52 65 76 65 72 73 65
45 6e 67 69 6e 65 65 72 69 6e 67
66 6f 72 20 42 65 67 69 6e 6e 65 72 73

44 65 6e 6e 69 73 20 59 75 72 69 63 68 65 76

リバースエンジニアリング入門

Dennis Yurichev
<dennis@yurichev.com>

Text version (2018年7月9日).

この本の最新版 (そしてロシア版) は以下で見ることができます: beginners.re

カバーはAndy Nechaevskyが制作しました: facebook

翻訳者求む！

この作品を英語とロシア語以外の言語に翻訳するのを手伝ってください。どのように翻訳されたテキストを私に送っても（どれほど短くても）、私はLaTeXのソースコードに入れます。

Read here.

いくつかは既にあります。ドイツ語, フランス語, 少しですが イタリア語, ポルトガル語 と ポーランド語

スピードは重要ではありません。なぜなら、これはオープンソースプロジェクトなのですから。あなたの名前はプロジェクト寄稿者として言及されます。韓国語、中国語、ペルシャ語は出版社によって予約されています。英語とロシア語のバージョン私は自分でやっていますが、私の英語はまだひどいので、文法などに関するメモにはとても感謝しています。私のロシア語にも欠陥があるので、ロシア語のテキストについての注釈にも感謝しています！

だから私に連絡するのをためらうことはありません: `dennis@yurichev.com`

簡略版

目次

はじめに

「reverse engineering」にはよく知られた意味がいくつかあります。

1）ソフトウェアのリバースエンジニアリング、コンパイルされたプログラムの研究

2）3D構造のスキャンと、それらを複製するために必要なその後のデジタル操作

3）DBMS[1] 構造の再構成

本書は最初の意味についての本です。

前提条件

Cプログラミング言語（PL[2]）の基礎知識。推奨図書: 7.1.3 on page 163

練習問題やタスク

...`http://challenges.re` にあります。

著者について

Dennis Yurichevは経験豊富なリバースエンジニアでありまたプログラマです。彼にはメール **dennis@yurichev.com**.

賛辞リバースエンジニアリング入門

- 「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, GNU創設者、自由なソフトウェアの活動家
- 「It's very well done .. and for free .. amazing.」[3] Daniel Bilar, Siege Technologies, LLC.
- 「... excellent and free」[4] Pete Finnigan, セキュリティグル Oracle RDBMS.
- 「... [the] book is interesting, great job!」 Michael Sikorski, 以下の著作の著者です *Practical Malware Analysis: The Hands-On Guide to Dissecting Malicious Software*.
- 「... my compliments for the very nice tutorial!」Herbert Bos, 教授 Vrije Universiteit Amsterdam, 共著者 *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, SAP Netweaver & Security . スペシャリスト

---

[1] Database Management Systems

[2] Japanese text placeholder

[3] twitter.com/daniel_bilar/status/436578617221742593

[4] twitter.com/petefinnigan/status/400551705797869568

- 「... [a] reasonable intro to some of the techniques.」[5] Mike Stay, 教授 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.」[6] , セルゲイブラウス Research Assistant Professor Dartmouth College コンピュータサイエンス学部
- 「Dennis @Yurichev has published an impressive (and free!) book on reverse engineering」[7] Tanel Poder, オラクルRDBMSパフォーマンスチューニングエキスパート.
- 「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.

謝辞

忍耐強く質問に答えてくれた方々：Andrey 「herm1t」 Baranovich, Slava 「Avid」 Kazakov, SkullC0DEr

ミスや不正確な記述を指摘してくれた方々：Stanislav 「Beaver」 Bobrytskyy, Alexander Lysenko, Alexander 「Solar Designer」 Peslyak, Federico Ramondino, Mark Wilson, Xenia Galinskaya, Razikhova Meiramgul Kayratovna, Anatoly Prokofiev, Kostya Begunets, Valentin "netch" Nechayev, Aleksandr Plakhov, Artem Metla, Shell Rocket, Zhu Ruijin, Changmin Heo, Vitor Vidal, Stijn Crevits, Jean-Gregoire Foulon[8], Ben L., Etienne Khan, Norbert Szetei[9], Marc Remy, Michael Hansen, Derk Barten, The Renaissance[10], Hugo Chan.

他に手助けしてくれた方々：Andrew Zubinski, Arnaud Patard (rtp on #debian-arm IRC), noshadow on #gcc IRC, Aliaksandr Autayeu, Mohsen Mostafa Jokar.

簡体字中国語への翻訳：Antiy Labs(antiy.cn)、Archer

韓国語への翻訳：Byungho Min

オランダ語への翻訳：Cedric Sambre（AKA Midas）

スペイン語への翻訳：Diego Boy, Luis Alberto Espinosa Calvo, Fernando Guida, Diogo Mussi, Patricio Galdames

ポルトガル語への翻訳：Thales Stevan de A. Gois, Diogo Mussi

イタリア語への翻訳：Federico Ramondino[11], Paolo Stivanin[12], twyK

フランス語への翻訳：Florent Besnard[13], Marc Remy[14], Baudouin Landais, Téo Dacquet[15], BlueSkeye@GitHub[16]

ドイツ語への翻訳：Dennis Siekmeier[17], Julius Angres[18], Dirk Loser[19], Clemens Tamme

ポーランド語への翻訳：Kateryna Rozanova, Aleksander Mistewicz, Wiktoria Lewicka

日本語への翻訳：shmz@github[20].

校正者：Alexander 「Lstar」 Chernenkiy, Vladimir Botov, Andrei Brazhuk, Mark "Logxen" Cooper, Yuan Jochen Kang, Mal Malakov, Lewis Porter, Jarle Thorsen, Hong Xie.

Vasil Kolev[21] は大量の校正とミスを訂正してくれました。

イラストとカバーアート：Andy Nechaevsky

github.comでフォークして指摘や訂正してくれたすべての方々に感謝します：[22]

---

[5] reddit
[6] twitter.com/sergeybratus/status/505590326560833536
[7] twitter.com/TanelPoder/status/524668104065159169
[8] https://github.com/pixjuan
[9] https://github.com/73696e65
[10] https://github.com/TheRenaissance
[11] https://github.com/pinkrab
[12] https://github.com/paolostivanin
[13] https://github.com/besnardf
[14] https://github.com/mremy
[15] https://github.com/T30rix
[16] https://github.com/BlueSkeye
[17] https://github.com/DSiekmeier
[18] https://github.com/JAngres
[19] https://github.com/PolymathMonkey
[20] https://github.com/shmz
[21] https://vasil.ludost.net/
[22] https://github.com/DennisYurichev/RE-for-beginners/graphs/contributors

LATEX パッケージをたくさん使っています：パッケージの作者にも合わせて感謝します

ドナー

本の大半を執筆中にサポートしてくれた方々：

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 (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).

ドナーの方すべてに感謝します！

mini-FAQ

Q：この本を読むための前提条件は何ですか？

A：C/C ++の基本的な理解があるのが望ましいです。

Q：x86/x64/ARMとMIPSを本当にすぐに学ぶべきでしょうか？それはあまりにも大変ではないですか？

A：初心者は、ARMとMIPSの部分をスキップまたはスキミングしながら、x86/x64だけを読んでもいいです。

Q：ロシア語または英語のハードカバー/ペーパーブックを購入できますか？

A：残念ながら、いいえ。これまでにロシア語版や英語版を出版することに興味を持った出版社はいませんでした。その間に、お気に入りのコピーショップに印刷してバインドするよう依頼することができます。

Q：epubまたはmobiのバージョンはありますか？

A：いいえ。本はTeX / LaTeX固有のハッキングに強く依存しているので、HTML（epub / mobiはHTMLの集合です）への変換は簡単ではありません。

Q：最近、アセンブリ言語を学ばなければならないのはなぜですか？

A：あなたがOS[23]開発者でない限り、アセンブラでコード化する必要はないでしょう。最新のコンパイラ（2010s）は、人間よりも最適化を実行する方がはるかに優れています[24]

また、最新のCPUは非常に複雑なデバイスであり、アセンブリの知識は内部を理解するのに役立つものではありません。

それは、少なくとも2つの領域があり、アセンブリの理解を深めることが役立つことがあります。まず、セキュリティ/マルウェアの研究に役立つことです。また、デバッグ中にコンパイルされたコードをよりよく理解するための良い方法です。したがって、この本は、アセンブリ言語を記述するのではなく、アセンブリ言語を理解したい人のために用意されています。そのため、コンパイラ出力の例が多数含まれています。

Q：PDF文書内のハイパーリンクをクリックしましたが、どのように戻ってきますか？

A：Adobe Acrobat ReaderでAlt+左矢印をクリックします。Evinceで“ <”ボタンをクリックしてください。

Q：この本を印刷して教えてもいいですか？

A：もちろん！だからこの本はクリエイティブコモンズライセンス（CC BY-SA 4.0）に基づいてライセンスされています。

Q：なぜこの本は無料ですか？あなたは素晴らしい仕事をしてきました。これは他の多くの自由なものと同様に疑わしいものです。

---

[23] Japanese text placeholder

[24] このトピックに関する非常に良いテキスト：[Agner Fog, *The microarchitecture of Intel, AMD and VIA CPUs*, (2016)]

A：私自身の経験では、技術文献の著者は主に自己宣伝の目的で書いています。そのような仕事からまともな金を稼ぐことはできません。

Q：リバースエンジニアリングではどのように仕事をしていますか？

A：redditには時々現れる採用スレッドがあり、RE[25](2016) に専念しています。そこを見てみてください。

多少関連する採用スレッドは、netsecサブディレクトリ (2016) にあります。

Q：一般的にプログラミングを学ぶにはどうすればいいですか？

A：C言語とLISP言語の両方をマスターすると、プログラマーの生活はずっと簡単になります。私は [Brian W. Kernighan, Dennis M. Ritchie, *The C Programming Language*, 2ed, (1988)] と**SICP!**[26]の練習問題を解くことをお勧めします。

Q：質問があります...

A：私にメールしてください (dennis@yurichev.com)

韓国語版について

2015年1月、韓国のAcorn出版社 (www.acornpub.co.kr) は、この書籍を2014年8月に韓国語に翻訳して出版する際に膨大な作業をしました。

現在、彼らのウェブサイトで利用可能です。

翻訳者はByungho Min (twitter/tais9) です。カバーアートは、作家の友人Andy Nechaevsky(facebook/andydinka) によって行われました。Acornには、韓国語翻訳の著作権もあります。

そして、あなたが韓国語で本棚にリアルな本がほしくて、この作品をサポートしたいなら、現在購入可能です。

ペルシア/ファルシ語版について

2016年にこの本はMohsen Mostafa Jokar（Radadeのマニュアルを翻訳していて、イランのコミュニティにも知られています）によって翻訳されました。出版社のウェブサイト（Pendare Pars）で入手できます。

ここに40ページの抜粋へのリンクがあります：`https://beginners.re/farsi.pdf`

イラン国立図書館登録情報：`http://opac.nlai.ir/opac-prod/bibliographic/4473995`

中国語版について

2017年4月、中国語への翻訳は中国のPTPressによって完了しました。中国語の著作権者でもあります。

中国語版はこちらから注文できます：`http://www.epubit.com.cn/book/details/4174`

翻訳の背後にある部分的なレビューと履歴は、ここにあります：`http://www.cptoday.cn/news/detail/3155`

主な翻訳者はArcherです。彼は非常に細心の注意を払っており、この本のような文献では非常に重要な既知の間違いやバグのほとんどを報告していました。私は彼のサービスを他の著者に推薦します！

Antiy Labsのスタッフも翻訳を手伝ってくれました。ここに彼らが書いた序文があります。

---

[25] reddit.com/r/ReverseEngineering/

[26] **SICP!**

第 **1** 章

コードパターン

第**1.1**節方法

この本の著者はC言語、その後Cppを学び始めたとき、小さなコードを書いてコンパイルし、アセンブリ言語の出力を見ていました。これにより、彼が書いたコードで何が起こっているのかを理解することが非常に容易になりました。[1] 彼はこれを何度もやって、Cppコードとコンパイラが作り出したものとの関係が彼の心の中に深く刻まれていたことを知っています。今では、Cコードの外観と機能の概要を即座に想像するのは簡単です。おそらく、このテクニックは他の人に役立つかもしれません。

なお、PCにインストールせずに、さまざまなコンパイラを使ってPCと同じことができる素晴らしいWebサイトがあります。あなたもそれを使うことができます：`https://godbolt.org/`

練習問題

この本の作者がアセンブリ言語を学んだとき、彼はしばしば小さなC関数をコンパイルしてから、アセンブリを徐々に書き直してコードを可能な限り短くしようとしました。効率性の点で最新のコンパイラと競争するのは難しいため、現実のシナリオではこれはおそらく価値がありません。しかし、それはアセンブリのより良い理解を得るための非常に良い方法です。したがって、この本の中からアセンブリコードを取り出して短くしてみてください。しかし、あなたが書いたものをテストすることを忘れないでください。

最適化レベルとデバッグ情報

ソースコードはさまざまな最適化レベルを持つ異なるコンパイラによってコンパイルできます。典型的なコンパイラにはこのようなレベルが約3つあります。レベル0は最適化が完全に無効になっていることを意味します。最適化は、コードサイズやコードの速度に合わせることもできます。最適化されていないコンパイラはより高速でより理解しやすいコードを生成しますが、最適化コンパイラは遅くなり、実行速度の速いコードを生成しようとします（コンパクトである必要はありません）。最適化レベルに加えて、コンパイラは結果ファイルにいくつかのデバッグ情報を含めて、デバッグしやすいコードを生成することができます。「デバッグ」コードの重要な機能の1つは、ソースコードの各行とそれぞれのマシンコードアドレスとの間にリンクを含む可能性があることです。一方、コンパイラを最適化すると、ソースコードの行全体が最適化され、結果のマシンコードにも存在しない出力が生成される傾向があります。リバースエンジニアはいずれかのバージョンに遭遇する可能性があります。なぜなら、一部の開発者はコンパイラの最適化フラグをオンにし、他の開発者はそうしないからですこのため、可能であれば、本書に記載されているコードのデバッグ版とリリース版の両方の例を取り上げようとします。

最も短い（または最も単純な）コードスニペットを得るために、時にはかなり古いコンパイラがこの本で使われています。

---

[1] 実際、彼はコードの特定のビットが何をしているのか理解できないときもこれを実行します

第1.2節基本的な事柄

第1.2.1節簡単なCPU入門

CPU[2]は、プログラムが構成するマシンコードを実行するデバイスです。

短い用語集

命令 : プリミティブCPUコマンド。最も単純な例としては、レジスタ間のデータの移動、メモリの操作、プリミティブ算術演算などがあります。一般に、各CPUには独自の命令セットアーキテクチャ（ISA）があり、

マシンコード : CPUが直接処理するコード。各命令は、通常、数バイトで符号化されます。

アセンブリ言語 : ニーモニックコードと、マクロのようないくつかの拡張機能は、プログラマーの人生をより簡単にするためのものです。

CPUレジスタ : 各CPUには汎用レジスタ（GPR）の固定セットがあります。x86では約8、x86-64では約16、ARMでは約16です。レジスタを理解する最も簡単な方法は、それを型なしの一時変数と考えることです。高水準のPLで作業していて、8つの32ビット（または64ビット）変数しか使用できないとしたらどうでしょうか？しかし、これらを使って多くのことを行うことができます！

機械コードとPLの違いが必要な理由は不思議です。答えは、人間とCPUが似ていないという事実にあります。人間がCpp、Java、Pythonなどの高レベルのPLを使う方がはるかに簡単ですが、CPUがはるかに低いレベルを使用する方が簡単です。抽象化のおそらく、高レベルのPLコードを実行できるCPUを発明することは可能かもしれませんが、今日われわれが知っているCPUの何倍も複雑なものになるでしょう。同様の方法で、人間がアセンブリ言語で書くことは非常に不便です。なぜなら、それは低レベルであり、厄介な間違いを大量に作成することなく書き込むことが難しいからです。上位PLコードをアセンブリに変換するプログラムをコンパイラと呼びます。[3]

異なる **ISA[4]s** について**2、3**

x86 ISAは常に可変長命令を持っていたので、64ビット時代になるとx64拡張はISAに非常に大きな影響を与えませんでした。実際、x86 ISAには、16ビットの8086 CPUに最初に登場した命令がまだ多く含まれていますが、今日のCPUではまだ見つかっています。ARMは一定の長さの命令を念頭に置いて設計されたRISC[5] CPUであり、過去にいくつかの利点がありました。当初、すべてのARM命令は4バイトでエンコードされていました [6] これは現在、「ARMモード」と呼ばれています。それから、彼らは最初に想像したほど倹約的ではないことに気付きました。実際のアプリケーションで最も一般的なCPU命令（MOV/PUSH/CALL/Jccなど）は、より少ない情報を使用してエンコードできます。したがって、Thumbと呼ばれる別のISAを追加しました。そこでは、各命令はわずか2バイトでエンコードされていました。これを「Thumbモード」と呼びます。ただし、すべてのARM命令が2バイトでエンコードできるわけではないため、Thumb命令セットは多少制限されています。ARMモードとThumbモード用にコンパイルされたコードは、1つのプログラム内で共存できることに注意してください。ARMの作成者は、Thumbを拡張して、ARMv7に登場したThumb-2を生み出すことができると考えました。Thumb-2はまだ2バイトの命令を使用しますが、4バイトのサイズを持ついくつかの新しい命令があります。Thumb-2はARMとThumbが混在しているという誤解が一般的です。これは間違っています。むしろThumb-2はすべてのプロセッサ機能を完全にサポートするように拡張されており、ARMモードと競合する可能性があります。これは明らかに達成された目標で、 idevicesの大部分のアプリケーションはThumb-2命令セット用にコンパイルされています。（確かに、これは主にXcodeがデフォルトで行うためです）。後で64ビットARMが出ました。このISAには4バイトの命令があり、Thumbモードを追加する必要はありませんでした。しかし、64ビットの要件がISAに影響を与え、ARMモード、Thumbモード（Thumb-2を含む）、ARM64という3つのARM命令セットを持つようになりました。これらのISAは部分的に交差するが、同じISAであると言える。したがって、この本では3つのARM ISAすべてにコードの断片を追加しようとします。ところで、MIPS、PowerPC、Alpha AXPなど固定長の32ビット命令を持つ他の多くのRISC ISAがあります。

第1.2.2節数値システム

おそらくほとんどの人に10本の指があるので、人間は10進数字システムに慣れてきました。それにもかかわらず、数「10」は科学と数学では重要な意味を持ちません。デジタル電子機器の自然数システムはバイナリです.0は電線に電流が流れていないことを表し、1は存在を表します。バイナリで10は10進数で2、バイナリで100は小数点で4などです。

---

[2] Central Processing Unit

[3] 旧式のロシア文学でも、翻訳者という用語が使われています。

[4] **Instruction Set Architecture**

[5] Reduced Instruction Set Computing

[6] 固定長命令は、労力を要することなく次（または前）の命令アドレスを計算できるため、便利です。この機能については、switch () オペレータセクションで説明します。

数値システムが10桁の場合、基数は10です。2進数字システムの基数は2です。

思い出すべき重要なこと：

1）数字は数字であり、桁は書記体系からの言葉であり、通常は1文字

2）数値の値は別の基数に変換されても変更されません。その値に対する書き込み表記だけが変更されています（したがって、RAM[7]で表現する方法)。

第**1.2.3**節**1**つの基数から別の基数への変換

位置表記はほぼすべての数値システムで使用されます。これは、数字が大きな数字の中に置かれている場所に対する相対的な重みを持つことを意味します。2が右端に置かれている場合は2ですが、右端の前に1桁置かれている場合は20です。

1234は何を表しますか？

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

バイナリの数字は同じですが、ベースは10ではなく2です.0b101011は何を表していますか？

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

ローマ数字のような非定位表記のようなものがあります。[8] おそらく人類は紙で基本的な操作（加算、乗算など）を手作業で行う方が簡単であるため、位置表記法に切り替えました。

二進数は、学校で教えられたのと同じように追加、減算などが可能ですが、2桁しか利用できません。

2進数は、ソースコードとダンプで表現されているときにはかさばります。したがって、16進数表記が役立ちます。16進の基数は、0..9の数字と6つのラテン文字A..Fを使用します。各16進数字は4ビットまたは4バイナリの数字を取るので、バイナリの数字から16進数に変換したり、手動でさえ戻したりするのは非常に簡単です。

hexadecimal	binary	decimal
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

特定のインスタンスでどの基数が使用されているかをどのように知ることができますか？

小数点は通常1234のように書かれます。アセンブラの中には小数点以下の基数に識別子を付けることができますが、数字には接尾辞d（1234d）が付きます。

2進数には、接頭辞”0b” が付いていることがあります：0b100110111（GCC[9]にはこのための非標準言語拡張があります[10]）

もう1つの方法もあります。たとえば、”b” 接尾辞を使用します（例：100110111b)。この本では、バイナリ番号のために本の中で一貫して”0b” という接頭辞を使用しようとしています。

16進数の先頭には、Cppや他のPL：0x1234ABCDの接頭辞「0x」が付加されています。あるいは、”h” 接尾辞1234ABCDhが与えられます。これはアセンブラとデバッガでそれらを表現する一般的な方法です。この規則では、数字がLatin（A..F）桁で始まる場合、先頭に0が追加されます（0ABCDEFh)。ABCDのような $接頭辞を使っ

---

[7] Random-Access Memory

[8] 数値システムの進化については、195-213を参照してください。

[9] GNU Compiler Collection

[10] `https://gcc.gnu.org/onlinedocs/gcc/Binary-constants.html`

て8ビットの家庭用コンピュータ時代に普及した大会もありました。この本は16進数のために本の中に”0x” というプレフィックスを付けようとします。

数字を精神的に変換することを学ぶべきでしょうか？1桁の16進数の表を簡単に記憶できます。大きな数字については、おそらく自分自身を苦しめる価値はありません。

おそらく最も目に見える16進数はURLにあります。これは非ラテン文字がコード化される方法です。たとえば`https://en.wiktionary.org/wiki/na%C3%AFvet%C3%A9`は、「naïveté」という単語に関するWiktionaryの記事のURL[11]です。

8進数

コンピュータプログラミングの過去に多用された別の数字システムは8進数である。8進数では8桁（0..7）であり、それぞれが3ビットにマッピングされるので、数値を前後に変換するのは簡単です。ほぼすべての場所で16進法に取って代わられていますが、驚くべきことに、多くの人が頻繁に使う*NIXユーティリティがあります。これは引数として8進数をとります (chmod)。

多くの* NIXユーザーが知っているように、`chmod` 引数は3桁の数字にすることができます。最初の桁はファイル所有者の権利（読み込み、書き込み、実行）を表し、2番目はファイルが属するグループの権利で、3番目は他の人の権利です。`chmod` がとる各数字はバイナリ形式で表すことができます：

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

したがって、各ビットはフラグread / write / executeにマップされます。

ここで `chmod` の重要性は、引数の整数全体を8進数で表現できることです。`chmod 644 file` を実行すると、所有者の読み取り/書き込み権限、グループの読み取り権限、他のユーザーの読み取り権限が再度設定されます。8進数644を2進数に変換すると、`110100100`、または3ビットのグループで `110 100 100` になります。

各トリプレットは所有者/グループ/その他のパーミッションを記述しています。最初は `rw-`、2番目は `r--`、3番目は `r--`です。

8進数字システムは、PDP-8のような古いコンピュータでも人気がありました。なぜなら、そこには12,24、または36ビットが存在する可能性があり、これらの数値はすべて3で割り切れるからです。今日、普及しているすべてのコンピュータは16,32,64ビットのワード/アドレスサイズを使用しており、これらの数値はすべて4で割り切れるため、16進数のシステムはより自然です。

すべての標準Cppコンパイラでサポートされています。これは混乱の原因となることがあります。なぜなら、8進数はゼロの前に付加されています（0377は255など）。時には、タイプミスをして9の代わりに”09” と書くことがあります。GCCは次のようなことを報告するかもしれません：`error: 無効な数字"9" は8進定数です`

また、8進数のシステムは、Javaではやや人気があります。IDAが印刷不可能な文字を含むJava文字列を表示すると、16進数ではなく8進数でエンコードされます。JAD Javaデコンパイラも同じように動作します。

除算能力

120のような10進数を見ると、最後の桁がゼロであるため、itfsを10で割り切れるとすぐに推論することができます。同様に、最後の2桁が0であるため、123400は100で割り切れる。同様に、16進数の0x1230は0x10（または16）で割り切れ、0x123000は0x1000（または4096）で割り切れる

バイナリ番号0b1000101000は0b1000 (8) などで割り切れます。このプロパティは、メモリ内の一部のブロックのサイズがある境界に埋め込まれているかどうかを素早く認識するためによく使用されます。たとえば、PE12ファイルのセクションは、ほとんどの場合、0x41000、0x10001000など3つの16進ゼロで終わるアドレスで開始

---

[11] Uniform Resource Locator

されます。これは、ほとんどすべてのPE[12]セクションが0x1000（4096）バイトの境界にパディングされているためです。

多精度算術演算と基数

多精度算術演算では膨大な数を使用でき、それぞれが数バイトで格納されます。たとえば、公開鍵と秘密鍵の両方のRSA鍵は、最大4096ビットに及んでいます。

[Donald E. Knuth, *The Art of Computer Programming*, Volume 2, 3rd ed., (1997), 265] において、我々は多バイト数で多倍精度の数値を格納するとき、整数を表すことができる28 = 256の基数を有するものとして、各桁は対応するバイトに進む。同様に、複数の精度の数値を複数の32ビットの整数値に格納すると、各桁は32ビットの各スロットに移動し、この数値は基数232に格納されていると考えることができます。

非小数点の発音方法

非小数点の基数の数字は、通常、桁で数字によって発音されます。イチ・ゼロ・ゼロ・イチ・イチ。10や1000のような言葉は、小数点の基本システムとの混同を避けるために、通常は発音されません。

浮動小数点数

浮動小数点数を整数から区別するために、浮動小数点数は通常 0.0, 123.0 などの末尾に.0で書かれています。

第1.3節空関数

可能な限り単純な関数は、何もしない関数です。

Listing 1.1:

