oracle.com/javase/specs/jvms/se7/jvms7.pdf; http://docs.oracle.com/javase/specs/jvms/se7/html/

8.1.10 Cryptographie

- Bruce Schneier, *Applied Cryptography*, (John Wiley & Sons, 1994)
- (Free) lvh, *Crypto 101*[16]
- (Free) Dan Boneh, Victor Shoup, *A Graduate Course in Applied Cryptography*[17].

[16] Aussi disponible en https://www.crypto101.io/
[17] Aussi disponible en https://crypto.stanford.edu/~dabo/cryptobook/

Chapitre 9

Communautés

Il existe deux excellents subreddits liés à la RE[1] (rétro-ingénierie) sur reddit.com : reddit.com/r/ReverseEngineering et reddit.com/r/remath (en lien avec la liaison de la RE et des mathématiques).

Il y a également une section sur l'RE sur le site Stack Exchange :

reverseengineering.stackexchange.com.

Sur IRC, il y a un channel dédié au ##ŕeverseśur FreeNode[2].

[1] Reverse Engineering
[2] freenode.net

Epilogue

9.1 Des questions ?

Pour toute question, n'hésitez pas à envoyer un mail à l'auteur : <dennis@yurichev.com>. Vous avez une suggestion ou une proposition de contenu supplémentaire pour le livre ? N'hésitez pas à envoyer toute correction (grammaire incluse), etc...

L'auteur travaille beaucoup sur cette oeuvre, c'est pourquoi les numéros de pages et les numéros de parties peuvent rapidement changer. S'il vous plaît, ne vous fiez pas à ces derniers si vous m'envoyez un email. Il existe une méthode plus simple : faites une capture d'écran de la page, puis dans un éditeur graphique, soulignez l'endroit où il y a une erreur et envoyez-moi l'image. Je la corrigerai plus rapidement. Et si vous êtes familier avec git et LaTeX vous pouvez corriger l'erreur directement dans le code source :

GitHub.

N'hésitez surtout pas à m'envoyer la moindre erreur que vous pourriez trouver, aussi petite qu'elle soit, même si vous n'êtes pas certain que ce soit une erreur. J'écris pour les débutants après tout, il est donc crucial pour mon travail d'avoir les retours de débutants.

Acronymes utilisés

OS Système d'exploitation (Operating System) xiii

POO Programmation orientée objet 575

LP Langage de programmation x

PRNG Nombre généré pseudo-aléatoirement 378

ROM Mémoire morte 81

UAL Unité arithmétique et logique 27

LF Line feed (10 ou '\n' en C/C++) 530

CR Carriage return (13 ou '\r' en C/C++) 530

LIFO Dernier entré, premier sorti 30

MSB Bit le plus significatif 319

LSB Bit le moins significatif

NSA National Security Agency (Agence Nationale de la Sécurité) 468

SICP Structure and Interpretation of Computer Programs xiii

ABI Application Binary Interface 16

RA Adresse de retour 22

PE Portable Executable 5

SP stack pointer. SP/ESP/RSP dans x86/x64. SP dans ARM. 19

DLL Dynamic-Link Library 567

PC Program Counter. IP/EIP/RIP dans x86/64. PC dans ARM. 20

LR Link Register 6

IDA Désassembleur interactif et débogueur développé par Hex-Rays 5

IAT Import Address Table 568

INT Import Name Table 568

RVA Relative Virtual Address 568

VA Virtual Address 568

OEP Original Entry Point 557

MSVC Microsoft Visual C++

ASLR Address Space Layout Randomization 568

MFC Microsoft Foundation Classes 571

TLS Thread Local Storage 285

AKA Aussi connu sous le nom de 30

CRT C Runtime library 10

CPU Central Processing Unit xiii

FPU Floating-Point Unit v

CISC Complex Instruction Set Computing 20

RISC Reduced Instruction Set Computing 2

GUI Graphical User Interface 564

BSS Block Started by Symbol 25

SIMD Single Instruction, Multiple Data 195

BSOD Blue Screen of Death 558

DBMS Database Management Systems 455

ISA Instruction Set Architecture 2

HPC High-Performance Computing 522

SEH Structured Exception Handling 37

ELF Format de fichier exécutable couramment utilisé sur les systèmes *NIX, Linux inclus 79