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

コンパイルしましょう！

第1.3.1節x86

x86プラットフォーム上でGCCコンパイラとMSVCコンパイラの両方の生成物は次のとおりです。

Listing 1.2: 最適化 GCC/MSVC (アセンブリ出力)

```
f:
        ret
```

RET 命令のみです。callerに戻る命令です。

第1.3.2節ARM

Listing 1.3: 最適化 Keil 6/2013 (ARMモード) アセンブリ出力

```
f       PROC
        BX       lr
        ENDP
```

リターンアドレスはARM ISAのローカルスタックには保存されず、リンクレジスタに保存されるため、BX LR 命令は実行をそのアドレスにジャンプさせるため、callerへの実行が効果的に戻ります。

---

[12] Portable Executable

第**1.3.3**節**MIPS**

レジスタの命名には、数値（0□ 31）または擬似名（$V0$□ A0など）の2つの命名規則がMIPSの世界で使用されています。

以下のGCCアセンブリ出力は、レジスタを番号順にリストしています。

Listing 1.4: 最適化 GCC 4.4.5 (アセンブリ出力)

```
        j       $31
        nop
```

...IDA[13] は疑似名を使います

Listing 1.5: 最適化 GCC 4.4.5 (IDA)

```
        j       $ra
        nop
```

最初の命令は、実行フローをcallerに返すジャンプ命令（JまたはJR）で、$31（または $RA）レジスタのアドレスにジャンプします。

これはARMのLR[14]に類似したレジスタです。

2番目の命令はNOP[15]で、何もしません。私たちは今それを無視することができます。

MIPS命令とレジスタ名についての注意

MIPSの世界では、レジスタと命令の名前は、伝統的に小文字で書かれています。しかし、一貫性を保つために、この本は大文字の使用で通します。

第**1.3.4**節実際の空関数

空の関数は役に立たないように見えますが、低レベルのコードでは頻繁に使用されます。

まず第一に、次のようなデバッグ機能でとてもポピュラーです。

Listing 1.6:

```
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");

        ...
};
```

非デバッグビルドでは（greleasehのように）、`_DEBUG` は定義されていないので、`dbg_print()` 関数は実行中に呼び出されているにもかかわらず、空になります。

同様に、ソフトウェア保護の一般的な方法は、合法な顧客向けに1つのビルドを作成し、他にはデモビルドを作成することです。デモビルドには、この例のようにいくつかの重要な機能が欠けています。

---

[13] Japanese text placeholder

[14] Link Register

[15] No Operation

Listing 1.7:

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

`save_file()` 関数は、ユーザーがメニューの `File->Save` をクリックすると呼び出すことができます。デモ版は、このメニュー項目を無効にしてお届けできますが、ソフトウェアクラッカーが有効にしても、役に立つコードのない空の関数だけが呼び出されます。

IDAは、`nullsub_00`、`nullsub_01` などの名前でこのような機能をマークします。

第**1.4**節戻り値

もう1つの単純な関数は、単に定数値を返す関数です。

Listing 1.8:

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

コンパイルしてみましょう。

第**1.4.1**節**x86**

ここでは、GCCコンパイラとMSVCコンパイラの両方でx86プラットフォーム上で（最適化を使用して）生成されるものを示します。

Listing 1.9: 最適化 GCC/MSVC (アセンブリ出力)

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

命令はたった2つです：最初に値123を EAX レジスタに入れます。これは戻り値を格納するために慣例によって使用され、2つ目はcallerに実行を返す RET です。

呼び出し元は EAX レジスタから結果を取得します。

第**1.4.2**節**ARM**

ARMプラットフォームにはいくつかの違いがあります。

Listing 1.10: 最適化 Keil 6/2013 (ARMモード) ASM Output

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

ARMは関数の結果を返すためにレジスタ `R0` を使用するため、123が `R0` にコピーされます。

MOV は、x86とARMのISAの両方で命令の誤解を招く名前であることに注意する価値があります。

データは実際には移動されませんが、コピーされます。

第**1.4.3**節**MIPS**

以下のGCCアセンブリ出力は、レジスタを番号順にリストしています。

Listing 1.11: 最適化 GCC 4.4.5 (アセンブリ出力)

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

...IDA は擬似名でリストします

Listing 1.12: 最適化 GCC 4.4.5 (IDA)

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

$2（または $V0）レジスタは、関数の戻り値を格納するために使用されます。LI は「即時ロード」を表し、MOV に相当するMIPSです。

もう1つの命令は、実行フローをcallerに返すジャンプ命令（JまたはJR）です。

ロード命令（LI）とジャンプ命令（JまたはJR）の位置が入れ替えられたのはなぜだろうか。これは、“branch delay slot” と呼ばれるRISC機能が原因です。

これが起こる理由は、いくつかのRISC ISAのアーキテクチャーでは奇抜であり、私たちの目的にとって重要ではありません。MIPSでは、ジャンプ命令または分岐命令に続く命令は、ジャンプ/分岐命令自体の前に実行される。

結果として、分岐命令は、直前に実行された命令で常に場所を入れ替える。

実際には、単に1（真）または0（偽）を返す関数はよく使われます。

標準のUNIXユーティリティである*/bin/true* と */bin/false* の中でも最小のものがそれぞれ0と1を終了コードとして返します。（ゼロは終了コードとして通常成功を意味し、ゼロ以外はエラーを意味します)。

第**1.5**節ハローワールド!

[Brian W. Kernighan, Dennis M. Ritchie, *The C Programming Language*, 2ed, (1988)] という本の有名な例を使ってみましょう

Listing 1.13: C/C++ Code

```
#include <stdio.h>

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

第**1.5.1**節**x86**

MSVC

MSVC 2010でコンパイルしてみましょう。

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

(/Fa オプションは、アセンブリリストファイルを生成するようにコンパイラに指示します)

Listing 1.14: 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は、Intel構文でアセンブリリストを生成します。Intel構文とAT&T構文の違いについては、1.5.1 on the next pageで説明します。

コンパイラは、`1.exe` にリンクされる `1.obj` というファイルを生成しました。私たちの場合、ファイルにはCONST（データ定数用）と _TEXT（コード用）の2つのセグメントが含まれています。

C/C++の文字列 `hello, world` には、`const char[]`[Bjarne Stroustrup, *The C++ Programming Language, 4th Edition*, (2013)p176, 7.3.2] 型がありますが、独自の名前はありません。コンパイラは何らかの形で文字列を処理する必要があるため、内部名 `$SG3830` を定義します。

そのため、この例は次のように書き換えられます。

```
#include <stdio.h>

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

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

アセンブリリストに戻りましょう。わかるように、文字列は C/C++ 文字列の標準であるNULLバイトで終了します。C/C++ 文字列の詳細は：**??** on page ??

_TEXT というコードセグメントでは、`main()` 関数が1つしかありません。関数 `main()` は、プロローグコードで始まり、エピローグコードで終わります（ほぼすべての関数のように）[16]

関数のプロローグの後に、`printf()` 関数の呼び出しがあります。`CALL _printf`. 呼び出しの前に、PUSH 命令の助けを借りて、挨拶を含む文字列アドレス（またはそのポインタ）がスタックに置かれます。

`printf()` 関数が `main()` 関数に制御を返すと、文字列アドレス（またはそのポインタ）はまだスタック上にあります。もはや必要がないので、スタックポインタ（ESPレジスタ）を修正する必要があります。

`ADD ESP, 4` は ESP レジスタ値に4を加算することを意味します。

なぜ4？これは32ビットプログラムなので、スタックを通過するアドレスには正確に4バイトが必要です。x64コードの場合は8バイト必要です。`ADD ESP, 4` は `POP register` と事実上同等ですが、レジスタを使用しません[17]

同じ目的のために、インテルC++コンパイラのようなコンパイラの中には、ADD の代わりに `POP ECX` を発行するものもある（例えば、インテルC++コンパイラでコンパイルされているので、このようなパターンは Oracle RDBMS コードで見ることができる）。この命令はほとんど同じ効果を持ちますが、ECX レジスタの内容は上書きされます。インテルC++コンパイラは、この命令命令コードが `ADD ESP, x`（POP の場合は1バイト、ADD の場合は3バイト）よりも短いため、`POP ECX` を使用すると思われます。

Oracle RDBMS から ADD の代わりに POP を使用する例を次に示します。

Listing 1.15: Oracle RDBMS 10.2 Linux (app.o file)

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

`printf()` を呼び出した後、元の C/C++ コードには、`main()` 関数の結果として `return 0` というステートメントが含まれています。

生成されたコードでは、これは命令 `XOR EAX, EAX` によって実装されます。

---

[16] プロローグとエピローグ関数についてのセクションでは、詳細を見ていきます (1.6 on page 28)

[17] CPU flags, however, are modified

XOR は実際には「eXclusive OR」[18]ですが、コンパイラでは `MOV EAX, 0` の代わりに使用されることがよくあります。もう少し短いオペコード（MOV の場合は5に対して XOR の場合は2バイト）であるからです。

一部のコンパイラは `SUB EAX, EAX` を出力します。これは、EAX の値を EAX の値から 差し引くことを意味します。それはどんな場合でもゼロになります。

最後の命令RETは、callerに制御を返します。通常、これは C/C++ CRT[19]コードであり、これはOSに制御を戻します。

GCC

LinuxのGCC 4.4.1コンパイラと同じ C/C++ コードをコンパイルしてみましょう：`gcc 1.c -o 1` 次に、IDA 逆アセンブラの助けを借りて、どのように `main()` 関数が作成されるのかを見ていきましょう。IDA は、MSVCと同様に、Intel-syntaxを使用します。[20]

Listing 1.16: 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"
                mov     [esp+10h+var_10], eax
                call    _printf
                mov     eax, 0
                leave
                retn
main            endp
```

結果はほぼ同じです。helloのワールド文字列（データセグメントに格納されている）のアドレスは、最初にEAXレジスタにロードされ、スタックに保存されます。

さらに、関数プロローグには `AND ESP, 0FFFFFFF0h` があります。この命令は、ESP レジスタ値を16バイトの境界に揃えます。この結果、スタック内のすべての値が同じ方法で整列されます（処理中の値が4バイト境界または16バイト境界に整列したアドレスにある場合、CPUのパフォーマンスは向上します）。

`SUB ESP, 10h` はスタックに16バイトを割り当てます。しかし、私たちはこれから見るように、ここでは4つだけが必要です。

これは、割り当てられたスタックのサイズも16バイトの境界に揃えられているためです。

文字列アドレス（または文字列へのポインタ）は、PUSH 命令を使用せずにスタックに直接格納されます。*var_10* はローカル変数であり、`printf()` の引数でもあります。それについて下記を読んでください。

`printf()` 関数が呼び出されます。

MSVCと異なり、GCCは最適化をオンにしないでコンパイルすると、より短いオペコードの代わりに `MOV EAX, 0` を発行します。

最後の命令、LEAVE は、`MOV ESP, EBP`、および `POP EBP` 命令ペアと同等です。言い換えれば、この命令はstack pointer (ESP) を戻し、EBP レジスタを初期状態に戻す。これは、これらのレジスタ値（ESP と EBP ）を関数の始めに（`MOV EBP, ESP` / `AND ESP, …` を実行することによって）変更したので必要です。

GCC: AT&T構文

これをアセンブリ言語のAT&T構文でどのように表現できるかを見てみましょう。この構文は UNIXの世界でずっと人気があります。

Listing 1.17: let's compile in GCC 4.7.3

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

---

[18] Wikipedia

[19] C Runtime library

[20] `-S -masm=intel` オプションを適用することで、Intel構文でGCCのアセンブリリストを生成させることもできます

下記を得ます。

Listing 1.18: 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
        .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
```

リストには、多くのマクロ（ドットで始まる部分）が含まれています。現時点では興味深いものではありません。

今のところ、簡単にするため、無視してもかまいません（C言語の文字列のように終端文字列をエンコードする*.string* マクロを除く）。それからこれを見てみましょう。[21]

Listing 1.19: 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
```

インテルとAT&T構文の主な違いのいくつかは次のとおりです。

- ソースオペランドとデスティネーションオペランドは逆の順序で記述されます。

  インテル構文では：<命令> <デスティネーション・オペランド> <ソース・オペランド>

  AT&T構文の場合：<命令> <ソースオペランド> <デスティネーションオペランド>

  違いを覚えるのは簡単な方法です：インテル構文を扱うとき、オペランド間に等号 (=) があり、AT&T-syntaxを扱うときに右矢印 (→) があると想像してください。[22]
- AT&T：レジスタ名の前にはパーセント記号 (%) を、数字の前にはドル記号 ($) を書く必要があります。角カッコのかわりに丸カッコが使用されています。
- AT&T：オペランドサイズを定義する命令に接尾辞が追加されます

---

[21] このGCCオプションは「不要な」マクロを削除するために使用できます：*-fno-asynchronous-unwind-tables*

[22] ところで、いくつかのC標準関数（例えば、memcpy()、strcpy()）では、インテル構文と同じ方法で引数がリストされています。まず、デスティネーションメモリブロックへのポインタ、次にソースメモリブロックへのポインタです

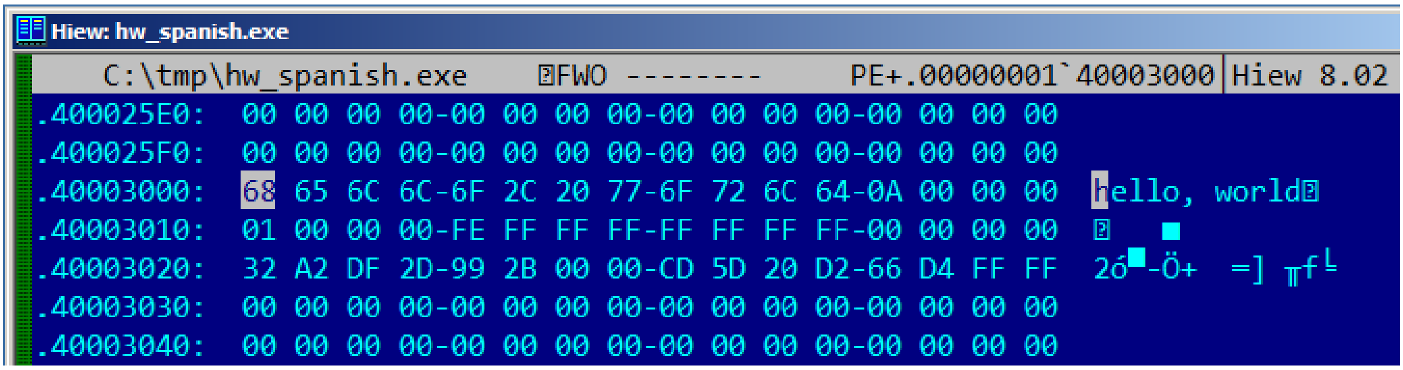
- q — quad (64 bits)
- l — long (32 bits)
- w — word (16 bits)
- b — byte (8 bits)

コンパイル結果に戻るには、IDA が表示した結果とほぼ同じです。微妙な違いが1つあります。0FFFFFFF0h は $-16 として表示されます。これは同じことです：10進数の 16 は16進数で 0x10 です。-0x10 は 0xFFFFFFF0 に等しくなります（32ビットデータ型の場合）。

もう1つ：戻り値は、XOR ではなく通常の MOV を使用して0に設定されます。MOV はレジスタに値をロードするだけです。誤ってつけられた名前です（データは移動せず、コピーされるため）。他のアーキテクチャでは、この命令の名前は「LOAD」または「STORE」などと同様です。

文字列のパッチ (Win32)

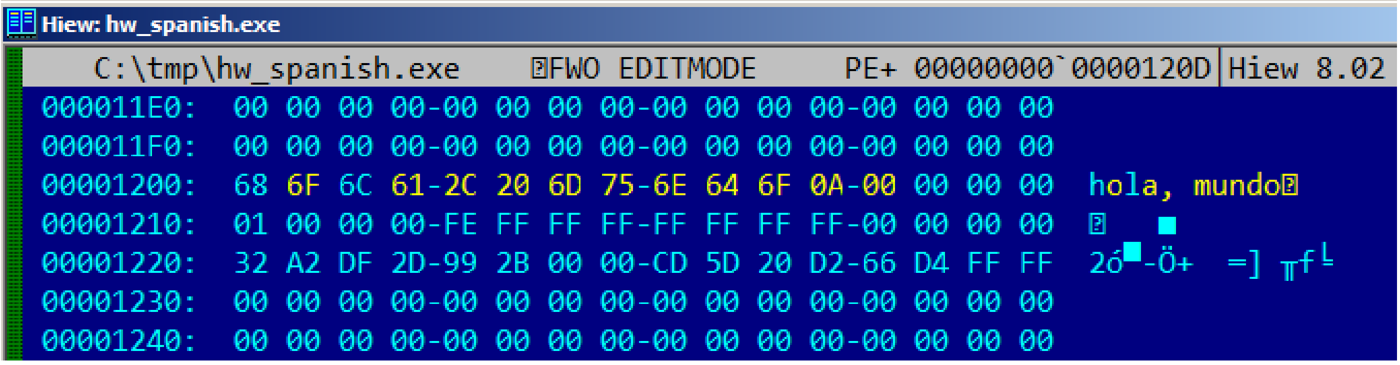
Hiewを使用して、実行可能ファイル内の “hello、world” 文字列を簡単に見つけることができます：



```
Hiew: hw_spanish.exe
    C:\tmp\hw_spanish.exe    ▯FWO --------      PE+.00000001`40003000 │Hiew 8.02
.400025E0:  00 00 00 00-00 00 00 00-00 00 00 00-00 00 00 00
.400025F0:  00 00 00 00-00 00 00 00-00 00 00 00-00 00 00 00
.40003000:  68 65 6C 6C-6F 2C 20 77-6F 72 6C 64-0A 00 00 00  hello, world▯
.40003010:  01 00 00 00-FE FF FF FF-FF FF FF FF-00 00 00 00  ▯   ■
.40003020:  32 A2 DF 2D-99 2B 00 00-CD 5D 20 D2-66 D4 FF FF  2ó▀-Ö+  =] ╥f╘
.40003030:  00 00 00 00-00 00 00 00-00 00 00 00-00 00 00 00
.40003040:  00 00 00 00-00 00 00 00-00 00 00 00-00 00 00 00
```

図 1.1: Hiew

メッセージをスペイン語に翻訳しようとすることができます



```
Hiew: hw_spanish.exe
    C:\tmp\hw_spanish.exe    ▯FWO EDITMODE      PE+ 00000000`0000120D │Hiew 8.02
000011E0:  00 00 00 00-00 00 00 00-00 00 00 00-00 00 00 00
000011F0:  00 00 00 00-00 00 00 00-00 00 00 00-00 00 00 00
00001200:  68 6F 6C 61-2C 20 6D 75-6E 64 6F 0A-00 00 00 00  hola, mundo▯
00001210:  01 00 00 00-FE FF FF FF-FF FF FF FF-00 00 00 00  ▯   ■
00001220:  32 A2 DF 2D-99 2B 00 00-CD 5D 20 D2-66 D4 FF FF  2ó▀-Ö+  =] ╥f╘
00001230:  00 00 00 00-00 00 00 00-00 00 00 00-00 00 00 00
00001240:  00 00 00 00-00 00 00 00-00 00 00 00-00 00 00 00
```

図 1.2: Hiew

スペイン語のテキストは英語より1バイト短くなっているので、最後に0x0Aバイト (\n) に続けてNULLバイトを追加しました。

うまくいきました。

より長いメッセージを挿入する場合はどうすればよいですか？元の英語テキストの後には、ゼロバイトがいくつかあります。上書きできるかどうかはなんとも言えません：CRTコードのどこかで使われるかもしれないし、そうでないかもしれません。とにかく、自分が行っていることを本当に知っていれば、それらを上書きするだけです。

文字列のパッチ (Linux x64)

rada.re を使ってLinux x64実行ファイルにパッチを当ててみましょう

Listing 1.20: 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
```

ここでは何が起こっているの：私は/コマンドを使用して「hello」文字列を検索し、そのアドレスにカーソル（rada.re 用語でシーク）を設定します。次に、これが本当にその場所であることを確かめたい：px がダンプします。oo+ はrada.re を読み書きモードに切り替えます。w は現在のシーク時にASCII文字列を書き込みます。最後の`\00` に注意してください。これはNULLバイトです。q で終了します。

MS-DOS時代におけるソフトウェアのローカライズ

このやり方は、1980年代と1990年代にMS-DOSソフトウェアをロシア語に翻訳する一般的な方法でした。ロシア語の言葉や文章は、英語の文章と比べて通常若干長いので、ローカライズされたソフトウェアには奇妙な頭字語やとても読みにくい略語が含まれています。

他の国の他の言語でも、この時代に起きていたことでしょう。

第**1.5.2**節**x86-64**

MSVC: x86-64

64ビットのMSVCも試してみましょう。

Listing 1.21: 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
```

x86-64では、すべてのレジスタが64ビットに拡張されましたが、その名前には R-プレフィックスが付いています。スタックをあまり頻繁に使用しないように（言い換えると、外部メモリ/キャッシュにアクセスする頻度を減らす）、レジスタ (*fastcall*) **??** on page ??を介して関数引数を渡す一般的な方法があります。すなわち、関数の引数の一部はレジスタに渡され、残りはスタックに渡されます。Win64では、4つの関数引数が RCX 、RDX 、R8 、および R9 レジスタに渡されます。ここで見るものは：`printf()` の文字列へのポインタはスタックにではなく、RCX レジスタに渡されます。ポインタは現在64ビットであるため、64ビットレジスタ（R-プレフィックスを持つ）に渡されます。ただし、下位互換性を保つために、E-接頭辞を使用して32ビットのパーツにアクセスすることは可能です。これは、RAX/EAX/AX/AL レジスタがx86-64のように見える方法です。

バイトの並び順							
第7	第6	第5	第4	第3	第2	第1	第0
RAX^{x64}							
				EAX			
						AX	
						AH	AL

`main()` 関数は *int* 型の値を返します。これは C/C++ では32ビットのままで、下位互換性と移植性を向上させるため、関数終了時に RAX レジスタの代わりに EAX レジスタがクリアされる理由です。（すなわちレジスタの32ビットの部分）ローカルスタックには40バイトも割り当てられています。これは「シャドースペース」と呼ばれます。これについては後で説明します：1.10.2 on page 99

GCC: x86-64

64ビット環境のLinuxでGCCを試してみましょう。

Listing 1.22: GCC 4.4.6 x64

```
.string "hello, world\n"
main:
        sub     rsp, 8
        mov     edi, OFFSET FLAT:.LC0 ; "hello, world\n"
        xor     eax, eax  ; number of vector registers passed
        call    printf
        xor     eax, eax
        add     rsp, 8
        ret
```

Linux、*BSDと Mac OS X は関数引数をレジスタに渡すためのメソッドも使います: [Michael Matz, Jan Hubicka, Andreas Jaeger, Mark Mitchell, *System V Application Binary Interface. AMD64 Architecture Processor Supplement*, (2013)] [23]

最初の6つの引数は、RDI, RSI, RDX, RCX, R8 および R9 レジスタに渡され、残りはスタックを介して渡されます。

そのため、文字列へのポインタは EDI （レジスタの32ビット部分）に渡されます。なぜ64ビット版の RDI を使用しないのでしょうか。

下位の32ビットレジスタ部分に何かを書き込む64ビットモードのすべての `MOV` 命令も上位32ビットをクリアすることが重要です（インテルマニュアル：7.1.4 on page 164を参照）。つまり、`MOV EAX, 011223344h` は、上位ビットがクリアされるため、RAX に値を正しく書き込みます。

コンパイルされたオブジェクトファイル（.o）を開くと、すべての命令のオペコードも見ることができます。[24]:

Listing 1.23: 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
```

---

[23] 以下で利用可能 `https://software.intel.com/sites/default/files/article/402129/mpx-linux64-abi.pdf`

[24] これは オプション → ディスアセンブル → オペコードバイト数で有効になるはずです

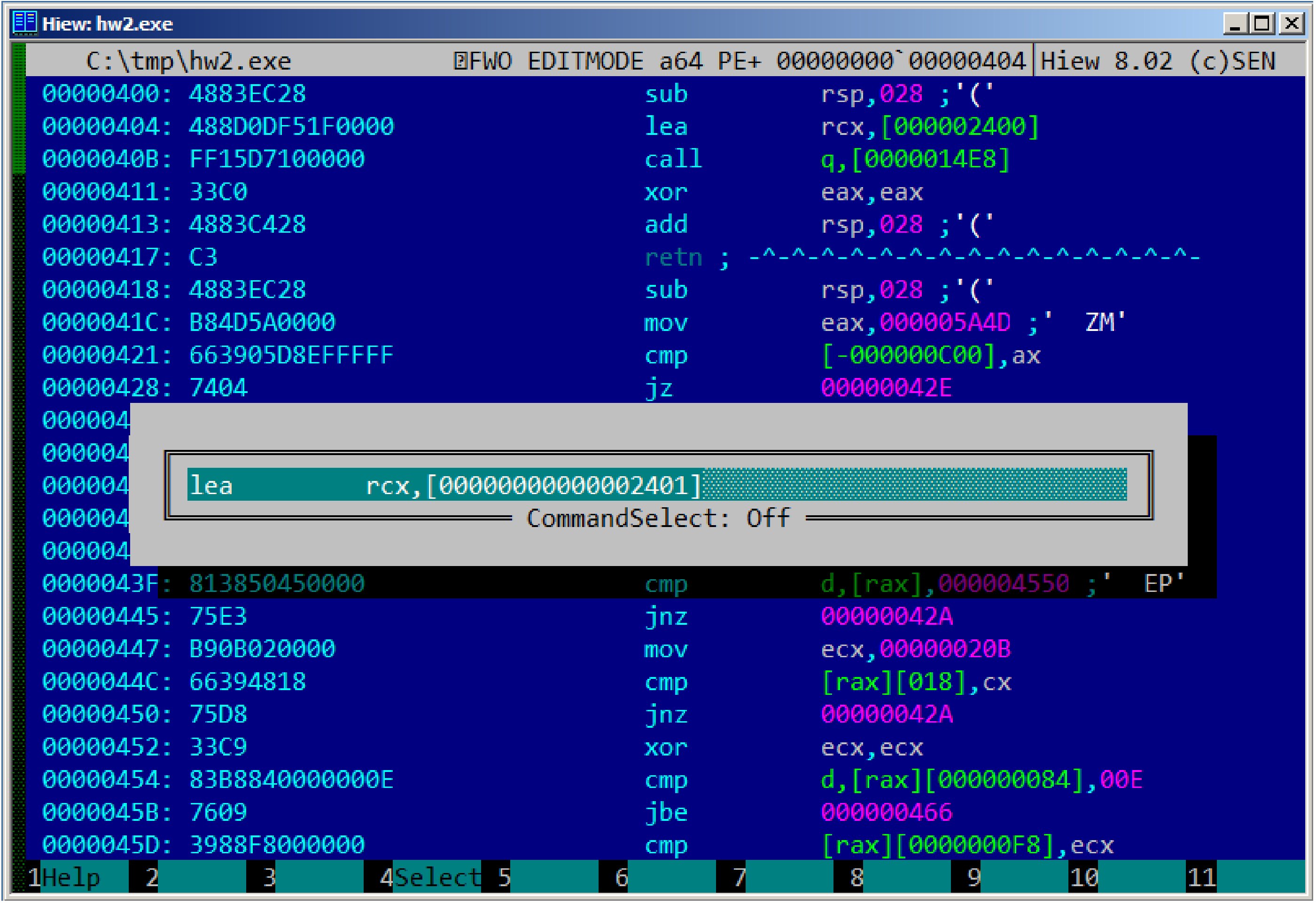
```
.text:00000000004004DB E8 D8 FE FF FF    call    _printf
.text:00000000004004E0 31 C0             xor     eax, eax
.text:00000000004004E2 48 83 C4 08       add     rsp, 8
.text:00000000004004E6 C3                retn
.text:00000000004004E6                   main  endp
```

ご覧のように、`0x4004D4` の `EDI` に書き込む命令は5バイトを占有します。64ビット値を `RDI` に書き込む同じ命令は7バイトを占有します。明らかに、GCCはいくらかのスペースを節約しようとしています。さらに、文字列を含むデータセグメントが4GiB以上のアドレスに割り当てられないことが保証されます。

また、`printf()` 関数呼び出しの前に `EAX` レジスタがクリアされていることがわかります。上記の**ABI!**[25]標準によれば、使用されたベクトルレジスタの数はx86-64の*NIXシステムで `EAX` に渡されるためです。

アドレスのパッチ (Win64)

この例が\MD スイッチ（`MSVCR*.DLL` ファイルリンケージのために小さな実行可能ファイルを意味します）を使用してMSVC 2013でコンパイルされた場合、`main()` 関数が最初に来て簡単に見つかります



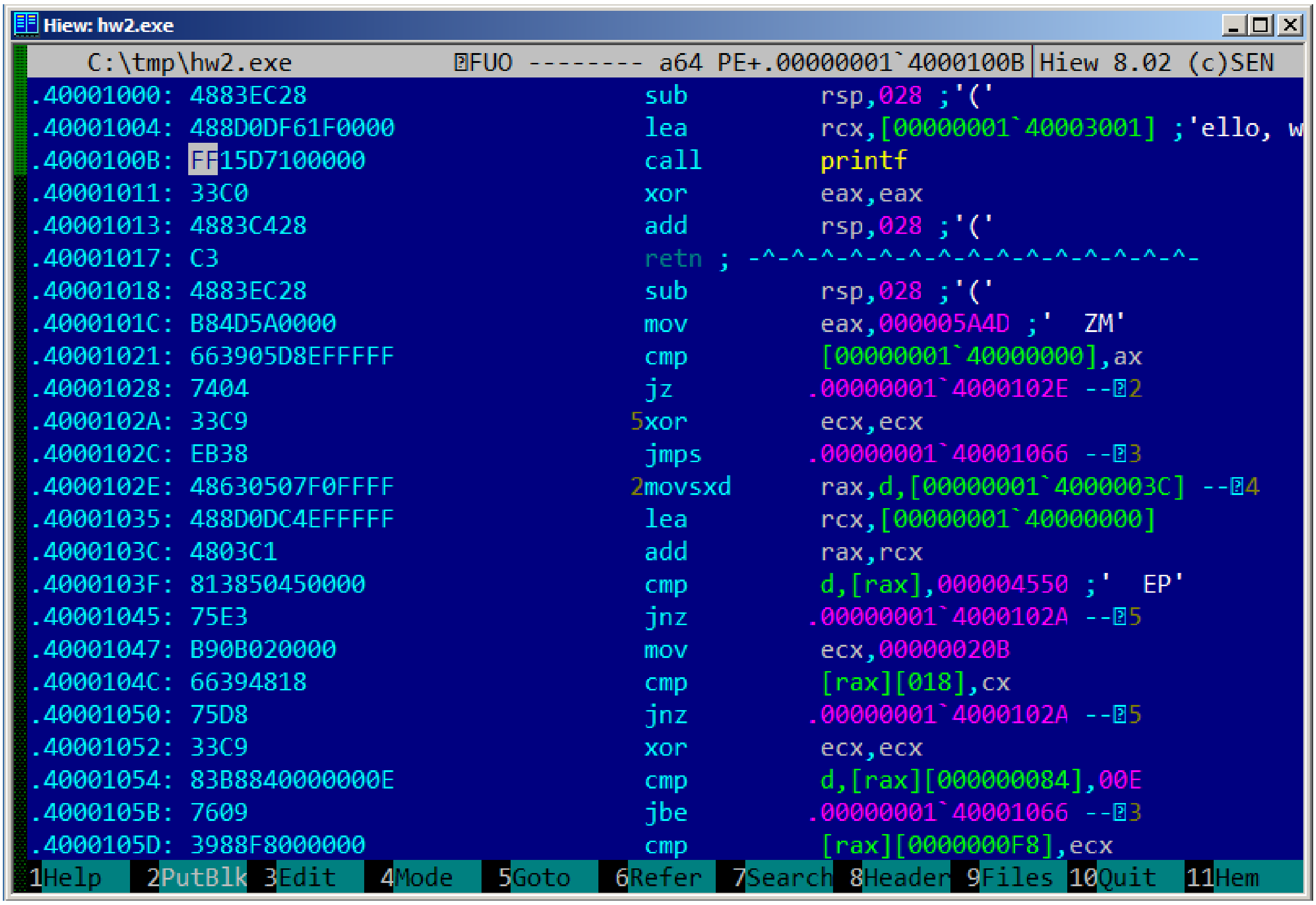
```
   C:\tmp\hw2.exe            ␦FWO EDITMODE a64 PE+ 00000000`00000404 │Hiew 8.02 (c)SEN
 00000400: 4883EC28                      sub         rsp,028 ;'('
 00000404: 488D0DF51F0000                lea         rcx,[000002400]
 0000040B: FF15D7100000                  call        q,[0000014E8]
 00000411: 33C0                          xor         eax,eax
 00000413: 4883C428                      add         rsp,028 ;'('
 00000417: C3                            retn ; -^-^-^-^-^-^-^-^-^-^-^-^-^-^-^-
 00000418: 4883EC28                      sub         rsp,028 ;'('
 0000041C: B84D5A0000                    mov         eax,000005A4D ;'  ZM'
 00000421: 663905D8EFFFFF                cmp         [-000000C00],ax
 00000428: 7404                          jz          00000042E
 000004
 000004
 000004  lea          rcx,[00000000000002401]
 000004  ═════════════ CommandSelect: Off ═════════════
 000004
 0000043F: 813850450000                  cmp         d,[rax],000004550 ;'  EP'
 00000445: 75E3                          jnz         00000042A
 00000447: B90B020000                    mov         ecx,00000020B
 0000044C: 66394818                      cmp         [rax][018],cx
 00000450: 75D8                          jnz         00000042A
 00000452: 33C9                          xor         ecx,ecx
 00000454: 83B8840000000E                cmp         d,[rax][000000084],00E
 0000045B: 7609                          jbe         000000466
 0000045D: 3988F8000000                  cmp         [rax][0000000F8],ecx
1Help  2      3      4Select 5      6      7      8      9      10     11
```

図 1.3: Hiew

実験として、アドレスを1ずつincrementすることができます：

---

[25] **ABI!**



```
Hiew: hw2.exe
    C:\tmp\hw2.exe          ▯FUO -------- a64 PE+.00000001`4000100B Hiew 8.02 (c)SEN
.40001000: 4883EC28                  sub         rsp,028 ;'('
.40001004: 488D0DF61F0000            lea         rcx,[00000001`40003001] ;'ello, w
.4000100B: FF15D7100000              call        printf
.40001011: 33C0                      xor         eax,eax
.40001013: 4883C428                  add         rsp,028 ;'('
.40001017: C3                        retn ; -^-^-^-^-^-^-^-^-^-^-^-^-^-^-^-
.40001018: 4883EC28                  sub         rsp,028 ;'('
.4000101C: B84D5A0000                mov         eax,000005A4D ;'  ZM'
.40001021: 663905D8EFFFFF            cmp         [00000001`40000000],ax
.40001028: 7404                      jz         .00000001`4000102E --▯2
.4000102A: 33C9                     5xor         ecx,ecx
.4000102C: EB38                      jmps       .00000001`40001066 --▯3
.4000102E: 48630507F0FFFF           2movsxd      rax,d,[00000001`4000003C] --▯4
.40001035: 488D0DC4EFFFFF            lea         rcx,[00000001`40000000]
.4000103C: 4803C1                    add         rax,rcx
.4000103F: 813850450000              cmp         d,[rax],000004550 ;'  EP'
.40001045: 75E3                      jnz        .00000001`4000102A --▯5
.40001047: B90B020000                mov         ecx,00000020B
.4000104C: 66394818                  cmp         [rax][018],cx
.40001050: 75D8                      jnz        .00000001`4000102A --▯5
.40001052: 33C9                      xor         ecx,ecx
.40001054: 83B8840000000E            cmp         d,[rax][000000084],00E
.4000105B: 7609                      jbe        .00000001`40001066 --▯3
.4000105D: 3988F8000000              cmp         [rax][0000000F8],ecx
 1Help   2PutBlk  3Edit    4Mode    5Goto    6Refer   7Search  8Header  9Files  10Quit   11Hem
```

図 1.4: Hiew

Hiewは 「ello, world」を示しています。そしてパッチが適用された実行可能ファイルを実行すると、この文字列が表示されます。

バイナリイメージから他の文字列を抜き取る (Linux x64)

Linux x64ボックスでGCC 5.4.0を使用して私たちの例をコンパイルしたときに得たバイナリファイルには、他の多くのテキスト文字列があります。ほとんどはインポートされた関数名とライブラリ名です。

objdumpを実行して、コンパイル済みファイルのすべてのセクションの内容を取得します。

```
$ 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.

...
```

テキスト文字列「/lib64/ld-linux-x86-64.so.2」のアドレスを `printf()` に渡すのは問題ではありません

```
#include <stdio.h>

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

信じがたいですが、このコードは前述の文字列を表示します。

アドレスを `0x400260` に変更すると、「GNU」文字列が出力されます。このアドレスは、私の特定のGCCバージョン、GNUツールセットなどに当てはまります。あなたのシステムでは、実行ファイルは若干異なる場合があり、すべてのアドレスも異なります。また、このソースコードに/からコードを追加/削除すると、おそらくすべてのアドレスが前後に移動します。

第**1.5.3**節**GCC—**もう**1**つ

匿名の C文字列が *const* 型 (1.5.1 on page 9) を持ち、定数セグメントに割り当てられたC文字列が不変であることが保証されているという事実は興味深い結果をもたらします：コンパイラは文字列の特定の部分を使用するかもしれません。

下記の例で試してみましょう。

```
#include <stdio.h>

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

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

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

一般的な C/C++ コンパイラ（MSVCを含む）は2つの文字列を割り当てますが、GCC 4.8.1の動作を見てみましょう。

Listing 1.24: 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
```

確かに、「hello world」という文字列を印刷すると、これらの2つの単語はメモリに隣接して配置され、`f2()` 関数から呼び出される `puts()` 関数はこの文字列が分割されていることを認識しません。実際、分割されていません。このリストには、「仮想的に」分けられています。

`puts()` が `f1()` から呼び出されると、「world」文字列とNULLバイトを使用します。`puts()` はこの文字列の前に何かがあることを認識していません！

この巧妙なトリックは、少なくともGCCでよく使用され、メモリを節約できます。これは文字列インターンに似ています。

別の関連する例がここにあります：**??** on page ??

第1.5.4節ARM

ARMプロセッサを使用した実験では、いくつかのコンパイラを使用しました。

- 組み込みの分野で人気があります：Keilリリース 2013/6
- LLVM-GCC 4.2コンパイラを搭載したApple Xcode 4.6.3 IDE[26]
- GCC 4.9（Linaro）（ARM64用）は`http://go.yurichev.com/17325`で入手可能です。

特に記載がない場合、このマニュアルのすべてのケースで32ビットARMコード（ThumbおよびThumb-2モードを含む）が使用されます。64ビットARMについて話すときは、ARM64と呼びます。

非最適化 **Keil 6/2013 (ARMモード)**

Keilの例をコンパイルすることから始めましょう。

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

armcc コンパイラはIntel構文でアセンブリリストを生成しますが、[27] それには高レベルのARMプロセッサに関連するマクロがありますが、「そのまま」の命令を見ることが重要ですので、IDA のコンパイル結果を見てみましょう。

Listing 1.25: 非最適化 Keil 6/2013 (ARMモード) 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
```

この例では、各命令のサイズが4バイトであることを簡単に確認できます。実際、Thumb用ではなくARMモード用のコードをコンパイルしました。

最初の命令 `STMFD SP!, {R4,LR}`[28]は、2つのレジスタ (R4 と LR) の値をスタックに書き込むx86 PUSH 命令として機能します。

実際、*armcc* コンパイラの出力リストには、簡略化のために実際に `PUSH {r4,lr}` 命令が示されています。しかしそれはかなり正確ではありません。PUSH 命令は、Thumbモードでのみ使用できます。したがって、物事をあまり混乱させないために、私たちは IDA でこれをやっています。

この命令は、最初に SP[30] を decrements して、新しいエントリがないスタック内の場所をポイントし、R4 および LR レジスタの値を変更された SP に格納されたアドレスに保存します。

この命令（Thumbモードの PUSH 命令のような）は、一度にいくつかのレジスタ値を保存することができ、非常に便利です。ところで、これはx86には同等の機能はありません。また、STMFD 命令は、SPだけでなく、どのレジス

---

[26] Apple Xcode 4.6.3は、オープンソースのGCCをフロントエンドコンパイラとLLVMコードジェネレータとして使用しています

[27] 例えば ARMモードには PUSH/POP 命令がありません

[28] STMFD[29]

[30] stack pointer. SP/ESP/RSP in x86/x64. SP in ARM.

タでも動作できるため、PUSH 命令の一般化（機能拡張）であることにも注意してください。換言すれば、STMFD は、指定されたメモリアドレスにレジスタのセットを格納するために使用することもできます。

ADR R0, aHelloWorld 命令は、PC[31]レジスタの値を hello, world 文字列が配置されているオフセットに加算または減算します。ここでPCレジスタはどのように使用されるのですか？これは「位置独立コード」[32]と呼ばれます。

このようなコードは、メモリ内の固定されていないアドレスで実行することができます。換言すれば、これはPC相対アドレッシングである。

ADR 命令は、この命令のアドレスと文字列が配置されているアドレスとの間の差異を考慮する。この違い（オフセット）は、OSによってコードがロードされるアドレスに関係なく、常に同じになります。だから私たちが必要とするのは、現在の命令のアドレス（PCから）を追加して、C文字列の絶対メモリアドレスを取得することだけです。

BL __2printf[33]命令は printf() 関数を呼び出します。この命令の仕組みは次のとおりです。

- BL 命令（0xC）に続くアドレスをLRに格納する
- そのアドレスをPCレジスタに書き込むことによって、コントロールを printf() に渡します。

printf() の実行が終了すると、コントロールを返す必要がある場所に関する情報が必要です。各機能がLRレジスタに格納されたアドレスに制御を渡す理由です。

これはARMのような「純粋な」 RISCプロセッサとx86のようなCISC[34]プロセッサとの違いです。リターンアドレスは通常スタックに格納されます。これについての詳細は、次のセクション (1.7 on page 29) を参照してください。

ところで、絶対32ビットのアドレスまたはオフセットは、24ビットのためのスペースしか有していないので、32ビット BL 命令では符号化することができない。思い出されるように、すべてのARMモード命令は4バイト（32ビット）のサイズを持ちます。したがって、それらは4バイトの境界アドレスにのみ配置することができます。これは、命令アドレスの最後の2ビット（常にゼロビット）が省略されることを意味する。要約すると、オフセットエンコーディングには26ビットがあります。これは $current\_PC \pm \approx 32M$ をエンコードするのに十分です。

次に、MOV R0, #0[35]命令は、R0 レジスタに0を書き込むだけです。これは、C関数が0を返し、戻り値が R0 レジスタに格納されるためです。

最後の命令 LDMFD SP!, R4,PC[36] スタック（または他のメモリ場所）から値をロードして R4 とPCに保存し、stack pointer SPをincrementsします。ここで POP のように動作します。注意：最初の命令 STMFD は R4 とLRレジスタのペアをスタックに保存しましたが、R4 とPCは LDMFD の実行中にリストアされます。

すでにわかっているように、各関数が制御を返さなければならない場所のアドレスは、通常、LRレジスタに保存されます。最初の命令は、printf() を呼び出すときに main() 関数が同じレジスタを使用するため、その値をスタックに保存します。関数の終わりでは、この値を直接PCレジスタに書き込むことができ、したがって関数が呼び出された場所に制御を渡します。

main() は通常 C/C++ の主要な関数なので、コントロールはOSローダーやCRTのような点に返されます。

すべての機能を使用すると、関数の最後に BX LR 命令を省略できます。

DCB は、x86アセンブリ言語のDBディレクティブと同様に、バイトまたはASCII文字列の配列を定義するアセンブリ言語ディレクティブです。

非最適化 **Keil 6/2013 (Thumbモード)**

ThumbモードでKeilを使って同じ例をコンパイルしましょう。

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

IDAに入ってみましょう。

Listing 1.26: 非最適化 Keil 6/2013 (Thumbモード) + 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
```

---

[31] Program Counter. IP/EIP/RIP in x86/64. PC in ARM.

[32] 関連セクションの詳細を読む:(**??** on page ??)

[33] Branch with Link

[34] Complex Instruction Set Computing

[35] MOVeの意味

[36] LDMFD[37]はSTMFDとは逆の命令です

```
.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
```

2バイト（16ビット）のオペコードを簡単に見つけることができます。これは既に述べたように、Thumbです。ただし、BL 命令は2つの16ビット命令で構成されています。これは、1つの16ビットオペコードの小さなスペースを使用している間に `printf()` 関数のオフセットをロードすることが不可能なためです。したがって、第1の16ビット命令はオフセットの上位10ビットをロードし、第2命令はオフセットの下位11ビットをロードする。

前述したように、Thumbモードの命令はすべて2バイト（または16ビット）のサイズです。これは、Thumb命令が奇妙なアドレスにあることはまったく不可能であることを意味します。上記を前提として、命令を符号化する間に最後のアドレスビットを省略することができる。

まとめると、BLThumb命令は $current\_PC \pm \approx 2M$ のアドレスを符号化することができます。

関数内の他の命令については、`PUSH` と `POP` はここで説明した `STMFD`/`LDMFD` のように動作しますが、ここではSPレジスタのみが明示的に言及されていません。`ADR` は前の例と同様に動作します。`MOVS` は、0を返すために `R0` レジスタに0を書き込みます。

最適化 **Xcode 4.6.3 (LLVM) (ARMモード)**

最適化を有効にしない場合のXcode 4.6.3では、冗長なコードが多数生成されるため、命令カウントができるだけ小さい最適化された出力を検討し、コンパイラスイッチ `-O3` を設定します。

Listing 1.27: 最適化 Xcode 4.6.3 (LLVM) (ARMモード)

```
__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
```

命令 `STMFD` と `LDMFD` はもうよく知っていますね。

`MOV` 命令は、`R0` レジスタに数値 `0x1686` を書き込むだけです。これは「Hello world !」文字列を指すオフセットです。

`R7` レジスタ（[*iOS ABI Function Call Guide*, (2010)][38]で標準化されている）はフレームポインタです。以下でもっとみてみましょう。

MOVT R0、＃0（MOVe Top）命令は、レジスタの上位16ビットに0を書き込みます。ここでの問題点は、ARMモードの汎用MOV命令がレジスタの下位16ビットだけを書き込むことができることです。

ARMモードの命令オペコードはすべて32ビットに制限されています。もちろん、この制限はレジスタ間でのデータの移動には関係しません。そのため、上位ビット（16から31まで）に書き込むための命令 `MOVT` が追加されています。ただし、ここでの使用方法は冗長です。これは、`MOV R0, #0x1686` 命令がレジスタの上位部分をクリアしたためです。これはおそらくコンパイラの欠点です。

`ADD R0, PC, R0` 命令は、PCの値を `R0` の値に加算し、「Hello world!」文字列の絶対アドレスを計算します。すでにわかっているように、それは「位置独立コード」なので、この修正はここでは必須です。

`BL` 命令は `printf()` の代わりに `puts()` 関数を呼び出します。

GCCは最初の `printf()` 呼び出しを `puts()` に置き換えました。確かに、唯一の引数を持つ `printf()` は、`puts()` とほぼ同じです。

ほとんどの場合、文字列に% で始まるprintf形式識別子が含まれていない場合にのみ、2つの関数が同じ結果を生成するためです。その場合、これらの2つの機能の効果は異なります [39]

なぜコンパイラは `printf()` を `puts()` に置き換えたのでしょうか？おそらく `puts()` が高速であるためです。[40].

---

[38] 以下で利用可能 http://go.yurichev.com/17276

[39] `puts()` は文字列の最後に改行記号 '\n' を必要としないので、ここでは見られません

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

これは、文字を% と一緒に比較することなく、文字をstdoutに渡すだけです。

次に、R0 レジスタを0に設定するための使い慣れた `MOV R0, #0` 命令があります。

最適化 Xcode 4.6.3 (LLVM) (Thumb-2モード)

Xcode 4.6.3ではThumb-2のコードがデフォルトでは次のように生成されます。

Listing 1.28: 最適化 Xcode 4.6.3 (LLVM) (Thumb-2モード)

```
__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
```

Thumbモードの BL と BLX 命令は、16ビット命令のペアとしてエンコードされています。Thumb-2では、これらの代理オペコードは、新しい命令がここで32ビット命令として符号化されるように拡張される。

これは、Thumb-2命令のオペコードが常に `0xFx` または `0xEx` で始まることを考慮すると明らかです

しかし、IDA のリストでは、opcodeバイトはスワップされます。これは、ARMプロセッサの場合、命令は次のようにエンコードされるためです。最後のバイトが最初に来て、最初のバイトが来ると（ThumbおよびThumb-2モードの場合)、ARMモードの命令の場合、第1、第3、第2、そして最後に第1（異なるエンディアンのため）です。

つまり、バイトがIDAリストにどのように配置されているかです。

- ARMおよびARM64モードの場合：4-3-2-1;
- Thumbモードの場合：2-1;
- Thumb-2モードの16ビット命令の場合は2-1-4-3になります。

したがって、`MOVW`、`MOVT.W` および BLXX命令は `0xFx` で始まります。

Thumb-2命令の1つは `MOVW R0, #0x13D8` です。16ビット値を R0 レジスタの下部に格納し、上位ビットをクリアします。

また、`MOVT.W R0, #0` は、前の例の `MOVT` と同様に動作し、Thumb-2でのみ動作します。

BLX 命令は、BL の代わりにこの場合に使用されます。

違いは、RA[41]をLRレジスタに保存し、`puts()` 関数に制御を渡すことに加えて、プロセッサはThumb/Thumb-2モードからARMモード（またはその逆）にも切り替わります。

この命令は、制御が渡される命令が次のようになっているため、ここに配置されています（ARMモードでエンコードされています)。

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

これは本質的に、importsセクションに `puts()` のアドレスが書き込まれる場所へのジャンプです。

したがって、注意深い読者が質問するかもしれません：コードのどこに必要なところにputs（）を呼び出すのはなぜですか？

非常にスペース効率が良いわけではないからです。

ほぼすべてのプログラムは外部のダイナミックライブラリ（WindowsではDLL、*NIXでは.so、Mac OS X では.dylib）を使用します。動的ライブラリには、標準のC関数puts（）を含む、頻繁に使用されるライブラリ関数が含まれています。

---

[41] リターンアドレス

実行可能バイナリファイル（Windows PE .exe、ELFまたはMach-O）には、インポートセクションが存在します。これは、外部モジュールからインポートされたシンボル（関数またはグローバル変数）のリストと、モジュール自体の名前です。

OSローダは、必要なすべてのモジュールをロードし、プライマリモジュールのインポートシンボルを列挙しながら、各シンボルの正しいアドレスを決定します。

私たちの場合、*__imp__puts* は、OSローダーが外部ライブラリに関数の正しいアドレスを格納するために使用する32ビットの変数です。次に、LDR 命令はこの変数から32ビットの値を読み込み、それを制御に渡してPCレジスタに書き込みます。

したがって、この手順を完了するためにOSローダが必要とする時間を短縮するには、各シンボルのアドレスを専用の場所に1回だけ書き込むことをお勧めします。

さらに、すでにわかっているように、メモリアクセスなしで1つの命令だけを使用している間は、32ビットの値をレジスタにロードすることは不可能です。

したがって、最適な解決策は、ダイナミックライブラリに制御を渡し、次にThumbコードからこの短い1命令関数（いわゆるthunk function）にジャンプするという唯一の目的で、ARMモードで動作する別の関数を割り当てることです。

ところで、（ARMモード用にコンパイルされた）前の例では、コントロールは
TTBL によって同じthunk functionに渡されます。ただし、プロセッサモードは切り替えられていません（したがって、命令ニーモニックに 「X」がありません）。

thunk-functionsの追加情報

サンク関数は、誤った名前のために、明らかに理解するのが難しいです。1つのタイプのジャックのアダプターまたはコンバーターとして別のタイプのジャックに理解する最も簡単な方法です。たとえば、イギリスの電源プラグをアメリカのコンセントに差し込むことができるアダプタ、またはその逆。サンク関数はラッパーと呼ばれることもあります。

これらの関数についてもう少し詳しく説明します：

> 1961年にAlgol-60プロシージャコールの正式な定義に実際のパラメータをバインドする手段としてThunksを発明したP.Z. Ingermanによると、「アドレスを提供するコーディング」: 仮パラメータの代わりに式を使用してプロシージャーを呼び出すと、コンパイラーは式を計算するサンクを生成し、結果のアドレスを何らかの標準の場所に残します。
>
> …
>
> マイクロソフトとIBMは、Intelベースのシステムでは、（ブレティックセグメントレジスタと64Kアドレス制限付きの）「16ビット環境」とフラットアドレッシングとセミリアルメモリ管理を備えた「32ビット環境」を定義しています。この2つの環境は、同じコンピュータとOS上で動作することができます（Microsoftの世界では、Windows on Windowsの略です）。MSとIBMはどちらも、16ビットから32ビットへの変換プロセスを「サンク」と呼んでいます。Windows 95には、THUNK.EXEというツールがあります。これは” サンクコンパイラ” と呼ばれています。

(The Jargon File)

他の例としてLAPACK libraryがあります。FORTRANで書かれた “Linear Algebra PACKage” です。C/C++ 開発者もLAPACKを使いたいと思っていますが、C/C++ に書き直していくつかのバージョンを維持するのは難しいことです。ですから、C/C++ 環境から呼び出し可能な短いC関数があります。これは、順番にFORTRAN関数を呼び出し、他の何かを実行します。