TIB Thread Information Block 285

PIC Position Independent Code 559

NOP No Operation 6

BEQ (PowerPC, ARM) Branch if Equal 95

BNE (PowerPC, ARM) Branch if Not Equal 210

XOR eXclusive OR (OU exclusif) 660

MCU Microcontroller Unit 500

Glossaire

tas Généralement c'est un gros bout de mémoire fournit par l'OS et utilisé par les applications pour le diviser comme elles le souhaitent. malloc()/free() fonctionnent en utilisant le tas . 31, 350, 566, 567

nombre réel nombre qui peut contenir un point. ceci est *float* et *double* en C/C++ . 218

décrémenter Décrémenter de 1 . 19, 184, 203, 443

incrémenter Incrémenter de 1 . 16, 20, 184, 188, 203, 209, 328, 331, 443

nombre usuel, mais pas un réel. peut être utilisé pour passer des variables de type booléen et des énumérations . 233

produit Résultat d'une multiplication . 98, 225, 228, 409, 434, 458

moyenne arithmétique la somme de toutes les valeurs, divisé par leur nombre . 524

pointeur de pile Un registre qui pointe dans la pile. 10, 11, 20, 31, 35, 42, 54, 55, 73, 100, 544–547, 655

tail call C'est lorsque le compilateur (ou l'interpréteur) transforme la récursion (ce qui est possible: *tail recursion*) en une itération pour l'efficacité: wikipedia . 485

quotient Résultat de la division . 218, 220, 223, 224, 228, 433, 501, 525

anti-pattern En général considéré comme une mauvaise pratique . 33, 76, 470

atomic operation « $\alpha\tau o\mu o\varsigma$ » signifie « indivisible » en grec, donc il est garantie qu'une opération atomique ne sera pas interrompue par d'autres threads . 599

basic block . 645

callee Une fonction appelée par une autre . 33, 46, 66, 86, 97, 100, 102, 422, 470, 544–547, 549, 550

caller Une fonction en appelant une autre . 6, 8, 10, 30, 46, 86, 97, 98, 101, 108, 155, 422, 473, 544, 546, 547, 550

endianness Ordre des octets: 2.8 on page 468. 22, 78

GiB Gibioctet: 2^{30} or 1024 mebioctets ou 1073741824 octets . 16

jump offset une partie de l'opcode de l'instruction JMP ou Jcc, qui doit être ajoutée à l'adresse de l'instruction suivante, et c'est ainsi que le nouveau PC est calculé. Peut-être négatif . 94, 133

leaf function Une fonction qui n'appelle pas d'autre fonction . 29, 33

link register (RISC) Un registre où l'adresse de retour est en général stockée. Ceci permet d'appeler une fonction leaf sans utiliser la pile, i.e, plus rapidemment . 32

loop unwinding C'est lorsqu'un compilateur, au lieu de générer du code pour une boucle de n itérations, génère juste n copies du corps de la boucle, afin de supprimer les instructions pour la gestion de la boucle . 186

NaN pas un nombre: un cas particulier pour les nombres à virgule flottante, indiquant généralement une erreur . 236, 258, 644

PDB (Win32) Fichier contenant des informations de débogage, en général seulement les noms des fonctions, mais aussi parfois les arguments des fonctions et le nom des variables locales . 569

register allocator La partie du compilateur qui assigne des registes du CPU aux variables locales . 202, 308, 423

reverse engineering action d'examiner et de comprendre comment quelque chose fonctionne, parfois dans le but de le reproduire . iv

security cookie Une valeur aléatoire, différente à chaque exécution. Vous pouvez en lire plus à ce propos ici : 1.20.3 on page 284. 589

stack frame Une partie de la pile qui contient des informations spécifiques à la fonction courante: variables locales, arguments de la fonction, RA, etc. . 68, 98, 481, 589

stdout standard output. 22, 36, 155

thunk function Minuscule fonction qui a un seul rôle: appeler une autre fonction . 23, 395

tracer Mon propre outil de debugging. Vous pouvez en lire plus à son propos ici : 5.2.1 on page 601. 189–191, 585, 594, 639

Windows NT Windows NT, 2000, XP, Vista, 7, 8, 10. 294, 420, 557, 568, 598

word . Dans les ordinateurs plus vieux que les PCs, la taille de la mémoire était souvent mesurée en mots plutôt qu'en octet . 450–453, 459

xoring souvent utilisé en angalis, qui signifie appliquer l'opération XOR[3] . 589

[3] eXclusive OR (OU exclusif)

Index