```
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);
}
```

また、そのような関数は “ラッパー” と呼ばれます。

ARM64

GCC

ARM64環境で、GCC 4.8.1を使用してサンプルをコンパイルしましょう。

Listing 1.29: 非最適化 GCC 4.8.1 + 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!..
```

ARM64にはThumbモードとThumb-2モードはなく、ARMのみであるため、32ビット命令のみがあります。レジスタ数は2倍になります：**??** on page ?? 64ビットレジスタは `X-`プレフィックスを持ち、32ビット部分は `W-`です。

STP 命令（ストアペア）は、スタック内の2つのレジスタ `X29` と `X30` を同時に保存します。

もちろん、この命令はメモリ内の任意の場所にこのペアを保存できますが、ここでSPレジスタが指定されているため、ペアはスタックに保存されます。

ARM64レジスタは64ビットのレジスタで、それぞれ8バイトのサイズを持つため、2つのレジスタを保存するために16バイト必要です。

オペランドの後の感嘆符 (“!”) は、最初に16がSPから減算され、次にスタックに書き込まれるレジスタ・ペアの値であることを意味します。これは事前インデックスとも呼ばれます。事後インデックスと事前インデックスの違いについては、**??** on page ?? を読んでください。

したがって、より使い慣れたx86では、最初の命令は `PUSH X29` と `PUSH X30` のペアのアナログに過ぎません。`X29` はARM64ではFP[42]として、LRでは `X30` として使用されているため、関数プロローグに保存され、関数エピローグで復元されます。

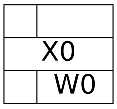
2番目の命令は `X29` （またはFP）のSPをコピーします。これは、関数スタックフレームを設定するために行われます。

ADRP 命令と ADD 命令は、最初の関数引数がこのレジスタに渡されるため、文字列「Hello!」のアドレスを `X0` レジスタに入力するために使用されます。命令長は4バイトに制限されているため、レジスタに多数の命令を格納できる命令はありません。詳細は **??** on page ??参照してください。したがって、いくつかの命令を利用する必要があります。最初の命令（ADRP）は、文字列が配置されている4KiBページのアドレスを `X0` に書き込み、2番目の命令（ADD ）は残りのアドレスをアドレスに追加するだけです。詳細については、**??** on page ??を参照してください。

`0x400000 + 0x648 = 0x400648` であり、このアドレスの`.rodata` データセグメントにある「Hello!」 C文字列を参照してください。

BL 命令を使用して `puts()` を呼び出します。これについては既に説明しました：1.5.4 on page 20

MOV は `W0` に0を書き込みます。`W0` は64ビット `X0` レジスタの下位32ビットです。



関数の結果は `X0` を介して返され、`main()` は0を返します。これで、リターンされる結果がどのように準備されるのかがわかります。しかし、なぜ32ビットの部分を使用するのでしょうか？

ARM64の *int* データ型はx86-64の場合と同じように、互換性を高めるため、32ビットとなっています。

関数が32ビット *int* を返す場合は、`X0` レジスタの下位32ビットのみを埋めなければなりません。

---

[42] Frame Pointer

これを確認するために、この例を少し変更して再コンパイルしましょう。`main()` は64ビット値を返します：

Listing 1.30: `main()` returning a value of `uint64_t` type

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

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

結果は同じですが、その行の `MOV` は次のようになります：

Listing 1.31: 非最適化 GCC 4.8.1 + objdump

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

LDP (*Load Pair*) は `X29` と `X30` レジスタを復元します。

命令の後には感嘆符はありません。これは、値が最初にスタックからロードされ、次にSPが16だけ増加したことを意味します。これは事後インデックスと呼ばれます。

ARM64：RET という新しい命令が登場しました。これは `BX LR` と同様に機能し、特別なヒントビットのみが追加され、これが別のジャンプ命令ではなく関数からの戻りであることをCPUに通知するので、より最適に実行できます。

関数の単純さのために、GCCの最適化はまさに同じコードを生成します。

第1.5.5節MIPS

「グローバルポインタ」について少し

1つの重要なMIPSコンセプトは、「グローバルポインタ」です。既にわかっているように、各MIPS命令のサイズは32ビットなので、32ビットアドレスを1つの命令に組み込むことは不可能ですこの例ではGCCのように対を使用しなければなりません読み込み)。ただし、1つの命令を使用してレジスタ $register-32768...register+32767$ の範囲のアドレスからデータをロードすることは可能です（16ビットの符号付きオフセットを1つの命令でエンコードできるため）。したがって、この目的のためにいくつかのレジスタを割り当てて、最も多く使用されているデータの64KiB領域を割り当てることができます。この割り当てられたレジスタは”グローバルポインタ”と呼ばれ、64KiB領域の中央を指します。この領域には通常、`printf()` のようなインポートされた関数のグローバル変数とアドレスが含まれています。なぜなら、GCCの開発者は、関数のアドレスを得ることは2つではなく1つの命令の実行と同じくらい速くなければならないと判断したからです。ELFファイルでは、この64KiB領域は初期化されていないデータの場合は.sbss（「small BSS[43]」)、初期化されたデータの場合は.sdata（「small data」）のセクションに部分的に配置されています。これはプログラマがどのデータを高速にアクセスしたいのかを選択して.sdata / .sbssに入れることを意味します。いくつかの古い学校のプログラマは、MS-DOSメモリモデル **??** on page ??、またはすべてのメモリが64KiBブロックに分割されたXMS / EMSのようなMS-DOSメモリマネージャ。

この概念はMIPS特有のものではありません。少なくともPowerPCはこの手法も使用しています。

最適化 GCC

グローバルポインタの概念を示す次の例を考えてみましょう。

Listing 1.32: 最適化 GCC 4.4.5 (アセンブリ出力)

```
$LC0:
; \000 is zero byte in octal base:
        .ascii  "Hello, world!\012\000"
main:
; function prologue.
; set the GP:
        lui     $28,%hi(__gnu_local_gp)
        addiu   $sp,$sp,-32
        addiu   $28,$28,%lo(__gnu_local_gp)
; save the RA to the local stack:
```

---

[43] Block Started by Symbol

```
11         sw      $31,28($sp)
12 ; load the address of the puts() function from the GP to $25:
13         lw      $25,%call16(puts)($28)
14 ; load the address of the text string to $4 ($a0):
15         lui     $4,%hi($LC0)
16 ; jump to puts(), saving the return address in the link register:
17         jalr    $25
18         addiu   $4,$4,%lo($LC0) ; branch delay slot
19 ; restore the RA:
20         lw      $31,28($sp)
21 ; copy 0 from $zero to $v0:
22         move    $2,$0
23 ; return by jumping to the RA:
24         j       $31
25 ; function epilogue:
26         addiu   $sp,$sp,32 ; branch delay slot + free local stack
```

我々が見るように、$GPレジスタは関数のプロローグでこの領域の中央を指すように設定されています。RAレジスタもローカルスタックに保存されます。printf() の代わりに puts() もここで使用されます。

puts() 関数のアドレスは、LW 命令（「Load Word」）を使用して $25 にロードされます。 LUI （「Load Upper Immediate」） と ADDIU （「Add Immediate Unsigned Word」）命令のペアを使用して、テキスト文字列のアドレスが $4にロードされます。LUI はレジスタの上位16ビット（したがって「命令名の上位ワード」）を設定し、ADDIU はアドレスの下位16ビットを加算します。

ADDIU は JALR に従います（まだブランチ遅延スロットを覚えていますか？)。レジスタ $4 は $A0 とも呼ばれ、最初の関数引数を渡すために使用されます。[44].

JALR（「Jump and Link Register」）は、RAの次の命令（LW）のアドレスを保存している間、$25 レジスタ（puts() のアドレス）に格納されているアドレスにジャンプします。これはARMと非常によく似ています。ああ、重要なことの1つは、RAに保存されたアドレスは、次の命令のアドレスではないことです。（遅延スロットにあり、ジャンプ命令の前に実行されるため）したがって、$PC+8$ は JALR の実行中にRAに書き込まれます。私たちの場合、これは ADDIU の次の LW 命令のアドレスです。

20行目の LW (「Load Word」) は、ローカルスタックからRAを復元します（この命令は実際には関数のエピローグの一部です）。

22行目の MOVE は、$0（$ZERO）レジスタから $2（$V0）までの値をコピーします。

MIPSは定数レジスタを持ち、常に0を保持します。どうやら、MIPSの開発者たちは、実際にはゼロがコンピュータプログラミングで最も忙しいという考えを思いついたので、ゼロが必要なたびに $0レジスタを使用しましょう。

もう1つの興味深い事実は、MIPSにレジスタ間でデータを転送する命令がないことです。実際、MOVE DST, SRC は ADD DST, SRC, $ZERO ($DST = SRC + 0$) です。これは同じです。明らかに、MIPS開発者はコンパクトなopcodeテーブルを用意したいと考えました。これは、各 MOVE 命令で実際の加算が行われることを意味しません。ほとんどの場合、CPUはこれらの疑似命令を最適化し、ALU[45]は決して使用されません。

24行目の J は、RAのアドレスにジャンプします。これは、関数からの戻り値を効果的に実行しています。J の後の ADDIU は実際に J の前に実行されます（分岐遅延スロットを覚えていますか？)。そして関数のエピローグの一部です。ここに IDA によって生成されたリストもあります。ここの各レジスタには、独自の擬似名があります。

Listing 1.33: 最適化 GCC 4.4.5 (IDA)

```
.text:00000000 main:
.text:00000000
.text:00000000 var_10          = -0x10
.text:00000000 var_4           = -4
.text:00000000
; function prologue.
; set the GP:
.text:00000000                 lui     $gp, (__gnu_local_gp >> 16)
.text:00000004                 addiu   $sp, -0x20
.text:00000008                 la      $gp, (__gnu_local_gp & 0xFFFF)
; save the RA to the local stack:
.text:0000000C                 sw      $ra, 0x20+var_4($sp)
; save the GP to the local stack:
; for some reason, this instruction is missing in the GCC assembly output:
.text:00000010                 sw      $gp, 0x20+var_10($sp)
; load the address of the puts() function from the GP to $t9:
```

---

[44] MIPSレジスタの表はappendixで見られます：**??** on page ??

[45] Japanese text placeholder

```
17 .text:00000014                  lw      $t9, (puts & 0xFFFF)($gp)
18 ; form the address of the text string in $a0:
19 .text:00000018                  lui     $a0, ($LC0 >> 16)  # "Hello, world!"
20 ; jump to puts(), saving the return address in the link register:
21 .text:0000001C                  jalr    $t9
22 .text:00000020                  la      $a0, ($LC0 & 0xFFFF)  # "Hello, world!"
23 ; restore the RA:
24 .text:00000024                  lw      $ra, 0x20+var_4($sp)
25 ; copy 0 from $zero to $v0:
26 .text:00000028                  move    $v0, $zero
27 ; return by jumping to the RA:
28 .text:0000002C                  jr      $ra
29 ; function epilogue:
30 .text:00000030                  addiu   $sp, 0x20
```

15行目の命令は、GPの値をローカルスタックに保存します。この命令は、GCCの出力リストから不思議に見えます.GCCのエラーがあります[46] GPの値は実際に保存しなければなりません。データウィンドウ。`puts()` アドレスを含むレジスタは $T9と呼ばれ、T-が前に付いたレジスタは「一時的」と呼ばれ、その内容は保持されない可能性があるためです。

非最適化 **GCC**

非最適化 GCCはもっと冗長です。

Listing 1.34: 非最適化 GCC 4.4.5 (アセンブリ出力)

```
$LC0:
        .ascii  "Hello, world!\012\000"
main:
; function prologue.
; save the RA ($31) and FP in the stack:
        addiu   $sp,$sp,-32
        sw      $31,28($sp)
        sw      $fp,24($sp)
; set the FP (stack frame pointer):
        move    $fp,$sp
; set the GP:
        lui     $28,%hi(__gnu_local_gp)
        addiu   $28,$28,%lo(__gnu_local_gp)
; load the address of the text string:
        lui     $2,%hi($LC0)
        addiu   $4,$2,%lo($LC0)
; load the address of puts() using the GP:
        lw      $2,%call16(puts)($28)
        nop
; call puts():
        move    $25,$2
        jalr    $25
        nop  ; branch delay slot

; restore the GP from the local stack:
        lw      $28,16($fp)
; set register $2 ($V0) to zero:
        move    $2,$0
; function epilogue.
; restore the SP:
        move    $sp,$fp
; restore the RA:
        lw      $31,28($sp)
; restore the FP:
        lw      $fp,24($sp)
        addiu   $sp,$sp,32
; jump to the RA:
        j       $31
        nop  ; branch delay slot
```

---

[46] 明らかに、リストを生成する関数はGCCユーザーにとってあまり重要ではないので、修正されていないエラーがまだ存在するかもしれません

レジスタFPはスタックフレームへのポインタとして使用されることがわかります。3つのNOPも見てみましょう。2番目と3番目は分岐命令に従います。おそらく、GCCコンパイラは分岐命令の後に常に分岐遅延スロットのためにNOPを追加し、最適化がオンになっていればそれらを削除するかもしれません。したがって、この場合、それらはここに残されます。

IDA のリストもあります：

Listing 1.35: 非最適化 GCC 4.4.5 (IDA)

```
 1 .text:00000000 main:
 2 .text:00000000
 3 .text:00000000 var_10          = -0x10
 4 .text:00000000 var_8           = -8
 5 .text:00000000 var_4           = -4
 6 .text:00000000
 7 ; function prologue.
 8 ; save the RA and FP in the stack:
 9 .text:00000000                 addiu   $sp, -0x20
10 .text:00000004                 sw      $ra, 0x20+var_4($sp)
11 .text:00000008                 sw      $fp, 0x20+var_8($sp)
12 ; set the FP (stack frame pointer):
13 .text:0000000C                 move    $fp, $sp
14 ; set the GP:
15 .text:00000010                 la      $gp, __gnu_local_gp
16 .text:00000018                 sw      $gp, 0x20+var_10($sp)
17 ; load the address of the text string:
18 .text:0000001C                 lui     $v0, (aHelloWorld >> 16)  # "Hello, world!"
19 .text:00000020                 addiu   $a0, $v0, (aHelloWorld & 0xFFFF)  # "Hello, world!"
20 ; load the address of puts() using the GP:
21 .text:00000024                 lw      $v0, (puts & 0xFFFF)($gp)
22 .text:00000028                 or      $at, $zero ; NOP
23 ; call puts():
24 .text:0000002C                 move    $t9, $v0
25 .text:00000030                 jalr    $t9
26 .text:00000034                 or      $at, $zero ; NOP
27 ; restore the GP from local stack:
28 .text:00000038                 lw      $gp, 0x20+var_10($fp)
29 ; set register $2 ($V0) to zero:
30 .text:0000003C                 move    $v0, $zero
31 ; function epilogue.
32 ; restore the SP:
33 .text:00000040                 move    $sp, $fp
34 ; restore the RA:
35 .text:00000044                 lw      $ra, 0x20+var_4($sp)
36 ; restore the FP:
37 .text:00000048                 lw      $fp, 0x20+var_8($sp)
38 .text:0000004C                 addiu   $sp, 0x20
39 ; jump to the RA:
40 .text:00000050                 jr      $ra
41 .text:00000054                 or      $at, $zero ; NOP
```

興味深いことに、IDA は LUI/ADDIU 命令のペアを認識し、15行目の1つの LA （「Load Address」）疑似命令に統合しました。この疑似命令のサイズは8バイトです。これは実際のMIPS命令ではなく、むしろ命令対のための便利な名前であるため、疑似命令（またはマクロ）です。

もう1つのことは、IDA はNOP命令を認識していないことです。22行目、26行目、41行目です。`OR $AT, $ZERO` です。基本的に、この命令は、$AT レジスタの内容にOR演算を0（もちろんアイドル命令）で適用します。MIPSは他の多くのISAと同様に、独立したNOP命令を持っていません。

スタックフレームの役割

テキスト文字列のアドレスはレジスタに渡されます。とにかくローカルスタックをセットアップする理由は？これは、`printf()` が呼び出されるため、レジスタRAとGPの値をどこかに保存する必要があり、ローカルスタックがこの目的のために使用されているという事実にあります。これが leaf function であれば、関数のプロローグとエピローグを取り除くことができました。例：1.4.3 on page 8

最適化 **GCC: GDB**にロードしてみる

Listing 1.36: sample GDB session

```
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)
```

第1.5.6節結論

x86/ARMとx64/ARM64コードの主な違いは、文字列へのポインタが64ビット長になったことです。確かに、現代の CPU は64ビットになりました。これは、現代のアプリケーションではメモリの節約と大きな需要の両方があるからです。私たちは32ビットポインタよりもはるかに多くのメモリをコンピュータに追加することができます。そのため、すべてのポインタは64ビットになりました。

第1.5.7節練習問題

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

第1.6節関数のプロローグとエピローグ

関数プロローグは、関数の先頭にある一連の命令です。それはしばしば以下のコード断片のように見えます。

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

これらの命令が行うこと：EBP レジスタに値を保存し、EBP レジスタの値をESPの値に設定し、ローカル変数のためにスタック上に領域を割り当てます。

EBP の値は、関数実行の期間にわたって同じままであり、ローカル変数および引数アクセスに使用されます。同じ目的のために ESP を使うことができますが、時間の経過とともに変化するので、この方法はあまり便利ではありません。

関数のエピローグは、スタック内の割り当てられた領域を解放し、EBP レジスタの値を初期状態に戻し、制御フローをcallerに返します。

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

関数のプロローグとエピローグは、通常、逆アセンブラで関数の区切りとして検出されます。

第**1.6.1**節再帰

エピローグとプロローグは、再帰のパフォーマンスに悪影響を及ぼします。

この本の再帰の詳細は下記を参照: **??** on page ??

第**1.7**節スタック

スタックは、コンピュータサイエンスにおける最も基本的なデータ構造の1つです。[47]. AKA[48] **LIFO!**[49].

技術的には、それは、プロセスメモリ内のメモリのブロックであり、x86またはx64の ESP または RSP レジスタ、またはARMのSPレジスタをそのブロック内のポインタとして使用します。

最も頻繁に使用されるスタックアクセス命令は、PUSH と POP （x86およびARM Thumbモードの両方）です。PUSH は、32ビットモード（または64ビットモードでは8）で ESP/RSP/SP 4を減算し、その単独オペランドの内容を ESP/RSP/SPが指すメモリアドレスに書き込みます。

POP は逆の操作です：SPが指し示すメモリ位置からデータを取り出し、命令オペランド（しばしばレジスタ）にロードし、stack pointerに4（または8）を追加します。

スタック割り当ての後、stack pointerはスタックの一番下を指します。PUSH はstack pointerを減らし、POP はそれを増やします。スタックの最下部は実際にスタックブロックに割り当てられたメモリの先頭にあります。それは奇妙に見えますが、それはそうです。

ARMは降順スタックと昇順スタックの両方をサポートしています。

例えば、STMFD/LDMFD、STMED[50]/LDMED[51]命令は、降順のスタックを扱うことを意図しています（下位に向かって、高いアドレスから始まり、低いアドレスに進む)。STMFA[52]/LDMFA[53]、STMEA[54]/LDMEA[55]命令は、昇順のスタックを扱うことを意図しています（上位アドレスから始まり、上位アドレスに向かって進みます)。

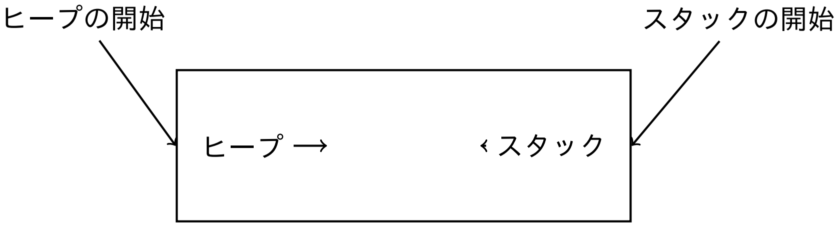
第**1.7.1**節スタックはなぜ後方に進むのか

直感的には、他のデータ構造と同様に、スタックが上方に、すなわちより高いアドレスに向かって成長すると考えるかもしれません。

スタックが後方に成長する理由はおそらく歴史的なものです。コンピュータが大きくて部屋全体を占有していた時代、メモリを2つの部分に分けるのは簡単でした。1つは heap 用、もう1つはスタック用です。もちろん、プログラムの実行中にheapとスタックがどれだけ大きくなるかは不明であったため、この解決策は最も簡単でした。

---

[47] wikipedia.org/wiki/Call_stack
[48] Japanese text placeholder
[49] **LIFO!**
[50] Store Multiple Empty Descending ()
[51] Load Multiple Empty Descending ()
[52] Store Multiple Full Ascending ()
[53] Load Multiple Full Ascending ()
[54] Store Multiple Empty Ascending ()
[55] Load Multiple Empty Ascending ()



[D. M. Ritchie and K. Thompson, *The UNIX Time Sharing System*, (1974)][56]では、以下のように書かれています。

> 画像のユーザコア部分は、3つの論理セグメントに分割される。プログラムテキストセグメントは仮想アドレス空間の位置0で始まります。実行中、このセグメントは書き込み保護されており、同じプログラムを実行しているすべてのプロセス間でこのセグメントが共有されます。仮想アドレス空間のプログラムテキストセグメントの上の最初の8Kバイト境界では、共有されない書き込み可能なデータセグメントが開始されます。このデータセグメントのサイズはシステムコールによって拡張されます。仮想アドレス空間の最上位アドレスから始まるスタックセグメントは、ハードウェアのスタックポインタが変動すると自動的に下に向かって成長します。

これは、一部の学生が1つのノートブックを使用して2つの講義ノートを書く方法を思い出させます。最初の講義のノートはいつものように書かれ、2つ目のノートはノートブックの最後からそれを反転させて書き込まれます。空き領域がない場合に、ノートはその間のどこかで互いに会うことになります。

第**1.7.2**節スタックは何に使用されるか

関数のリターンアドレスを保存する

x86

CALL 命令で別の関数を呼び出すと、CALL 命令の直後のポイントのアドレスがスタックに保存され、CALL オペランドのアドレスへの無条件ジャンプが実行されます。

CALL 命令は、PUSHの `PUSH address_after_call / JMP operand` 命令対に相当する。

RET はスタックから値を取り出し、ジャンプします。これは `POP tmp / JMP tmp` 命令の対に相当します。

スタックのオーバーフローは簡単です。永遠の再帰を実行するだけです：

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

MSVC 2008が問題をレポートします：

```
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
```

…しかし、正しいコードを生成します。

```
?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
```

---

[56] 以下で利用可能 http://go.yurichev.com/17270

```
        ret     0
?f@@YAXXZ ENDP                  ; f
```

…また、コンパイラ最適化（/Ox オプション）を有効にすると、最適化されたコードはスタックをオーバーフローせず、代わりに正しく[57]動作します。

```
?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はどちらの場合も問題の警告を出さずに同様のコードを生成します。

ARM

また、ARMプログラムはスタックを使用してリターンアドレスを保存しますが、別の方法でスタックを使用します。「ハローワールド!」（1.5.4 on page 18) で述べたように、RAはLR（link register）に保存されます。ただし、別の関数を呼び出してもう一度LRレジスタを使用する必要がある場合は、その値を保存する必要があります。通常、関数プロローグに保存されます。

多くの場合、`PUSH R4-R7,LR` のような命令が、エピローグで `POP R4-R7,PC` とともに見られます。したがって、関数で使用されるレジスタ値は、LRを含めてスタックに保存されます。

それにもかかわらず、ある関数が他の関数を呼び出すことがなければ、RISCの用語ではそれを *leaf function*[58]と呼びます。その結果、リーフ関数はLRレジスタを保存しません（LRレジスタを変更しないため）。このような関数が小さく、少数のレジスタを使用する場合は、スタックをまったく使用しないことがあります。したがって、スタックを使用せずにリーフ関数を呼び出すことができます。[59] これは、外部RAMがスタックに使用されないため、古いx86マシンよりも高速になる可能性があります。これは、スタックのメモリがまだ割り当てられていない状況または利用できません。

リーフ関数のいくつかの例：1.10.3 on page 102, 1.10.3 on page 102, **??** on page ??, **??** on page ??, **??** on page ??, **??** on page ??, **??** on page ??, **??** on page ??.

関数の引数を渡す

x86でパラメータを渡す最も一般的な方法は、「cdecl」です。

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

callee関数はスタックポインタを介して引数を取得します。

したがって、`f()` 関数の最初の命令が実行される前に、引数の値がスタックにどのように格納されているかがわかります。

ESP	return address
ESP+4	引数#1, IDA にマークする `arg_0`
ESP+8	引数#2, IDA にマークする `arg_4`
ESP+0xC	引数#3, IDA にマークする `arg_8`
…	…

他の呼び出し規約の詳細については、セクション (**??** on page ??) も参照してください。

ちなみに、callee関数には、渡された引数の数に関する情報はありません。（`printf()` のような）可変数の引数を持つC関数は、フォーマット文字列指定子（% 記号で始まる）を使ってその数を決定します。

私たちが次のように書くとします。

---

[57] ここの皮肉

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

[59] いくつかの時間前、PDP-11とVAXでは、CALL命令（他の関数を呼び出す）は高価でした。実行時間の50%までが費やされる可能性があるため、小さな機能を多数持つことはanti-pattern [Eric S. Raymond, *The Art of UNIX Programming*, (2003)Chapter 4, Part II]

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

`printf()` は1234を出力し、次にそのスタックの隣にある2つの乱数[60]を出力します。

だから、`main()` 関数を宣言する方法はあまり重要ではありません：`main()` `main(int argc, char *argv[])` または `main(int argc, char *argv[], char *envp[])` のいずれかです。

実際、CRTコードは `main()` を以下のように呼び出しています：

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

引数なしで `main()` を `main()` として宣言すると、`main()` はスタックにまだ残っていますが使用されません。`main()` を `main(int argc, char *argv[])` として宣言すると、最初の2つの引数を使用することができ、3つ目の引数は関数の「不可視」のままになります。さらに、`main(int argc)` を宣言することも可能です。これは動作します。

引数を渡す別の方法

プログラマがスタックを介して引数を渡すことは何も必要ではないことは注目に値する。それは要件ではありません。スタックをまったく使用せずに他の方法を実装することもできます。

アセンブリ言語初心者の間でやや普及している方法は、グローバル変数を介して引数を渡すことです

Listing 1.37: Assembly code

```
        ...

        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
```

しかし、このメソッドには明白な欠点があります。*do_something()* 関数は、独自の引数をzapする必要があるため、再帰的に（または別の関数を介して）呼び出すことはできません。ローカル変数を使った同じ話：グローバル変数でそれらを保持すると、関数は自分自身を呼び出すことができませんでした。また、これはスレッドセーフ [61] ではありません。このような情報をスタックに格納する方法は、これをより簡単にします。多くの関数の引数や値、スペースを確保できます。

[Donald E. Knuth, *The Art of Computer Programming*, Volume 1, 3rd ed., (1997), 189] は、IBM System/360上で特に便利な奇妙なスキームについても言及しています。

MS-DOSには、レジスタを介してすべての関数引数を渡す方法がありました。たとえば、古代16ビットMS-DOSの “Hello, world!” コードのコードです。

```
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"
```

---

[60] 厳密な意味でランダムではなく、むしろ予測不可能: 1.7.4 on page 36

[61] 正しく実装され、各スレッドは独自の引数/変数を持つ独自のスタックを持ちます

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

これは、**??** on page ??のメソッドと非常によく似ています。また、Linuxのsyscalls((**??** on page ??))とWindowsを呼び出すのと非常によく似ています。

MS-DOS関数がブール値（すなわち単一ビット、通常はエラー状態を示す）を返す場合、CF フラグがしばしば使用されます。

例えば：

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

エラーの場合、CF フラグが立てられます。それ以外の場合は、新しく作成されたファイルのハンドルが AX を介して返されます。

このメソッドは、アセンブリ言語プログラマによって引き続き使用されます。Windows Research Kernelのソースコード（Windows 2003と非常に似ています）では、次のようなものが見つかります

（ファイル *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
```

ローカル変数記憶域

関数は、スタックの底に向かってstack pointerを減らすだけで、ローカル変数のためにスタックに領域を割り当てることができます。

したがって、どれだけ多くのローカル変数が定義されていても、非常に高速です。スタックにローカル変数を格納する必要もありません。あなたは好きな場所にローカル変数を格納することができますが、伝統的にはこれがどのように行われています。

x86: alloca() 関数

`alloca()` 関数に注目することは重要です[62] この関数は `malloc()` のように動作しますが、スタックに直接メモリを割り当てます。関数のエピローグ (1.6 on page 28) は ESP を初期状態に戻し、割り当てられたメモリは単に破棄されるため、割り当てられたメモリチャンクは `free()` 関数呼び出しで解放する必要はありません。`alloca()` がどのように実装されているかは注目に値する。簡単に言えば、この関数は必要なバイト数だけスタック底部に向かって ESP を下にシフトさせ、割り当てられたブロックへのポインタとして ESP を設定します。

やってみましょう。

```
#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);
};
```

`_snprintf()` 関数は `printf()` と同じように動作しますが、結果をstdout（ターミナルやコンソールなど）にダンプする代わりに、`buf` バッファに書き込みます。`puts()` 関数は `buf` の内容をstdoutにコピーします。もちろん、これらの2つの関数呼び出しは1つの `printf()` 呼び出しで置き換えることができますが、小さなバッファの使用法を説明する必要があります。

MSVC

コンパイルしてみましょう（MSVC 2010で）

Listing 1.38: 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
```

---

[62] MSVCでは、関数の実装は `C:\Program Files (x86)\Microsoft Visual Studio 10.0\VC\crt\src\intel` の `alloca16.asm` と `chkstk.asm` にあります

```
    call    _puts
    add     esp, 28

...
```

`alloca()` の唯一の引数は EAX 経由で（スタックにプッシュするのではなく）渡されます。[63]

GCC + インテル構文

GCC 4.4.1は、外部関数を呼び出すことなく同じことを行います

Listing 1.39: GCC 4.7.3

```
.LC0:
        .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 + AT&T構文

同じコードをAT&T構文で見てみましょう

Listing 1.40: 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
```

---

[63] alloca () はコンパイラ組み込み関数（(**??** on page ??)）ではなく、通常の関数です。MSVC[64]のalloca() の実装には、割り当てられたメモリから読み込むコードが含まれているため、OSが物理メモリをVM領域にマップするために、コード内の命令が数個ではなく別々の関数を必要とする理由の1つです。`alloca()` 呼び出しの後、ESPは600バイトのブロックを指し、`buf` 配列のメモリとして使用できます。

コードは前のリストと同じです。

ちなみに、`movl $3, 20(%esp)` は、Intel構文の `mov DWORD PTR [esp+20], 3` に対応しています。AT&Tの構文では、アドレス指定メモリのレジスタ+オフセット形式は `offset(%register)` のように見えます。

(Windows) SEH

SEH[65] レコードはスタックにも格納されます（存在する場合）。それについてもっと読む：(4.2.1 on page 157)

バッファオーバーフロー保護

詳細はこちら (1.19.2 on page 153)

スタック内のデータの自動解放

おそらく、ローカル変数とSEHレコードをスタックに格納する理由は、スタックポインタを修正するための命令を1つだけ使用して（通常は ADD です）、関数が終了すると自動的に解放されるからです。関数の引数は、関数の終わりに自動的に割り当て解除されます。対照的に、ヒープに格納されているものはすべて明示的に割り当て解除する必要があります。

第**1.7.3**節典型的なスタックレイアウト

最初の命令を実行する前の、関数の開始時の32ビット環境での典型的なスタックレイアウトは次のようになります。

…	…
ESP-0xC	ローカル変数#2, IDA にマークする `var_8`
ESP-8	ローカル変数#1, IDA にマークする `var_4`
ESP-4	saved value ofEBP
ESP	リターンアドレス
ESP+4	引数#1, IDA にマークする `arg_0`
ESP+8	引数#2, IDA にマークする `arg_4`
ESP+0xC	引数#3, IDA にマークする `arg_8`
…	…

第**1.7.4**節スタックのノイズ

> ある人が何かがランダムに見えると言うとき、実際には、その中に何らかの規則性を見ることができないということです
>
> Stephen Wolfram, A New Kind of Science.

多くの場合、この本では「ノイズ」や「ガベージ」の値がスタックやメモリに記述されています。彼らはどこから来たのか？これらは、他の関数の実行後にそこに残っているものです。短い例：

```
#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);
};
```

---

[65] Structured Exception Handling

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

コンパイルすると ...

Listing 1.41: 非最適化 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
```

コンパイラは少し不満そうです...

```
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.
```

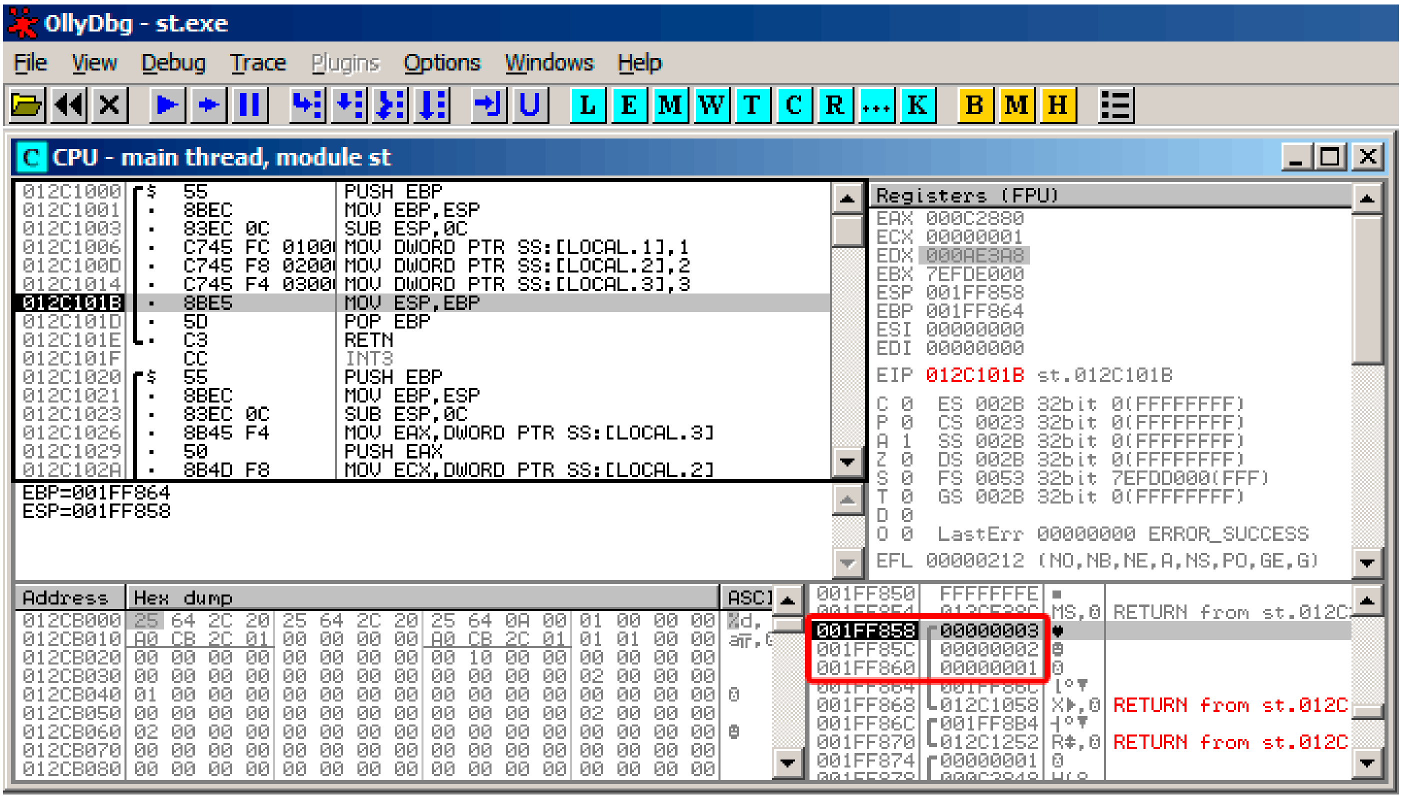
```

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

しかし、コンパイルされたプログラムを実行すると ...

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

ああ、なんて奇妙なんでしょう！我々は `f2()` に変数を設定しませんでした。これらは「ゴースト」値であり、まだスタックに入っています。

サンプルを OllyDbg にロードしましょう。


OllyDbg - st.exe
File View Debug Trace Plugins Options Windows Help
L E M W T C R ... K B M H
CPU - main thread, module st

```
012C1000  $  55           PUSH EBP
012C1001  .  8BEC         MOV EBP,ESP
012C1003  .  83EC 0C      SUB ESP,0C
012C1006  .  C745 FC 0100 MOV DWORD PTR SS:[LOCAL.1],1
012C100D  .  C745 F8 0200 MOV DWORD PTR SS:[LOCAL.2],2
012C1014  .  C745 F4 0300 MOV DWORD PTR SS:[LOCAL.3],3
012C101B  .  8BE5         MOV ESP,EBP
012C101D  .  5D           POP EBP
012C101E  .  C3           RETN
012C101F     CC           INT3
012C1020  $  55           PUSH EBP
012C1021  .  8BEC         MOV EBP,ESP
012C1023  .  83EC 0C      SUB ESP,0C
012C1026  .  8B45 F4      MOV EAX,DWORD PTR SS:[LOCAL.3]
012C1029  .  50           PUSH EAX
012C102A  .  8B4D F8      MOV ECX,DWORD PTR SS:[LOCAL.2]
EBP=001FF864
ESP=001FF858
```

```
Registers (FPU)
EAX 000C2880
ECX 00000001
EDX 000AE3A8
EBX 7EFDE000
ESP 001FF858
EBP 001FF864
ESI 00000000
EDI 00000000

EIP 012C101B st.012C101B

C 0  ES 002B 32bit 0(FFFFFFFF)
P 0  CS 0023 32bit 0(FFFFFFFF)
A 1  SS 002B 32bit 0(FFFFFFFF)
Z 0  DS 002B 32bit 0(FFFFFFFF)
S 0  FS 0053 32bit 7EFDD000(FFF)
T 0  GS 002B 32bit 0(FFFFFFFF)
D 0
O 0  LastErr 00000000 ERROR_SUCCESS

EFL 00000212 (NO,NB,NE,A,NS,PO,GE,G)
```

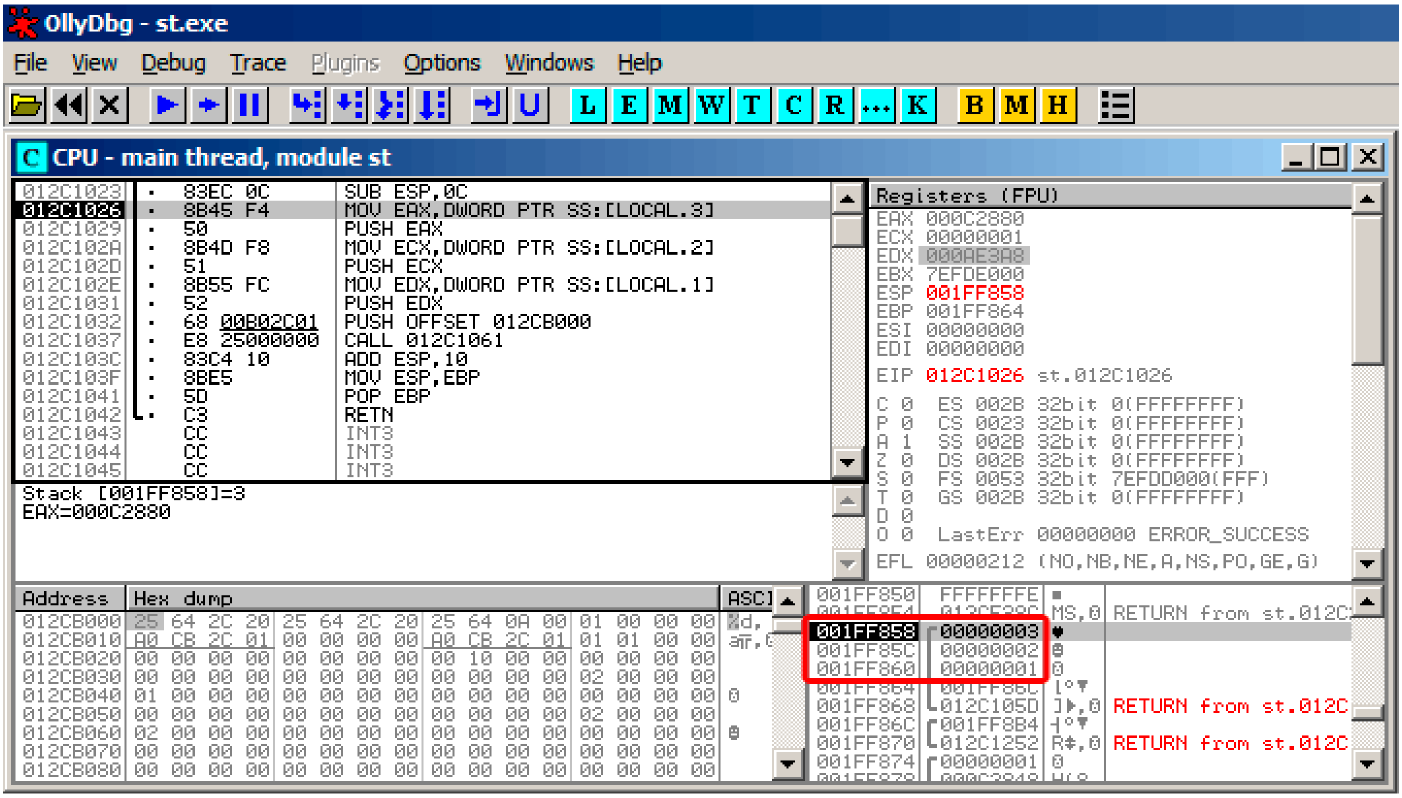
```
Address   Hex dump                                         ASCI
012CB000  25 64 2C 20 25 64 2C 20 25 64 0A 00 01 00 00 00  %d,
012CB010  A0 CB 2C 01 00 00 00 00 A0 CB 2C 01 01 01 00 00
012CB020  00 00 00 00 00 00 00 00 00 10 00 00 00 00 00 00
012CB030  00 00 00 00 00 00 00 00 00 00 00 00 02 00 00 00
012CB040  01 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
012CB050  00 00 00 00 00 00 00 00 00 00 00 00 02 00 00 00
012CB060  02 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
012CB070  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
012CB080  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
```

```
001FF850   FFFFFFFE
                     MS,0  RETURN from st.012C
001FF858  ┌00000003
001FF85C  │00000002
001FF860  │00000001
001FF868  └012C1058  X▸,0  RETURN from st.012C
001FF86C  ┌001FF8B4
001FF870  └012C1252  R‡,0  RETURN from st.012C
001FF874  ┌00000001
```

図 1.5: OllyDbg: f1()

f1() が変数 a、b、c を代入すると、その値はアドレス 0x1FF860 に格納されます。

そして f2() が実行されるとき：


OllyDbg - st.exe
File View Debug Trace Plugins Options Windows Help
L E M W T C R ... K B M H
CPU - main thread, module st

```
012C1023  .  83EC 0C        SUB ESP,0C
012C1026  .  8B45 F4        MOV EAX,DWORD PTR SS:[LOCAL.3]
012C1029  .  50             PUSH EAX
012C102A  .  8B4D F8        MOV ECX,DWORD PTR SS:[LOCAL.2]
012C102D  .  51             PUSH ECX
012C102E  .  8B55 FC        MOV EDX,DWORD PTR SS:[LOCAL.1]
012C1031  .  52             PUSH EDX
012C1032  .  68 00B02C01    PUSH OFFSET 012CB000
012C1037  .  E8 25000000    CALL 012C1061
012C103C  .  83C4 10        ADD ESP,10
012C103F  .  8BE5           MOV ESP,EBP
012C1041  .  5D             POP EBP
012C1042  .  C3             RETN
012C1043     CC             INT3
012C1044     CC             INT3
012C1045     CC             INT3
```

```
Stack [001FF858]=3
EAX=000C2880
```

```
Registers (FPU)
EAX 000C2880
ECX 00000001
EDX 000AE3A8
EBX 7EFDE000
ESP 001FF858
EBP 001FF864
ESI 00000000
EDI 00000000

EIP 012C1026 st.012C1026

C 0  ES 002B 32bit 0(FFFFFFFF)
P 0  CS 0023 32bit 0(FFFFFFFF)
A 1  SS 002B 32bit 0(FFFFFFFF)
Z 0  DS 002B 32bit 0(FFFFFFFF)
S 0  FS 0053 32bit 7EFDD000(FFF)
T 0  GS 002B 32bit 0(FFFFFFFF)
D 0
O 0  LastErr 00000000 ERROR_SUCCESS

EFL 00000212 (NO,NB,NE,A,NS,PO,GE,G)
```

```
Address   Hex dump
012CB000  25 64 2C 20 25 64 2C 20 25 64 0A 00 01 00 00 00
012CB010  A0 CB 2C 01 00 00 00 00 A0 CB 2C 01 01 01 00 00
012CB020  00 00 00 00 00 00 00 00 00 10 00 00 00 00 00 00
012CB030  00 00 00 00 00 00 00 00 00 00 00 00 02 00 00 00
012CB040  01 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
012CB050  00 00 00 00 00 00 00 00 00 00 00 00 02 00 00 00
012CB060  02 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
012CB070  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
012CB080  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
```

```
001FF850  FFFFFFFE
          MS,0  RETURN from st.012C
001FF858  00000003
001FF85C  00000002
001FF860  00000001
001FF864  001FF86C
001FF868  012C105D  RETURN from st.012C
001FF86C  001FF8B4
001FF870  012C1252  RETURN from st.012C
001FF874  00000001
```

図 1.6: OllyDbg: f2()

… f2() の a、b、c は同じアドレスにあります！誰もまだ値を上書きしていないので、その時点でまだ変更はありません。したがって、この奇妙な状況が発生するためには、いくつかの関数を次々と呼び出さなければならず、SPは各関数エントリで同じでなければならない（すなわち、それらは同じ数の引数を有する）。次に、ローカル変数はスタック内の同じ位置に配置されます。要約すると、スタック（およびメモリセル）内のすべての値は、以前の関数実行から残った値を持ちます。彼らは厳密な意味でランダムではなく、むしろ予測不可能な値を持っています。別のオプションがありますか？各関数の実行前にスタックの一部をクリアすることはおそらく可能ですが、余計な（そして不要な）作業です。

MSVC 2013

この例はMSVC 2010によってコンパイルされました。しかし、この本の読者は、このサンプルをMSVC 2013でコンパイルして実行し、3つの数字がすべて逆の結果になるでしょう。

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

どうして？私もMSVC 2013でこの例をコンパイルし、見てみました。

Listing 1.42: 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
```

MSVC 2010とは異なり、MSVC 2013は関数 `f2()` のa/b/c変数を逆順に割り当てました。これは完全に正しい動作です。C/C++ 標準にはルールがありません。ローカル変数をローカルスタックに割り当てる必要があれば、どういう順番でもよいのです。理由の違いは、MSVC 2010にはその方法があり、MSVC 2013はおそらくコンパイラの心臓部で何かが変わったと考えられるからです。

第1.7.5節練習問題

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

第1.8節printf() 引数を取って

さて、ハローワールド! (1.5 on page 8) の例では、`main()` 関数本体の `printf()` を次のように置き換えます。

```
#include <stdio.h>

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

第1.8.1節x86

x86: 3つの引数

MSVC

MSVC 2010 Expressでコンパイルすると、

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

...

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

ほぼ同じですが、`printf()` の引数が逆の順序でスタックにプッシュされるのが分かります。最初の引数は最後にプッシュされます。

ちなみに、32ビット環境での *int* 型の変数は、32ビット幅、つまり4バイトです。

だから、ここでは4つの引数があります。$4 * 4 = 16$。スタック内でちょうど16バイトを占める：文字列への32ビットポインタと *int* 型の3つの数字。

stack pointer（ESP レジスタ）が関数呼び出しの後の `ADD ESP, X` 命令によって元の状態に戻ったとき、関数引数の数はXを4で割るするだけで推測できます。

もちろん、これは *cdecl* 呼び出し規約に固有のものであり、32ビット環境のみに適用されます。

呼び出し規約を参照してください。 (**??** on page ??)

いくつかの関数が互いに直後に戻る特定のケースでは、コンパイラは最後の呼び出しの後に複数の 「ADD ESP, X」命令を1つにマージすることができます：

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

実際の例がここにあります。

Listing 1.43: x86

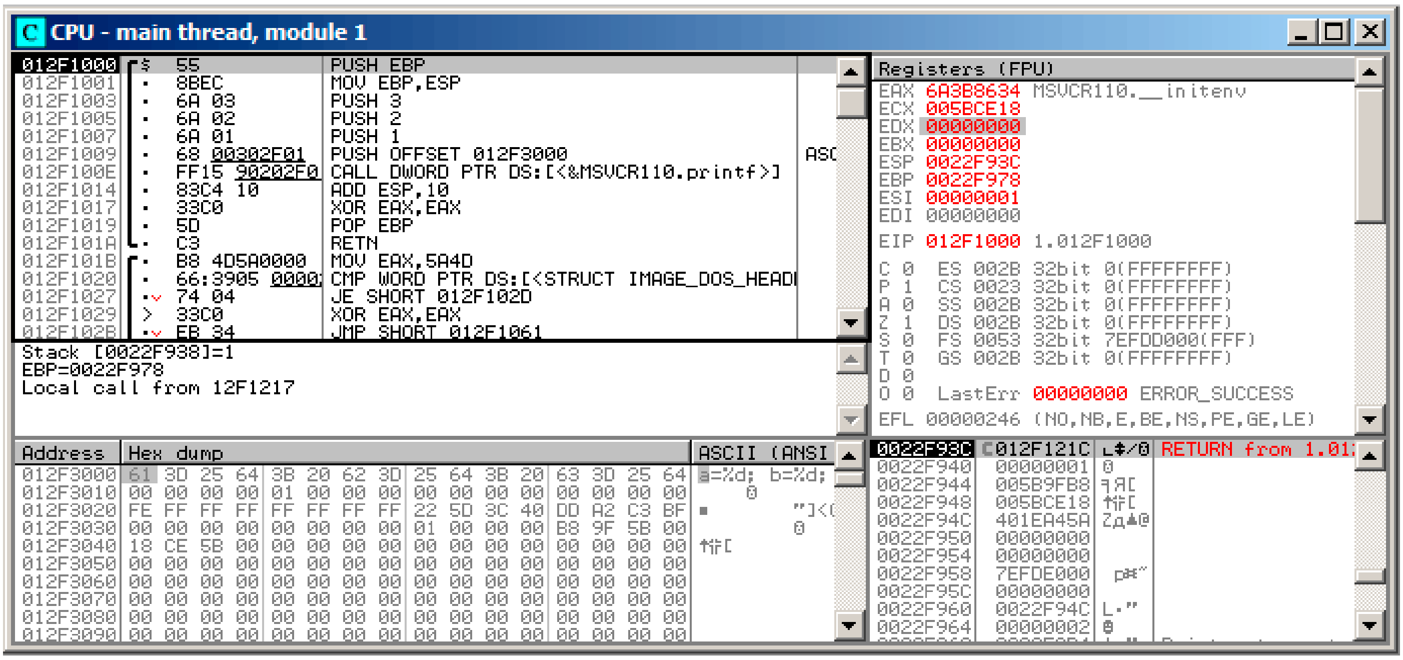
```
.text:100113E7    push    3
.text:100113E9    call    sub_100018B0 ; 引数を1つとる(3)
.text:100113EE    call    sub_100019D0 ; 引数をとらない
.text:100113F3    call    sub_10006A90 ; 引数をとらない
.text:100113F8    push    1
.text:100113FA    call    sub_100018B0 ; 引数を1つとる(1)
.text:100113FF    add     esp, 8       ; 一度にスタックから2つの引数を落とす
```

MSVC と OllyDbg

では、この例を OllyDbg に読み込みましょう。これは最もポピュラーなユーザーランドwin32デバッガーの1つです。MSVC 2012で/MD オプションを使用してサンプルをコンパイルすることができます。これは `MSVCR*.DLL` とリンクすることを意味し、インポートされた関数をデバッガではっきりと見ることができます。

その後、OllyDbg で実行可能ファイルをロードします。最初のブレークポイントは `ntdll.dll` にあり、F9（実行）を押します。2番目のブレークポイントはCRTコードです。`main()` 関数を見つけなければなりません。

コードを一番上までスクロールしてコードを見つけます（MSVCはコードセクションの最初のところで `main()` 関数を割り当てます)。



```
012F1000  55             PUSH EBP
012F1001  8BEC           MOV EBP,ESP
012F1003  6A 03          PUSH 3
012F1005  6A 02          PUSH 2
012F1007  6A 01          PUSH 1
012F1009  68 00302F01    PUSH OFFSET 012F3000                       ASC
012F100E  FF15 90202F0   CALL DWORD PTR DS:[<&MSVCR110.printf>]
012F1014  83C4 10        ADD ESP,10
012F1017  33C0           XOR EAX,EAX
012F1019  5D             POP EBP
012F101A  C3             RETN
012F101B  B8 4D5A0000    MOV EAX,5A4D
012F1020  66:3905 0000;  CMP WORD PTR DS:[<STRUCT IMAGE_DOS_HEADI
012F1027  74 04          JE SHORT 012F102D
012F1029  33C0           XOR EAX,EAX
012F102B  EB 34          JMP SHORT 012F1061
```

```
Stack [0022F938]=1
EBP=0022F978
Local call from 12F1217
```

```
Registers (FPU)
EAX 6A3B8634 MSVCR110.__initenv
ECX 005BCE18
EDX 00000000
EBX 00000000
ESP 0022F93C
EBP 0022F978
ESI 00000001
EDI 00000000

EIP 012F1000 1.012F1000

C 0  ES 002B 32bit 0(FFFFFFFF)
P 1  CS 0023 32bit 0(FFFFFFFF)
A 0  SS 002B 32bit 0(FFFFFFFF)
Z 1  DS 002B 32bit 0(FFFFFFFF)
S 0  FS 0053 32bit 7EFDD000(FFF)
T 0  GS 002B 32bit 0(FFFFFFFF)
D 0
O 0  LastErr 00000000 ERROR_SUCCESS

EFL 00000246 (NO,NB,E,BE,NS,PE,GE,LE)
```

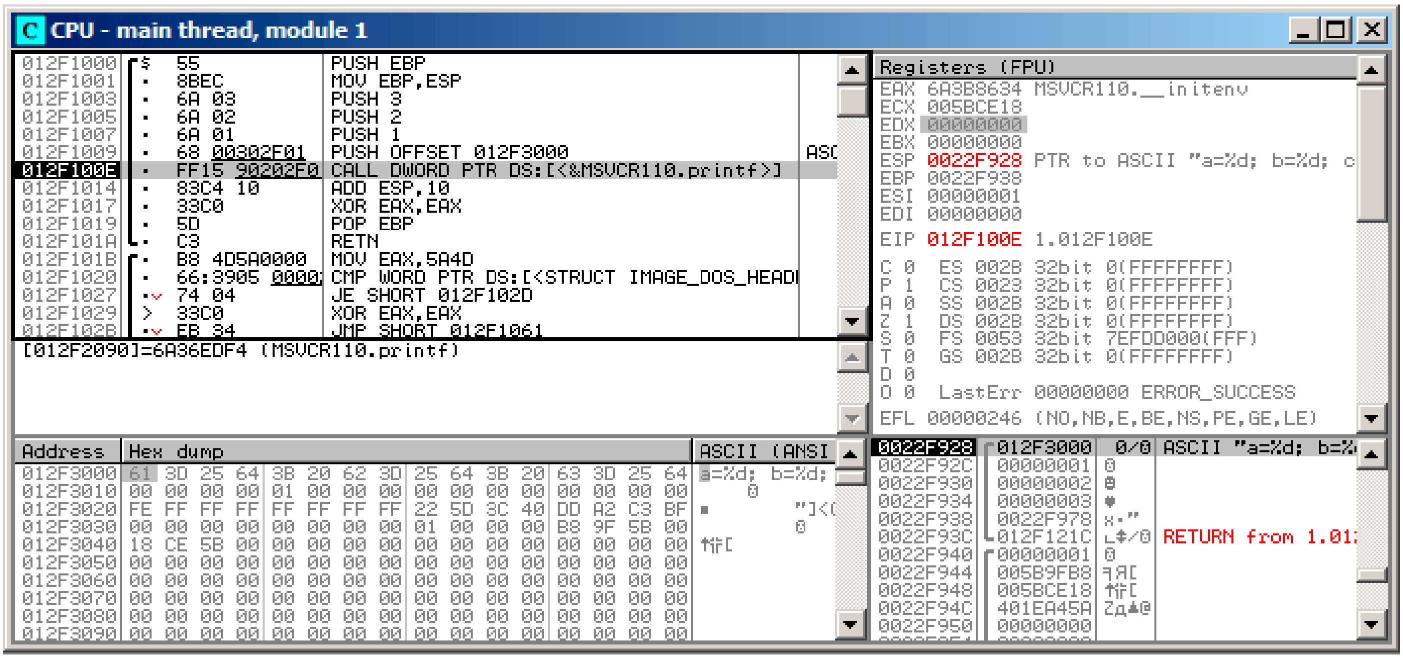
```
Address  Hex dump                                         ASCII (ANSI
012F3000 61 3D 25 64 3B 20 62 3D 25 64 3B 20 63 3D 25 64 a=%d; b=%d;
012F3010 00 00 00 00 01 00 00 00 00 00 00 00 00 00 00 00      0
012F3020 FE FF FF FF FF FF FF FF 22 5D 3C 40 DD A2 C3 BF
012F3030 00 00 00 00 00 00 00 00 01 00 00 00 B8 9F 5B 00          0
012F3040 18 CE 5B 00 00 00 00 00 00 00 00 00 00 00 00 00
012F3050 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
012F3060 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
012F3070 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
012F3080 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
012F3090 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
```

```
0022F93C  012F121C       RETURN from 1.01
0022F940  00000001  0
0022F944  005B9FB8
0022F948  005BCE18
0022F94C  401EA45A
0022F950  00000000
0022F954  00000000
0022F958  7EFDE000
0022F95C  00000000
0022F960  0022F94C  L."
0022F964  00000002
```

図 1.7: OllyDbg: the very start of the `main()` function

`PUSH EBP` 命令をクリックし、F2（ブレークポイントを設定）を押し、F9（実行）を押します。CRTコードをスキップするためには、これらの処理を実行する必要があります。なぜなら、実際にはまだ興味がないからです。

F8（ステップオーバー）を6回押します、つまり6つの命令をスキップします。

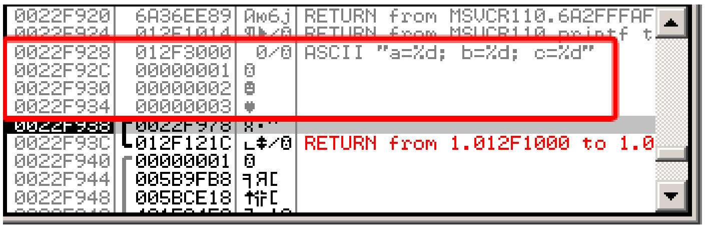


```
CPU - main thread, module 1

012F1000 ┌$  55           PUSH EBP
012F1001 │.  8BEC         MOV EBP,ESP
012F1003 │.  6A 03        PUSH 3
012F1005 │.  6A 02        PUSH 2
012F1007 │.  6A 01        PUSH 1
012F1009 │.  68 00302F01  PUSH OFFSET 012F3000                     ASC
012F100E │.  FF15 90202F0 CALL DWORD PTR DS:[<&MSVCR110.printf>]
012F1014 │.  83C4 10      ADD ESP,10
012F1017 │.  33C0         XOR EAX,EAX
012F1019 │.  5D           POP EBP
012F101A └.  C3           RETN
012F101B ┌.  B8 4D5A0000  MOV EAX,5A4D
012F1020 │.  66:3905 0000 CMP WORD PTR DS:[<STRUCT IMAGE_DOS_HEAD
012F1027 │.v 74 04        JE SHORT 012F102D
012F1029 │>  33C0         XOR EAX,EAX
012F102B │.v EB 34        JMP SHORT 012F1061
[012F2090]=6A36EDF4 (MSVCR110.printf)

Registers (FPU)
EAX 6A3B8634 MSVCR110.__initenv
ECX 005BCE18
EDX 00000000
EBX 00000000
ESP 0022F928 PTR to ASCII "a=%d; b=%d; c
EBP 0022F938
ESI 00000001
EDI 00000000

EIP 012F100E 1.012F100E

C 0  ES 002B 32bit 0(FFFFFFFF)
P 1  CS 0023 32bit 0(FFFFFFFF)
A 0  SS 002B 32bit 0(FFFFFFFF)
Z 1  DS 002B 32bit 0(FFFFFFFF)
S 0  FS 0053 32bit 7EFDD000(FFF)
T 0  GS 002B 32bit 0(FFFFFFFF)
D 0
O 0  LastErr 00000000 ERROR_SUCCESS

EFL 00000246 (NO,NB,E,BE,NS,PE,GE,LE)

Address  Hex dump                                         ASCII (ANSI
012F3000 61 3D 25 64 3B 20 62 3D 25 64 3B 20 63 3D 25 64  a=%d; b=%d;
012F3010 00 00 00 00 01 00 00 00 00 00 00 00 00 00 00 00      ☺
012F3020 FE FF FF FF FF FF FF FF 22 5D 3C 40 DD A2 C3 BF  ■       "]<(
012F3030 00 00 00 00 00 00 00 00 01 00 00 00 B8 9F 5B 00          ☺
012F3040 18 CE 5B 00 00 00 00 00 00 00 00 00 00 00 00 00  ↑╬[
012F3050 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
012F3060 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
012F3070 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
012F3080 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
012F3090 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00

0022F928 ┌012F3000  0/0 ASCII "a=%d; b=%
0022F92C │00000001 ☺
0022F930 │00000002 ☻
0022F934 │00000003 ♥
0022F938 │0022F978 x•"
0022F93C └012F121C └‡/0 RETURN from 1.01
0022F940 ┌00000001 ☺
0022F944 │005B9FB8 ╕Я[
0022F948 │005BCE18 ↑╬[
0022F94C │401EA45A Zд▲@
0022F950 │00000000
```

図 1.8: OllyDbg: before `printf()` execution

これで、PCは `CALL printf` 命令を指し示します。OllyDbg は他のデバッガと同様に、変更されたレジスタの値を強調表示します。したがって、F8を押すたびに EIP が変化し、その値が赤で表示されます。引数の値がスタックにプッシュされるため、ESP も変更されます。

スタック内の値はどこにありますか？右下のデバッガーウィンドウを見てみましょう：



```
0022F920  6A36EE89 Aw6j RETURN from MSVCR110.6A2FFFAF
0022F924  012F1014 ¶▶/0 RETURN from MSVCR110.printf t
0022F928  012F3000  0/0 ASCII "a=%d; b=%d; c=%d"
0022F92C  00000001 ☺
0022F930  00000002 ☻
0022F934  00000003 ♥
0022F938 ┌0022F978 x•"
0022F93C └012F121C └‡/0 RETURN from 1.012F1000 to 1.0
0022F940 ┌00000001 ☺
0022F944 │005B9FB8 ╕Я[
0022F948 │005BCE18 ↑╬[
```

図 1.9: OllyDbg ：引数の値がプッシュされた後のスタック（赤い長方形の枠線はグラフィックエディタで作者によって追加されました）

スタック内のアドレス、スタック内の値、および追加の OllyDbg コメントが3つあります。OllyDbg は `printf()` のような文字列を理解しているので、ここに文字列とそれに付随する 3つの値を報告します。

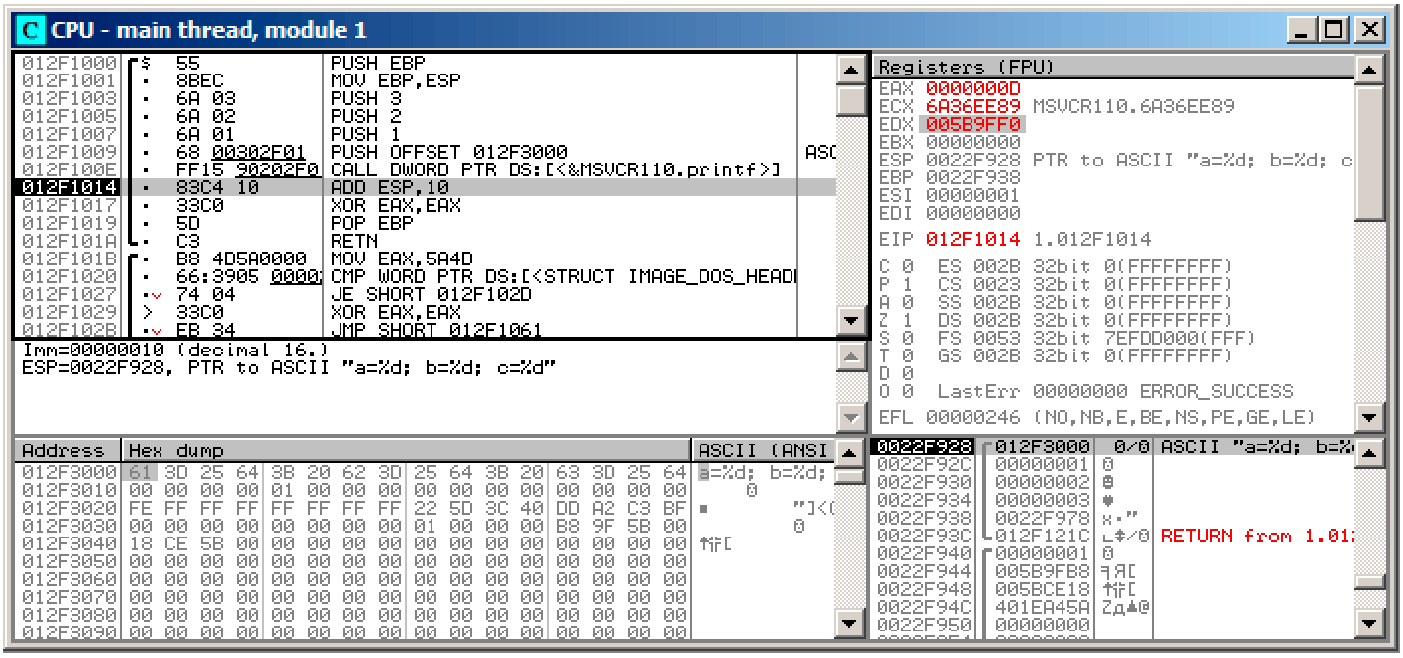
フォーマット文字列を右クリックし、「Follow in dump」をクリックすると、フォーマット文字列がデバッガの左下のウィンドウに表示され、メモリの一部が常に表示されます。これらのメモリ値は編集できます。書式文字列を変更することができます。この場合、例の結果は異なります。この特殊なケースではそれほど有用ではありませんが、エクササイズとしてはいいかもしれません。

F8キーを押します（ステップオーバー）。

コンソールに次の出力が表示されます。

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

レジスタとスタックの状態がどのように変化したかを見てみましょう。



```
012F1000 ┌$  55          PUSH EBP
012F1001 │.  8BEC        MOV EBP,ESP
012F1003 │.  6A 03       PUSH 3
012F1005 │.  6A 02       PUSH 2
012F1007 │.  6A 01       PUSH 1
012F1009 │.  68 00302F01 PUSH OFFSET 012F3000                          ASC
012F100E │.  FF15 90202F0 CALL DWORD PTR DS:[<&MSVCR110.printf>]
012F1014 │.  83C4 10     ADD ESP,10
012F1017 │.  33C0        XOR EAX,EAX
012F1019 │.  5D          POP EBP
012F101A └.  C3          RETN
012F101B ┌.  B8 4D5A0000 MOV EAX,5A4D
012F1020 │.  66:3905 0000 CMP WORD PTR DS:[<STRUCT IMAGE_DOS_HEAD
012F1027 │.v 74 04       JE SHORT 012F102D
012F1029 │>  33C0        XOR EAX,EAX
012F102B │.v EB 34       JMP SHORT 012F1061
```

```
Imm=00000010 (decimal 16.)
ESP=0022F928, PTR to ASCII "a=%d; b=%d; c=%d"
```

```
Registers (FPU)
EAX 0000000D
ECX 6A36EE89 MSVCR110.6A36EE89
EDX 005B9FF0
EBX 00000000
ESP 0022F928 PTR to ASCII "a=%d; b=%d; c
EBP 0022F938
ESI 00000001
EDI 00000000

EIP 012F1014 1.012F1014

C 0  ES 002B 32bit 0(FFFFFFFF)
P 1  CS 0023 32bit 0(FFFFFFFF)
A 0  SS 002B 32bit 0(FFFFFFFF)
Z 1  DS 002B 32bit 0(FFFFFFFF)
S 0  FS 0053 32bit 7EFDD000(FFF)
T 0  GS 002B 32bit 0(FFFFFFFF)
D 0
O 0  LastErr 00000000 ERROR_SUCCESS

EFL 00000246 (NO,NB,E,BE,NS,PE,GE,LE)
```

```
Address  Hex dump                                         ASCII (ANSI
012F3000 61 3D 25 64 3B 20 62 3D 25 64 3B 20 63 3D 25 64 a=%d; b=%d;
012F3010 00 00 00 00 01 00 00 00 00 00 00 00 00 00 00 00     0
012F3020 FE FF FF FF FF FF FF FF 22 5D 3C 40 DD A2 C3 BF ■       "]<(
012F3030 00 00 00 00 00 00 00 00 01 00 00 00 B8 9F 5B 00         0
012F3040 18 CE 5B 00 00 00 00 00 00 00 00 00 00 00 00 00 ↑╬[
012F3050 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
012F3060 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
012F3070 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
012F3080 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
012F3090 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
```

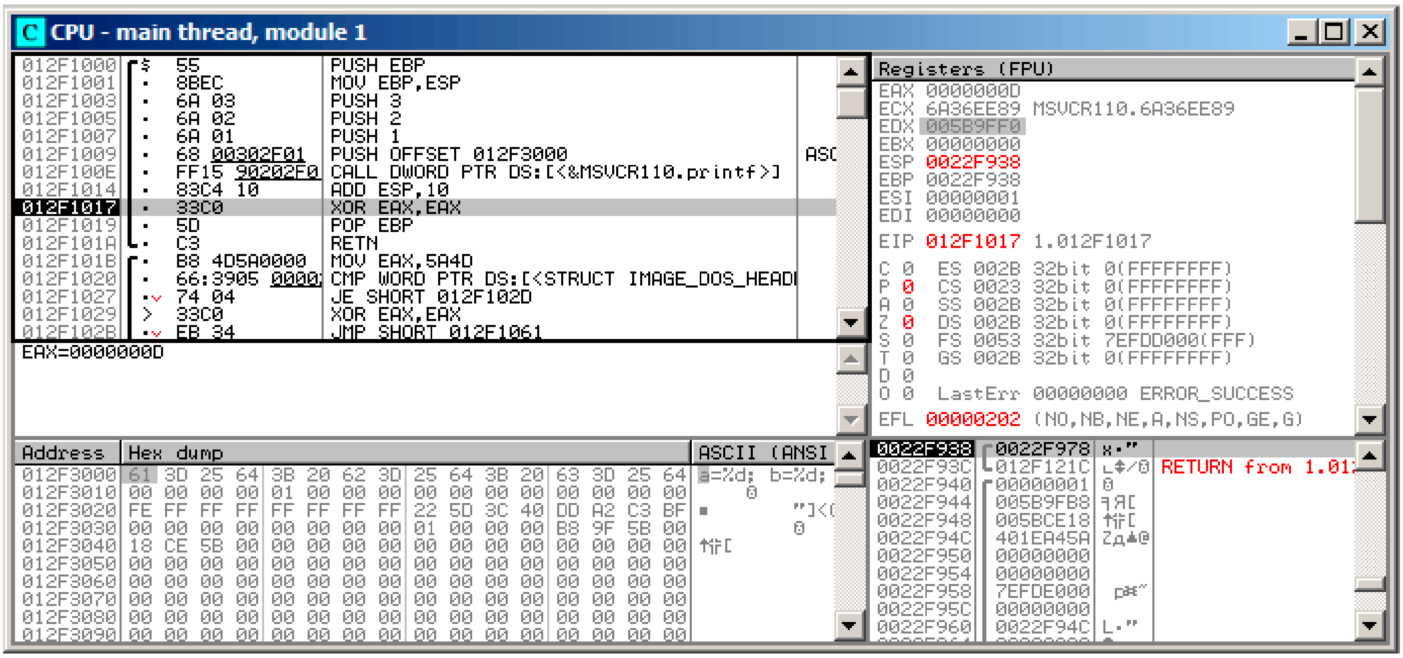
```
0022F928 ┌012F3000  0/0 ASCII "a=%d; b=%
0022F92C │00000001 0
0022F930 │00000002 ☻
0022F934 │00000003 ♥
0022F938 │0022F978 x•"
0022F93C └012F121C ∟‡/0 RETURN from 1.01:
0022F940 ┌00000001 0
0022F944 │005B9FB8 ╕Я[
0022F948 │005BCE18 ↑╬[
0022F94C │401EA45A Zд▲@
0022F950 │00000000
```

図 1.10: printf() 実行後の OllyDbg

レジスタ EAX に `0xD` (13) が含まれるようになりました。printf() は印刷された文字数を返すので正しい。EIP の値は変更されました。実際、CALL printf の後に来る命令のアドレスを含んでいます。ECX と EDX の値も変更されています。明らかに、printf() 関数の隠れた機構は、自身の必要のため、それらを使用しました。

非常に重要な事実は、ESP 値もスタック状態も変更されていないことです！フォーマット文字列とそれに対応する3つの値がまだ存在することがわかります。実際には、これは *cdecl* 呼び出し規約の動作です：calleeは ESP を以前の値に戻しません。callerはこれを行う責任があります。

F8をもう一度押して ADD ESP, 10 命令を実行します。



```
012F1000  $   55           PUSH EBP
012F1001  .   8BEC         MOV EBP,ESP
012F1003  .   6A 03        PUSH 3
012F1005  .   6A 02        PUSH 2
012F1007  .   6A 01        PUSH 1
012F1009  .   68 00302F01  PUSH OFFSET 012F3000                          ASC
012F100E  .   FF15 90202F0 CALL DWORD PTR DS:[<&MSVCR110.printf>]
012F1014  .   83C4 10      ADD ESP,10
012F1017  .   33C0         XOR EAX,EAX
012F1019  .   5D           POP EBP
012F101A  .   C3           RETN
012F101B  .   B8 4D5A0000  MOV EAX,5A4D
012F1020  .   66:3905 0000 CMP WORD PTR DS:[<STRUCT IMAGE_DOS_HEADI
012F1027  .v  74 04        JE SHORT 012F102D
012F1029  >   33C0         XOR EAX,EAX
012F102B  .v  EB 34        JMP SHORT 012F1061
EAX=0000000D
```

```
EAX 0000000D
ECX 6A36EE89 MSVCR110.6A36EE89
EDX 005B9FF0
EBX 00000000
ESP 0022F938
EBP 0022F938
ESI 00000001
EDI 00000000

EIP 012F1017 1.012F1017

C 0  ES 002B 32bit 0(FFFFFFFF)
P 0  CS 0023 32bit 0(FFFFFFFF)
A 0  SS 002B 32bit 0(FFFFFFFF)
Z 0  DS 002B 32bit 0(FFFFFFFF)
S 0  FS 0053 32bit 7EFDD000(FFF)
T 0  GS 002B 32bit 0(FFFFFFFF)
D 0
O 0  LastErr 00000000 ERROR_SUCCESS

EFL 00000202 (NO,NB,NE,A,NS,PO,GE,G)
```

```
Address  Hex dump                                         ASCII (ANSI
012F3000 61 3D 25 64 3B 20 62 3D 25 64 3B 20 63 3D 25 64 a=%d; b=%d;
012F3010 00 00 00 00 01 00 00 00 00 00 00 00 00 00 00 00
012F3020 FE FF FF FF FF FF FF FF 22 5D 3C 40 DD A2 C3 BF
012F3030 00 00 00 00 00 00 00 00 01 00 00 00 B8 9F 5B 00
012F3040 18 CE 5B 00 00 00 00 00 00 00 00 00 00 00 00 00
012F3050 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
012F3060 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
012F3070 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
012F3080 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
012F3090 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
```

```
0022F938  0022F978
0022F93C  012F121C       RETURN from 1.01
0022F940  00000001
0022F944  005B9FB8
0022F948  005BCE18
0022F94C  401EA45A
0022F950  00000000
0022F954  00000000
0022F958  7EFDE000
0022F95C  00000000
0022F960  0022F94C
```

図 1.11: ADD ESP, 10 命令実行後の OllyDbg

ESP は変更されましたが、値はまだスタックにあります！はい、もちろん; これらの値をゼロなどに設定する必要はありません。スタックポインタ（SP）の上にあるものはすべてノイズやガーベッジであり、まったく意味がありません。とにかく、未使用のスタックエントリをクリアするのに時間がかかり、誰も本当に必要とはしません。

GCC

GCC 4.4.1を使って同じプログラムをLinuxでコンパイルして、IDA で何が得られたかを見てみましょう。

```
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
```

MSVCコードとGCCコードの違いは、引数がスタックに格納される方法だけにあることに注目してください。ここでGCCは PUSH/POP を使用せずに、スタックを直接使用してスタックを操作しています

GCC and GDB

LinuxのGDB[66]でもこの例を試してみましょう。

---

[66] GNU Debugger

-g オプションは、デバッグ情報を実行可能ファイルに含めるようにコンパイラに指示します。

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

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

Listing 1.44: `printf()` にブレークポイントを設定しましょう

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

実行します。ここには `printf()` 関数のソースコードがありませんので、GDBはそれを表示することはできませんが、実行することはできます。

```
(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.
```

スタック要素を10個表示します。最も左の列には、スタック上のアドレスが含まれています。

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

最初の要素はRA（`0x0804844a`）です。このアドレスのメモリを逆アセンブルして確認することができます：

```
(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
```

2つの XCHG 命令は、NOPに類似したアイドル命令です。

2番目の要素（`0x080484f0`）はフォーマット文字列アドレスです。

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

次の3つの要素（1,2,3）は `printf()` の引数です。残りの要素はスタック上の「garbage」に過ぎないかもしれませんが、他の関数やローカル変数などからの値であってもかまいません。

「finish」を実行します。このコマンドは、GDBに「関数の最後まですべての命令を実行する」よう指示します。この場合、`printf()` の最後まで実行してください。

```
(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は、`printf()` が EAX（13）をリターンしたことを示しています。これは、OllyDbg の例のように、印刷される文字の数です。

また、「return 0;」と、この式が6行目の `1.c` ファイルにあるという情報も表示されます。実際には、`1.c` ファイルは現在のディレクトリにあり、GDBはその文字列を見つけます。どのCコード行が現在実行されているかをGDBはどうやってに知るのでしょうか？これは、コンパイラがデバッグ情報を生成している間に、ソースコードの行番号と命令アドレスの間の関係のテーブルも保存しているためです。GDBはソースレベルのデバッガです。

レジスタを調べてみましょう。EAXには13が入っています

```
(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>
...
```

現在の命令を逆アセンブルしましょう。矢印は、次に実行される命令を指し示します。

```
(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はデフォルトでAT&T構文を使用します。しかし、インテルの構文に切り替えることは可能です：

```
(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.
```

次の命令を実行します。GDBは、終わりの括弧を示します。つまり、ブロックを終了します。

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

MOV EAX, 0 命令実行後のレジスタを調べてみましょう。実際、EAX はその時点で0です。

```
(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つの引数

他の引数がスタックを介してどのように渡されているかを知るために、引数の数を9（`printf()` の書式文字列+8の *int* 変数）に増やして、この例を再度変更してみましょう。

```
#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

前に述べたように、最初の4つの引数は、Win64の `RCX` 、`RDX` 、`R8` 、`R8` レジスタに渡されなければなりません。それはまさに私たちがここに見るものです。ただし、`PUSH` ではなく `MOV` 命令がスタックの準備に使用されるため、値は直接的にスタックに格納されます。

Listing 1.45: 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

        ; 0をリターン
        xor     eax, eax

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

注意深い読者は、4が十分であるときになぜ *int* 値のために8バイトが割り振られているのか尋ねるかもしれません。はい、思い出す必要があります：64バイトより短い任意のデータ型に対して8バイトが割り当てられます。これは便宜上確立されています。任意の引数のアドレスを簡単に計算できます。また、それらはすべて整列したメモリアドレスに配置されています。32ビット環境でも同じです。すべてのデータ型に4バイトが予約されています。

GCC

最初の6つの引数が `RDI` 、`RSI` 、`RDX` 、`RCX` 、`R8` 、`R9` レジスタを通過する点を除いて、画像はx86-64 *NIX OSの場合と似ています。残りすべてはスタックを介して。GCCは、`RDI` の代わりに `EDI` に文字列ポインタを格納するコードを生成しています。これまでは：1.5.2 on page 15

また、以前は `printf()` 呼び出しの前に `EAX` レジスタがクリアされていることに注意してください：1.5.2 on page 15

Listing 1.46: 最適化 GCC 4.4.6 x64

```
.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 ; ベクトルレジスタの数を渡す
        mov     DWORD PTR [rsp+16], 8
        mov     DWORD PTR [rsp+8], 7
        mov     DWORD PTR [rsp], 6
        call    printf

        ; 0をリターン

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

GCC + GDB

GDBでこの例を試してみましょう

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

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

Listing 1.47: ブレークポイントを `printf()` に設定して実行しましょう

```
(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.
```

Registers RSI/RDX/RCX/R8/R9 have the expected values. RIP has the address of the very first instruction of the `printf()` function.

レジスタ RSI/RDX/RCX/R8/R9 は期待値を持っています。RIP には、`printf()` 関数の最初の命令のアドレスがあります。

```
(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 1.48: let’s inspect the format string

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

今度はx/gコマンドでスタックをダンプしましょう。*g* は、*giant words*、つまり64ビットの単語を表します。

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

最初のスタック要素は、前の例と同様、RAです。3の値もスタックに通されます：6,7,8。また、クリアされていない32ビットの8がが渡されたことがわかります：`0x00007fff00000008`。値は *int* 型（32ビット）なので、それは問題ありません。したがって、上位レジスタまたはスタック要素の部分には「ランダムなゴミ」が含まれている可能性があります。

`printf()` の実行後にコントロールが返る場所を見れば、GDBは `main()` 関数全体を表示します：

```
(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.
```

`printf()` の実行を終了し、EAX をゼロにする命令を実行し、EAX レジスタが正確にゼロの値を持つことに注意してください。RIP は、LEAVE 命令、すなわち `main()` 関数の最後から2番目の命令を指すようになりました。

```
(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>
...
```

第1.8.2節ARM

ARM: 3つの引数

引数を渡すためのARMの伝統的なスキーム（呼び出し規約）は、次のように動作します。最初の4つの引数は R0-R3 レジスタに渡されます。残りの引数はスタックを介して。これはfastcall (**??** on page ??) またはwin64 (**??** on page ??) の引数渡しスキームに似ています。

32ビットARM

非最適化 Keil 6/2013 (ARMモード)

Listing 1.49: 非最適化 Keil 6/2013 (ARMモード)

```
.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           ; return 0
.text:0000001C 10 80 BD E8   LDMFD   SP!, {R4,PC}
```

したがって、最初の4つの引数は、R0-R3 レジスタをこの順序で渡します。`R0` の `printf()` 形式文字列へのポインタ、R1 の1、R2 の2、R3 の3の順です。`0x18` の命令は `R0` に0を書き込みます。これは *return 0* となるCの命令文です。珍しいことは何もありません。

最適化 Keil 6/2013 は同じコードを生成します。

最適化 Keil 6/2013 (Thumbモード)

Listing 1.50: 最適化 Keil 6/2013 (Thumbモード)

```
.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}
```

ARMモードの最適化されていないコードとの大きな違いはありません。

最適化 Keil 6/2013 (ARMモード) + returnを削除してみる

return 0 を取り除いて例を少し修正しましょう：

```
#include <stdio.h>

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

結果はちょっと珍しくなりました。

Listing 1.51: 最適化 Keil 6/2013 (ARMモード)

```
.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
```

これはARMモード用に最適化された（`-O3`）バージョンであり、今回は `B` を使い慣れた `BL` ではなく最後の命令と見なします。この最適化されたバージョンと前のバージョン（最適化なしでコンパイルされたもの）との別の違いは、関数のプロローグとエピローグ（`R0` と LRレジスタの値を保持する命令）の欠如です。`B` 命令は、x86の `JMP` と同様に、LRレジスタの操作なしで別のアドレスにジャンプするだけです。それはなぜ機能するのでしょうか？実際、このコードは以前のコードと事実上同等です。主な理由は2つあります。1）スタックもSP（stack pointer）も変更されていません。2）`printf()` の呼び出しが最後の命令なので、その後は何も起こりません。完了すると、`printf()` 関数はLRに格納されているアドレスにコントロールを返します。LRは現在、関数が呼び出されたポイントのアドレスを格納しているので、`printf()` からの制御はそのポイントに返されます。したがって、LRを変更する必要がないため、LRを節約する必要はありません。`printf()` 以外の関数呼び出しがないため、LRを変更する必要はありません。さらに、この呼び出しの後、私たちは何もしません！これがそのような最適化が可能な理由です。

この最適化は、最後のステートメントが別の関数の呼び出しである関数でよく使用されます。同様の例をここに示します：**??** on page ??

ARM64

非最適化 **GCC (Linaro) 4.9**

Listing 1.52: 非最適化 GCC (Linaro) 4.9

```
.LC1:
        .string "a=%d; b=%d; c=%d"
f2:
; スタックフレームにFPとLRを保存する:
        stp     x29, x30, [sp, -16]!
; スタックフレームを設定する(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
; FP と LRをリストアする
        ldp     x29, x30, [sp], 16
        ret
```

第1の命令 STP (ストアペア) は、FP（X29）およびLR（X30）をスタックに保存します。2番目の `ADD X29, SP, 0` 命令がスタックフレームを形成します。SPの値をX29に書き込むだけです。

次に、使い慣れた ADRP/ADD 命令ペアを参照します。これは、文字列へのポインタを形成します。すなわち、*lo12* は、LC1アドレスの下位12ビットを ADD 命令のオペコードに書き込みます。`printf()` の文字列書式の%d は32ビット *int* なので、1,2および3は32ビットのレジスタ部分にロードされます。

最適化 GCC (Linaro) 4.9 は同じコードを生成します。

ARM: 8つの引数

前のセクションの9つの引数を使って例を再利用してみましょう：1.8.1 on page 49

```
#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;
};
```

最適化 Keil 6/2013: ARMモード

```
.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
```

このコードはいくつかの部分に分けることができます：

- 関数プロローグ:

  最初の `STR LR, [SP,#var_4]!` 命令は、このレジスタを `printf()` 呼び出しに使用する予定であるため、LRをスタックに保存します。最後の感嘆符は、事前索引を示します。

  これは、まずSPを4減少させた後、SPに格納されたアドレスにLRを保存することを意味します。これはx86のPUSHに似ています。もっと読む：**??** on page ??

  第2の `SUB SP, SP, #0x14` 命令は、スタック上に0x14（20）バイトを割り当てるためにSP（stack pointer）を減少させる。実際には、スタックを介して5つの32ビット値を `printf()` 関数に渡さなければならず、それぞれが4バイトを占めます。これは正確に $5 * 4 = 20$ です。他の4つの32ビット値は、レジスタに通される。

- スタックを介して5,6,7および8を渡す：それらはそれぞれ R0, R1, R2 と R3 レジスタに格納される。
  次に、`ADD R12, SP, #0x18+var_14` 命令は、これら4つの変数が格納されるスタックアドレスを R12 レジスタに書き込みます。*var_14* は、スタックにアクセスするコードを便利に表示するために IDA によって作成された-0x14に等しいアセンブリマクロです。IDA によって生成される *var_?* マクロは、スタック内のローカル変数を反映します。

  したがって、SP+4 は R12 レジスタに格納されます。
  次の `STMIA R12, R0-R3` 命令は、レジスタ R0-R3 の内容を R12 が指すメモリに書き込みます。STMIA は、後に複数のインクリメントを格納します。「*Increment After*」は、各レジスタ値が書き込まれた後に R12 が4ずつ増加することを意味する。

- スタックを介して4を渡す：4が R0 に格納され、`STR R0, [SP,#0x18+var_18]` 命令の助けを借りてこの値がスタックに保存されます。*var_18* は-0x18なので、オフセットは0になります。したがって、R0 レジスタ（4）の値はSPに書き込まれたアドレスに書き込まれます。

- レジスタ経由で1,2,3を渡す最初の3つの数値（a、b、c）（それぞれ1,2,3）の値は、printf () 呼び出しの直前に R1 、R2 、R3 レジスタに渡されます。他の5つの値はスタックを介して渡されます。
- `printf()` 呼び出し。
- 関数エピローグ：

  `ADD SP, SP, #0x14` 命令はSPポインタを元の値に戻して、スタックに格納されているものをすべて取り消します。もちろん、スタックに格納されているものはそこにとどまりますが、後続の関数の実行中にすべて書き換えられます。

  `LDR PC, [SP+4+var_4],#4` 命令は、保存されたLR値をスタックからPCレジスタにロードして、機能を終了させます。感嘆符はありません。次に、SPは ($4 + var\_4 = 4 + (-4) = 0$ に格納されているアドレスから最初にロードされるため、この命令は `LDR PC, [SP],#4` に似ています)、SPは4だけ増加します。これはポストインデックス[67]と呼ばれます。なぜ IDA はそのような指示を表示するのですか？なぜなら、スタックレイアウトと、var_4 がローカルスタックのLR値を保存するために割り当てられているという事実を説明したいからです。この命令は、x86の `POP PC` と多少似ています。[68]

最適化 **Keil 6/2013: Thumb**モード

```
.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}
```

出力は前の例とほぼ同じです。ただし、これはThumbコードであり、値はスタックに別々にパックされます。8が先に進み、次に5,6,7、および4が3番目に進みます。

最適化 **Xcode 4.6.3 (LLVM): ARM**モード

```
__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
```

---

[67] もっと読む：**??** on page ??

[68] x86では POP を使って IP/EIP/RIP の値を設定することは不可能ですが、アナロジーとしてはよいでしょう

```
__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}
```

STMFA （Store Multiple Increment Before）命令の同義語である STMIB （Store Multiple Full Ascending）命令を除き、既に見てきたものとほぼ同じです。この命令は、SPレジスタの値を増加させ、逆の順序でこれら2つの動作を実行するのではなく、次のレジスタ値をメモリに書き込むだけです。

目を引くもう一つのことは、命令が一見無作為に配置されていることです。例えば、R0レジスタの値は、アドレス 0x2918, 0x2920 and 0x2928 の3箇所で操作できます。

しかしながら、最適化コンパイラは、実行中により高い効率を達成するために、命令をどのように順序付けするかに関する独自の理由を有することができる。

通常、プロセッサは、並んで配置された命令を同時に実行しようと試みます。たとえば、MOVT R0, #0 、ADD R0, PC, R0 などの命令は、両方とも R0 レジスタを変更するため、同時に実行することはできません。一方、MOVT R0, #0 、MOV R2, #4 命令は、実行の影響が互いに矛盾しないため、同時に実行することができます。おそらく、コンパイラはそのような方法でコードを生成しようと試みます（どこでも可能です）。

最適化 **Xcode 4.6.3 (LLVM): Thumb-2**モード

```
__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}
```

Thumb命令が代わりに使用される点を除いて、出力は前の例とほぼ同じです。

ARM64

非最適化 GCC (Linaro) 4.9

Listing 1.53: 非最適化 GCC (Linaro) 4.9

```
.LC2:
        .string "a=%d; b=%d; c=%d; d=%d; e=%d; f=%d; g=%d; h=%d\n"
f3:
; スタックにスペースをあける:
        sub     sp, sp, #32
; スタックフレームにFPとLRを保存する:
        stp     x29, x30, [sp,16]
; スタックフレームを設定する(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           ; 9th argument
        str     w1, [sp]        ; store 9th argument in the stack
        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
; FPとLRとリストアする
        ldp     x29, x30, [sp,16]
        add     sp, sp, 32
        ret
```

最初の8つの引数は、XレジスタまたはWレジスタに渡されます。[*Procedure Call Standard for the ARM 64-bit Architecture (AArch64)*, (2013)][69]文字列ポインタは64ビットのレジスタを必要とするため、`X0` で渡されます。それ以外の値はすべてint型32ビット型なので、レジスタ（W-）の32ビット部分に格納されます。第9引数（8）はスタックを介して渡されます。実際には、レジスタの数が限られているため、多数の引数をレジスタに渡すことはできません。

最適化 GCC (Linaro) 4.9 は同じコードを生成します。

第1.8.3節MIPS

3 arguments

最適化 GCC 4.4.5

「ハローワールド!」 の例との主な違いは、`puts()` の代わりに `printf()` が呼び出され、さらに3つの引数がレジスタ $5…$7 (または $A0…$A2) に渡されるという点です。そのため、これらのレジスタの前にA-が付いています。これは、関数引数の受け渡しに使用されることを意味します。

Listing 1.54: 最適化 GCC 4.4.5 (アセンブリ出力)

```
$LC0:
        .ascii  "a=%d; b=%d; c=%d\000"
main:
; 関数プロローグ:
        lui     $28,%hi(__gnu_local_gp)
        addiu   $sp,$sp,-32
        addiu   $28,$28,%lo(__gnu_local_gp)
        sw      $31,28($sp)
; printf()のアドレスをロードする:
        lw      $25,%call16(printf)($28)
; テキスト文字列のアドレスをロードし、printf()の1番目の引数を設定する:
        lui     $4,%hi($LC0)
        addiu   $4,$4,%lo($LC0)
; printf()の2番目の引数を設定する:
```

---

[69] 以下で利用可能 http://go.yurichev.com/17287

```
        li      $5,1                    # 0x1
; printf()の3番目の引数を設定する:
        li      $6,2                    # 0x2
; printf()をコールする:
        jalr    $25
; printf()の4番目の引数を設定する (分岐遅延スロット):
        li      $7,3                    # 0x3

; 関数エピローグ:
        lw      $31,28($sp)
; 戻り値に0を設定する:
        move    $2,$0
; リターン
        j       $31
        addiu   $sp,$sp,32 ; branch delay slot
```

Listing 1.55: 最適化 GCC 4.4.5 (IDA)

```
.text:00000000 main:
.text:00000000
.text:00000000 var_10          = -0x10
.text:00000000 var_4           = -4
.text:00000000
; 関数プロローグ:
.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)
; printf()のアドレスをロードする:
.text:00000014                 lw      $t9, (printf & 0xFFFF)($gp)
; テキスト文字列のアドレスをロードし、printf()の1番目の引数を設定する:
.text:00000018                 la      $a0, $LC0        # "a=%d; b=%d; c=%d"
; printf()の2番目の引数を設定する:
.text:00000020                 li      $a1, 1
; printf()の3番目の引数を設定する:
.text:00000024                 li      $a2, 2
; printf()をコールする:
.text:00000028                 jalr    $t9
; printf()の4番目の引数を設定する (分岐遅延スロット):
.text:0000002C                 li      $a3, 3
; 関数エピローグ:
.text:00000030                 lw      $ra, 0x20+var_4($sp)
; 戻り値に0を設定する:
.text:00000034                 move    $v0, $zero
; リターン
.text:00000038                 jr      $ra
.text:0000003C                 addiu   $sp, 0x20 ; branch delay slot
```

IDA は、LUI および ADDIU 命令のペアを1つの LA 疑似命令に統合しました。だから、アドレス0x1Cに命令がないのは、LA が8バイトを占有しているからです。

非最適化 **GCC 4.4.5**

非最適化 GCC はもっと冗長です。

Listing 1.56: 非最適化 GCC 4.4.5 (アセンブリ出力)

```
$LC0:
        .ascii  "a=%d; b=%d; c=%d\000"
main:
; 関数プロローグ:
        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)
```

```
; テキスト文字列のアドレスをロードする:
        lui     $2,%hi($LC0)
        addiu   $2,$2,%lo($LC0)
; printf()の1番目の引数を設定する:
        move    $4,$2
; printf()の2番目の引数を設定する:
        li      $5,1                    # 0x1
; printf()の3番目の引数を設定する:
        li      $6,2                    # 0x2
; printf()の4番目の引数を設定する:
        li      $7,3                    # 0x3
; printf()のアドレスを取得する:
        lw      $2,%call16(printf)($28)
        nop
; printf()をコールする:
        move    $25,$2
        jalr    $25
        nop

; 関数エピローグ:
        lw      $28,16($fp)
; 戻り値に0を設定する:
        move    $2,$0
        move    $sp,$fp
        lw      $31,28($sp)
        lw      $fp,24($sp)
        addiu   $sp,$sp,32
; リターン
        j       $31
        nop
```

Listing 1.57: 非最適化 GCC 4.4.5 (IDA)

```
.text:00000000 main:
.text:00000000
.text:00000000 var_10          = -0x10
.text:00000000 var_8           = -8
.text:00000000 var_4           = -4
.text:00000000
; 関数プロローグ:
.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)
; テキスト文字列のアドレスをロード:
.text:0000001C                 la      $v0, aADBDCD     # "a=%d; b=%d; c=%d"
; printf()の1番目の引数を設定:
.text:00000024                 move    $a0, $v0
; printf()の2番目の引数を設定:
.text:00000028                 li      $a1, 1
; printf()の3番目の引数を設定:
.text:0000002C                 li      $a2, 2
; printf()の4番目の引数を設定:
.text:00000030                 li      $a3, 3
; printf()のアドレスを得る:
.text:00000034                 lw      $v0, (printf & 0xFFFF)($gp)
.text:00000038                 or      $at, $zero
; printf()をコールする:
.text:0000003C                 move    $t9, $v0
.text:00000040                 jalr    $t9
.text:00000044                 or      $at, $zero ; NOP
; 関数エピローグ:
.text:00000048                 lw      $gp, 0x20+var_10($fp)
; 戻り値に0を設定する:
.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
; リターン
.text:00000060                  jr      $ra
.text:00000064                  or      $at, $zero ; NOP
```

8つの引数

前のセクションの9つの引数を使用して、例を再度使用してみましょう：1.8.1 on page 49

```
#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;
};
```

最適化 **GCC 4.4.5**

最初の4つの引数だけが $A0 ...$A3 レジスタに渡され、残りはスタックを介して渡されます。

これはO32呼び出し規約（MIPS世界で最も一般的なもの）です。他の呼び出し規則（N32のような）は、異なる目的のためにレジスタを使用するかもしれません。

SW は「Store Word」（レジスタからメモリへ）の略語です。MIPSには値をメモリに格納する命令がないため、代わりに命令ペア（LI/SW ）を使用する必要があります。

Listing 1.58: 最適化 GCC 4.4.5 (アセンブリ出力)

```
$LC0:
        .ascii  "a=%d; b=%d; c=%d; d=%d; e=%d; f=%d; g=%d; h=%d\012\000"
main:
; 関数プロローグ:
        lui     $28,%hi(__gnu_local_gp)
        addiu   $sp,$sp,-56
        addiu   $28,$28,%lo(__gnu_local_gp)
        sw      $31,52($sp)
; スタックに5番目の引数を渡す:
        li      $2,4                    # 0x4
        sw      $2,16($sp)
; スタックに6番目の引数を渡す:
        li      $2,5                    # 0x5
        sw      $2,20($sp)
; スタックに7番目の引数を渡す:
        li      $2,6                    # 0x6
        sw      $2,24($sp)
; スタックに8番目の引数を渡す:
        li      $2,7                    # 0x7
        lw      $25,%call16(printf)($28)
        sw      $2,28($sp)
; $a0に1番目の引数を渡す:
        lui     $4,%hi($LC0)
; スタックに9番目の引数を渡す:
        li      $2,8                    # 0x8
        sw      $2,32($sp)
        addiu   $4,$4,%lo($LC0)
; $a1に2番目の引数を渡す:
        li      $5,1                    # 0x1
; $a2に3番目の引数を渡す:
        li      $6,2                    # 0x2
; printf()をコールする:
        jalr    $25
; $a3に4番目の引数を渡す (分岐遅延スロット):
        li      $7,3                    # 0x3

; 関数エピローグ:
        lw      $31,52($sp)
; 戻り値に0を設定する:
        move    $2,$0
; リターン
        j       $31
        addiu   $sp,$sp,56 ; branch delay slot
```

Listing 1.59: 最適化 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_4           = -4
.text:00000000
; 関数プロローグ:
.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)
; スタックに5番目の引数を渡す:
.text:00000014                 li      $v0, 4
.text:00000018                 sw      $v0, 0x38+var_28($sp)
; スタックに6番目の引数を渡す:
.text:0000001C                 li      $v0, 5
.text:00000020                 sw      $v0, 0x38+var_24($sp)
; スタックに7番目の引数を渡す:
.text:00000024                 li      $v0, 6
.text:00000028                 sw      $v0, 0x38+var_20($sp)
; スタックに8番目の引数を渡す:
.text:0000002C                 li      $v0, 7
.text:00000030                 lw      $t9, (printf & 0xFFFF)($gp)
.text:00000034                 sw      $v0, 0x38+var_1C($sp)
; $a0に1番目の引数を準備する:
.text:00000038                 lui     $a0, ($LC0 >> 16)  # "a=%d; b=%d; c=%d; d=%d; e=%d; f=%d↙
    ↪ ; g=%"...
; スタックに9番目の引数を渡す:
.text:0000003C                 li      $v0, 8
.text:00000040                 sw      $v0, 0x38+var_18($sp)
; $a0に1番目の引数を渡す:
.text:00000044                 la      $a0, ($LC0 & 0xFFFF)  # "a=%d; b=%d; c=%d; d=%d; e=%d; f↙
    ↪ =%d; g=%"...
; $a1に2番目の引数を渡す:
.text:00000048                 li      $a1, 1
; $a2に3番目の引数を渡す:
.text:0000004C                 li      $a2, 2
; printf()をコールする:
.text:00000050                 jalr    $t9
; $a3に4番目の引数を渡すpass 4th argument in $a3 (分岐遅延スロット):
.text:00000054                 li      $a3, 3
; 関数エピローグ:
.text:00000058                 lw      $ra, 0x38+var_4($sp)
; 戻り値に0を設定する:
.text:0000005C                 move    $v0, $zero
; リターン
.text:00000060                 jr      $ra
.text:00000064                 addiu   $sp, 0x38 ; branch delay slot
```

非最適化 GCC 4.4.5

非最適化 GCC はもっと冗長です。

Listing 1.60: 非最適化 GCC 4.4.5 (アセンブリ出力)

```
$LC0:
        .ascii  "a=%d; b=%d; c=%d; d=%d; e=%d; f=%d; g=%d; h=%d\012\000"
main:
; 関数プロローグ:
        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)
; スタックに5番目の引数を渡す:
        li      $3,4                    # 0x4
        sw      $3,16($sp)
; スタックに6番目の引数を渡す:
        li      $3,5                    # 0x5
        sw      $3,20($sp)
; スタックに7番目の引数を渡す:
        li      $3,6                    # 0x6
        sw      $3,24($sp)
; スタックに8番目の引数を渡す:
        li      $3,7                    # 0x7
        sw      $3,28($sp)
; スタックに9番目の引数を渡す:
        li      $3,8                    # 0x8
        sw      $3,32($sp)
; $a0に1番目の引数を渡す:
        move    $4,$2
; $a1に2番目の引数を渡す:
        li      $5,1                    # 0x1
; $a2に3番目の引数を渡す:
        li      $6,2                    # 0x2
; $a3に4番目の引数を渡す:
        li      $7,3                    # 0x3
; printf()をコールする:
        lw      $2,%call16(printf)($28)
        nop
        move    $25,$2
        jalr    $25
        nop
; 関数エピローグ:
        lw      $28,40($fp)
; 戻り値に0を設定する:
        move    $2,$0
        move    $sp,$fp
        lw      $31,52($sp)
        lw      $fp,48($sp)
        addiu   $sp,$sp,56
; リターン
        j       $31
        nop
```

Listing 1.61: 非最適化 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
; 関数プロローグ:
.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=%"...
; スタックに5番目の引数を渡す:
.text:00000024                 li      $v1, 4
.text:00000028                 sw      $v1, 0x38+var_28($sp)
; スタックに6番目の引数を渡す:
.text:0000002C                 li      $v1, 5
.text:00000030                 sw      $v1, 0x38+var_24($sp)
; スタックに7番目の引数を渡す:
.text:00000034                 li      $v1, 6
.text:00000038                 sw      $v1, 0x38+var_20($sp)
; スタックに8番目の引数を渡す:
.text:0000003C                 li      $v1, 7
.text:00000040                 sw      $v1, 0x38+var_1C($sp)
; スタックに9番目の引数を渡す:
.text:00000044                 li      $v1, 8
.text:00000048                 sw      $v1, 0x38+var_18($sp)
; $a0に1番目の引数を渡す:
.text:0000004C                 move    $a0, $v0
; $a1に2番目の引数を渡す:
.text:00000050                 li      $a1, 1
; $a2に3番目の引数を渡す:
.text:00000054                 li      $a2, 2
; $a3に4番目の引数を渡す:
.text:00000058                 li      $a3, 3
; printf()をコールする:
.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
; 関数エピローグ:
.text:00000070                 lw      $gp, 0x38+var_10($fp)
; 戻り値に0を設定する:
.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
; リターン
.text:00000088                 jr      $ra
.text:0000008C                 or      $at, $zero ; NOP
```

第**1.8.4**節結論

関数呼び出しの概略を以下に示します。

Listing 1.62: x86

```
...
PUSH 3rd argument
PUSH 2nd argument
PUSH 1st argument
CALL function
; スタックポインタを修正する（必要なら）
```

Listing 1.63: x64 (MSVC)

```
MOV RCX, 1st argument
MOV RDX, 2nd argument
MOV R8, 3rd argument
MOV R9, 4th argument
...
PUSH 5th, 6th argument, etc. (if needed)
```

```
CALL function
; スタックポインタを修正する（必要なら）
```

Listing 1.64: 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 7th, 8th argument, etc. (if needed)
CALL function
; スタックポインタを修正する（必要なら）
```

Listing 1.65: ARM

```
MOV R0, 1st argument
MOV R1, 2nd argument
MOV R2, 3rd argument
MOV R3, 4th argument
; 5番目、6番目の引数などをスタックに渡す（必要なら）
BL function
; スタックポインタを修正する（必要なら）
```

Listing 1.66: 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
; 9番目、10番目の引数などをスタックに渡す（必要なら）
BL function
; スタックポインタを修正する（必要なら）
```

Listing 1.67: 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
; 5番目、6番目の引数などをスタックに渡す（必要なら）
LW temp_reg, address of function
JALR temp_reg
```

第**1.8.5**節ところで

ところで、x86、x64、fastcall、ARM、およびMIPSで渡される引数の違いは、CPUが引数を関数にどのように引き渡すかを知らないという事実の良い例です。スタックをまったく使用せずに引数を特殊な構造体に渡すことができる仮想コンパイラを作成することもできます。

MIPS $A0 ...$A3 レジスタは、便宜上（O32呼び出し規約にある）このようにラベル付けされています。プログラマは、データを渡すために、あるいは他の呼び出し規約を使用するために、他のレジスタ（おそらく $ZERO を除く）を使用することができます。

CPUは呼び出し規約を認識していません。

他の関数に引数を渡す新しいアセンブリ言語プログラマが、通常はレジスタ経由で、明示的な順序なしに、あるいはグローバル変数を介して、どのように新しい関数を呼び出すかを思い出すかもしれません。もちろん、それは正常に動作します。

第1.9節scanf()

では、scanf() を使ってみましょう。

第1.9.1節Simple example

```
#include <stdio.h>

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

        scanf ("%d", &x);

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

        return 0;
};
```

最近、ユーザーとのやり取りに `scanf()` を使用するのは賢明ではありません。しかし、*int* 型の変数にポインタを渡す例で説明することができます。

ポインタについて

ポインタはコンピュータサイエンスの基本概念の1つです。多くの場合、大規模な配列、構造体またはオブジェクトを引数として別の関数に渡すことはコストがかかりすぎ、アドレスを渡すことはずっと安いです。たとえば、コンソールにテキスト文字列を印刷する場合、そのアドレスをOSカーネルに渡す方がはるかに簡単です。

さらに、callee関数が大きな配列または構造体の中の何かをパラメータとして受け取って構造体全体を返す必要がある場合、状況は不条理に近いです。したがって、配列や構造体のアドレスを呼び出し先関数に渡し、変更する必要のあるものを変更させるのが最も簡単です。

C/C++ のポインタは、単純にあるメモリ位置のアドレスです。

x86では、アドレスは32ビット数（すなわち、4バイトを占める）で表され、x86-64では64ビット数（8バイトを占める）です。ところで、それがx86-64への切り替えに関連する憤慨の裏にある理由は、x64アーキテクチャのポインタは、「高価な」メモリであるキャッシュメモリを含めて、2倍のスペースを必要とします。

何らかの努力があれば、型の指定されていないポインタでのみ作業することができます。例えば1つのメモリ位置から別のメモリ位置にブロックをコピーする標準のC関数 `memcpy()` は、コピーするデータの型を予測することが不可能なため、`void*` 型の2つのポインタを引数としてとります。データ型は重要ではなく、ブロックサイズだけが重要です。

ポインタは、関数が複数の値を返す必要がある場合にも広く使用されます。（これについては後で説明します: (1.12 on page 108)）

`scanf()` 関数が、このような場合です。

関数が正常に読み取られた値の数を示す必要があるという事実に加えて、これらの値もすべて返す必要があります。

C/C++ では、ポインタ型はコンパイル時の型チェックにのみ必要です。

内部的には、コンパイルされたコードには、ポインタ型に関する情報はまったくありません。

x86

MSVC

MSVC 2010でコンパイルした後に得られるものは次のとおりです。

```
CONST    SEGMENT
$SG3831    DB    'Enter X:', 0aH, 00H
$SG3832    DB    '%d', 00H
$SG3833    DB    'You entered %d...', 0aH, 00H
CONST    ENDS
PUBLIC    _main
EXTRN    _scanf:PROC
EXTRN    _printf:PROC
; 関数のコンパイルフラグ: /Odtp
_TEXT    SEGMENT
_x$ = -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

    ; 0をリターン
    xor    eax, eax
    mov    esp, ebp
    pop    ebp
    ret    0
_main    ENDP
_TEXT    ENDS
```

x はローカル変数です。

C/C++ 標準によれば、この関数でのみ表示でき、他の外部スコープでは表示できません。従来、ローカル変数はスタックに格納されていました。それらを割り当てる方法はおそらく他にもありますが、それはx86の方法です。

関数プロローグ、`PUSH ECX` に続く命令の目的は、ECX 状態を保存することではありません（関数の最後に対応する `POP ECX` が存在しないことに注意してください)。

実際、x 変数を格納するためにスタックに4バイトを割り当てます。

x は、_x$ マクロ (-4に等しい) と現在のフレームを指す EBP レジスタの助けを借りてアクセスされます。

関数の実行の範囲にわたって、EBP は現在のstack frameを指しており、EBP+オフセットを介してローカル変数と関数引数にアクセスすることができます。

同じ目的で ESP を使用することもできますが、ESP は頻繁に変更されるためあまり便利ではありません。EBP の値は、関数の実行開始時に ESP の値が固定された状態として認識される可能性があります。

32ビット環境での典型的なstack frameレイアウトを次に示します。

…	…
EBP-8	local variable #2, IDA にマークする var_8
EBP-4	local variable #1, IDA にマークする var_4
EBP	saved value of EBP
EBP+4	return address
EBP+8	引数#1, IDA にマークする arg_0
EBP+0xC	引数#2, IDA にマークする arg_4
EBP+0x10	引数#3, IDA にマークする arg_8
…	…

この例の `scanf()` 関数には2つの引数があります。

最初のものは%d を含む文字列へのポインタで、2番目のものは x 変数のアドレスです。

最初に、x 変数のアドレスが `lea eax, DWORD PTR _x$[ebp]` 命令によって EAX レジスタにロードされます。

LEA はロード実効アドレスの略で、アドレスを形成するためによく使用されます（(**??** on page ??))。

この場合、LEA は単に EBP レジスタ値と `_x$` マクロの合計を EAX レジスタに格納すると言うことができます。

これは `lea eax, [ebp-4]` と同じです。

したがって、EBP レジスタ値から4が減算され、その結果が EAX レジスタにロードされます。次に、EAX レジスタの値がスタックにプッシュされ、`scanf()` が呼び出されます。

`printf()` は最初の引数で呼び出されています。文字列へのポインタ：`You entered %d...\n`

2番目の引数は `mov ecx, [ebp-4]` で準備されています。命令は、ECX レジスタにそのアドレスではなく x 変数値を格納します。

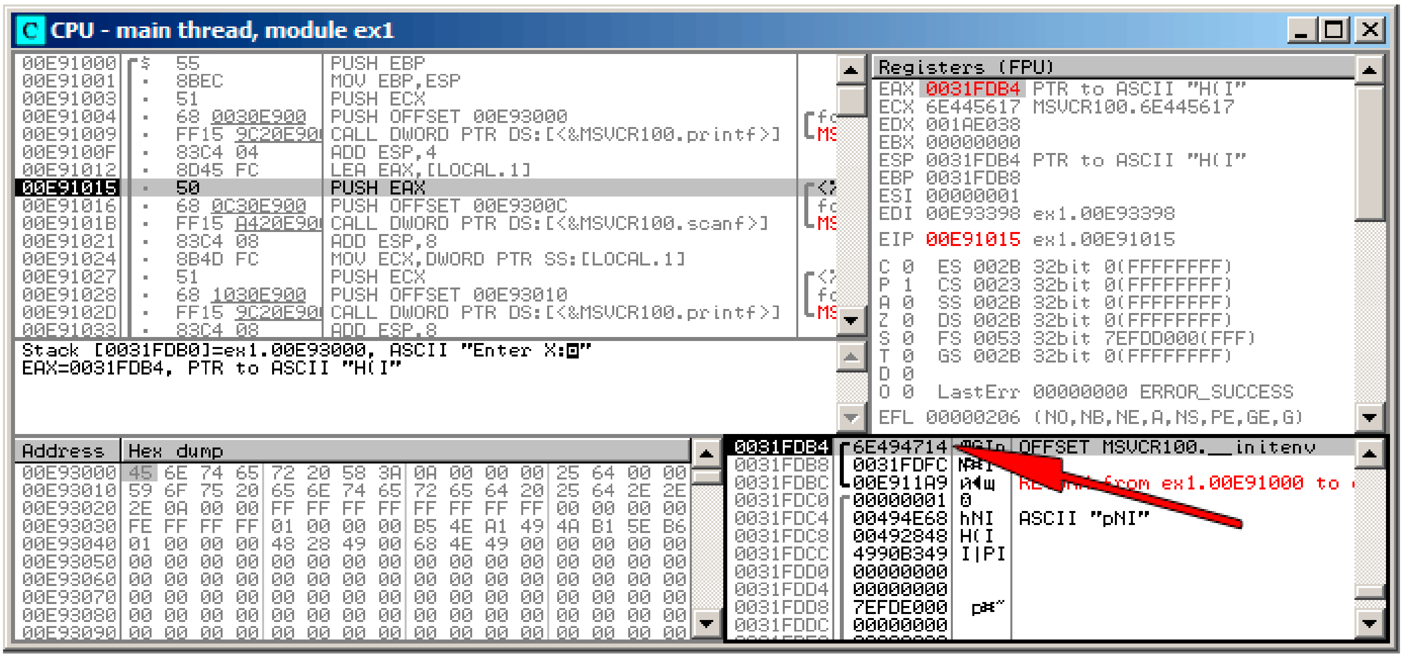
次に、ECX の値がスタックに格納され、最後の `printf()` が呼び出されます。

MSVC + OllyDbg

OllyDbg でこの例を試してみましょう。私たちが `ntdll.dll` の代わりに実行可能ファイルに達するまで、それをロードしてF8（ステップオーバー）を押し続けましょう。`main()` が表示されるまで上にスクロールします。

最初の命令（`PUSH EBP`）をクリックし、F2（ブレークポイントを設定)、次にF9（実行）を押します。`main()` が始まるとブレークポイントがトリガされます。

変数 x のアドレスが計算されるポイントまでトレースしましょう：



```
00E91000  $  55            PUSH EBP
00E91001  .  8BEC          MOV EBP,ESP
00E91003  .  51            PUSH ECX
00E91004  .  68 0030E900   PUSH OFFSET 00E93000
00E91009  .  FF15 9C20E90  CALL DWORD PTR DS:[<&MSVCR100.printf>]
00E9100F  .  83C4 04       ADD ESP,4
00E91012  .  8D45 FC       LEA EAX,[LOCAL.1]
00E91015  .  50            PUSH EAX
00E91016  .  68 0C30E900   PUSH OFFSET 00E9300C
00E9101B  .  FF15 A420E90  CALL DWORD PTR DS:[<&MSVCR100.scanf>]
00E91021  .  83C4 08       ADD ESP,8
00E91024  .  8B4D FC       MOV ECX,DWORD PTR SS:[LOCAL.1]
00E91027  .  51            PUSH ECX
00E91028  .  68 1030E900   PUSH OFFSET 00E93010
00E9102D  .  FF15 9C20E90  CALL DWORD PTR DS:[<&MSVCR100.printf>]
00E91033  .  83C4 08       ADD ESP,8
Stack [0031FDB0]=ex1.00E93000, ASCII "Enter X:◘"
EAX=0031FDB4, PTR to ASCII "H(I"
```

```
EAX 0031FDB4 PTR to ASCII "H(I"
ECX 6E445617 MSVCR100.6E445617
EDX 001AE038
EBX 00000000
ESP 0031FDB4 PTR to ASCII "H(I"
EBP 0031FDB8
ESI 00000001
EDI 00E93398 ex1.00E93398

EIP 00E91015 ex1.00E91015

C 0  ES 002B 32bit 0(FFFFFFFF)
P 1  CS 0023 32bit 0(FFFFFFFF)
A 0  SS 002B 32bit 0(FFFFFFFF)
Z 0  DS 002B 32bit 0(FFFFFFFF)
S 0  FS 0053 32bit 7EFDD000(FFF)
T 0  GS 002B 32bit 0(FFFFFFFF)
D 0
O 0  LastErr 00000000 ERROR_SUCCESS

EFL 00000206 (NO,NB,NE,A,NS,PE,GE,G)
```

```
00E93000 45 6E 74 65 72 20 58 3A 0A 00 00 00 25 64 00 00
00E93010 59 6F 75 20 65 6E 74 65 72 65 64 20 25 64 2E 2E
00E93020 2E 0A 00 00 FF FF FF FF FF FF FF FF 00 00 00 00
00E93030 FE FF FF FF 01 00 00 00 B5 4E A1 49 4A B1 5E B6
00E93040 01 00 00 00 48 28 49 00 68 4E 49 00 00 00 00 00
00E93050 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
00E93060 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
00E93070 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
00E93080 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
00E93090 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
```

```
0031FDB4  6E494714       OFFSET MSVCR100.__initenv
0031FDB8  0031FDFC
0031FDBC  00E911A9       from ex1.00E91000 to
0031FDC0  00000001  0
0031FDC4  00494E68  hNI  ASCII "pNI"
0031FDC8  00492848  H(I
0031FDCC  4990B349  I|PI
0031FDD0  00000000
0031FDD4  00000000
0031FDD8  7EFDE000
0031FDDC  00000000
```

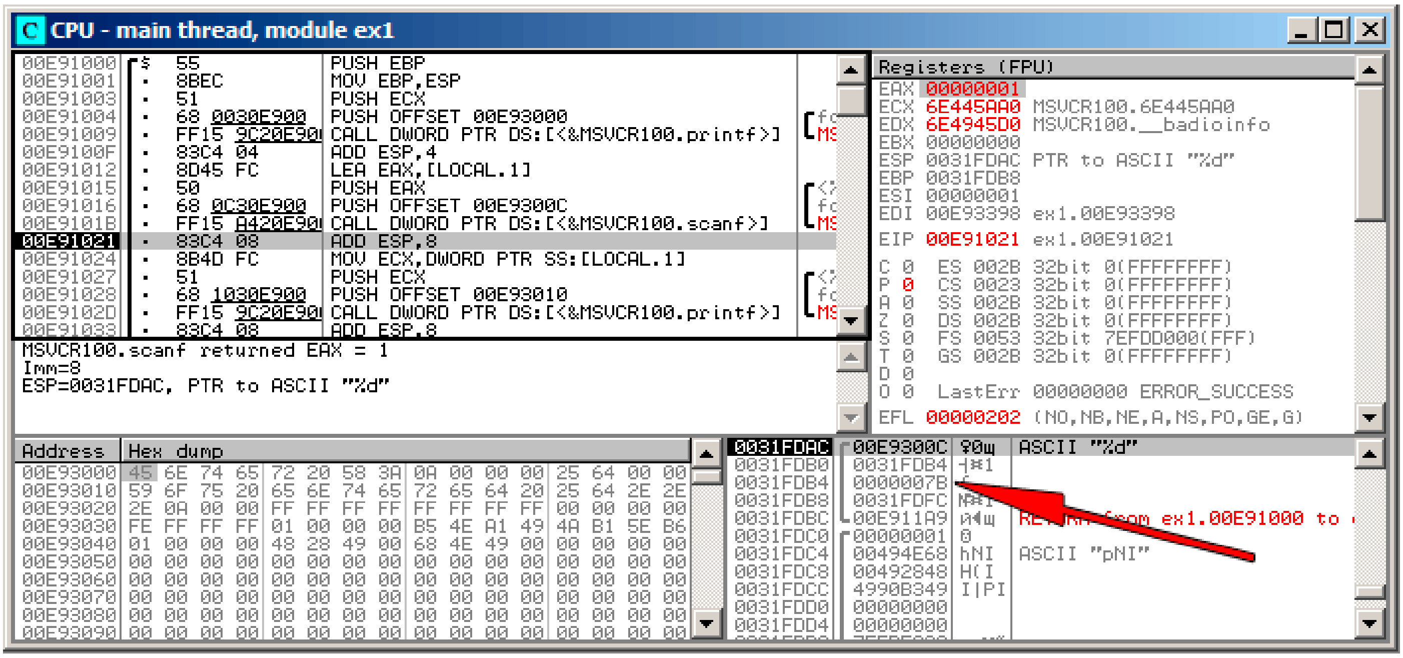
図 1.12: OllyDbg: ローカル変数のアドレスが計算されます。

レジスタウィンドウで EAX を右クリックして、「Follow in stack」を選択します。

このアドレスはスタックウィンドウに表示されます。赤い矢印が追加され、ローカルスタックの変数を指しています。その瞬間、この場所にはいくらかのゴミ（`0x6E494714`）が含まれています。今度は PUSH 命令の助けを借りて、このスタック要素のアドレスが次の位置の同じスタックに格納されます。`scanf()` の実行が完了するまで、F8を使ってトレースしてみましょう。`scanf()` の実行中に、コンソールウィンドウに123などを入力します。

```
Enter X:
123
```

scanf() はすでに実行を完了しました：



```
00E91000  $  55            PUSH EBP
00E91001  .  8BEC          MOV EBP,ESP
00E91003  .  51            PUSH ECX
00E91004  .  68 0030E900   PUSH OFFSET 00E93000
00E91009  .  FF15 9C20E90  CALL DWORD PTR DS:[<&MSVCR100.printf>]
00E9100F  .  83C4 04       ADD ESP,4
00E91012  .  8D45 FC       LEA EAX,[LOCAL.1]
00E91015  .  50            PUSH EAX
00E91016  .  68 0C30E900   PUSH OFFSET 00E9300C
00E9101B  .  FF15 A420E90  CALL DWORD PTR DS:[<&MSVCR100.scanf>]
00E91021  .  83C4 08       ADD ESP,8
00E91024  .  8B4D FC       MOV ECX,DWORD PTR SS:[LOCAL.1]
00E91027  .  51            PUSH ECX
00E91028  .  68 1030E900   PUSH OFFSET 00E93010
00E9102D  .  FF15 9C20E90  CALL DWORD PTR DS:[<&MSVCR100.printf>]
00E91033  .  83C4 08       ADD ESP,8
```

```
MSVCR100.scanf returned EAX = 1
Imm=8
ESP=0031FDAC, PTR to ASCII "%d"
```

```
Registers (FPU)
EAX 00000001
ECX 6E445AA0 MSVCR100.6E445AA0
EDX 6E4945D0 MSVCR100.__badioinfo
EBX 00000000
ESP 0031FDAC PTR to ASCII "%d"
EBP 0031FDB8
ESI 00000001
EDI 00E93398 ex1.00E93398

EIP 00E91021 ex1.00E91021

C 0  ES 002B 32bit 0(FFFFFFFF)
P 0  CS 0023 32bit 0(FFFFFFFF)
A 0  SS 002B 32bit 0(FFFFFFFF)
Z 0  DS 002B 32bit 0(FFFFFFFF)
S 0  FS 0053 32bit 7EFDD000(FFF)
T 0  GS 002B 32bit 0(FFFFFFFF)
D 0
O 0  LastErr 00000000 ERROR_SUCCESS

EFL 00000202 (NO,NB,NE,A,NS,PO,GE,G)
```

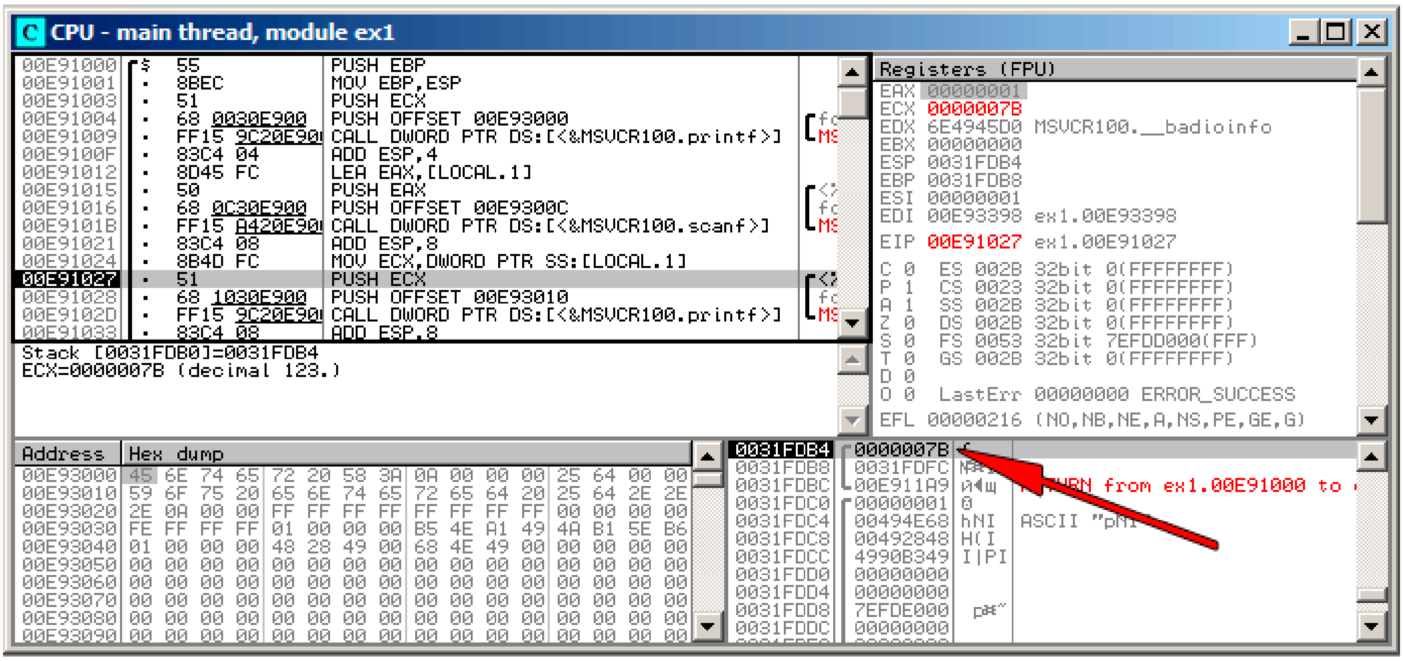
```
Address  Hex dump
00E93000 45 6E 74 65 72 20 58 3A 0A 00 00 00 25 64 00 00
00E93010 59 6F 75 20 65 6E 74 65 72 65 64 20 25 64 2E 2E
00E93020 2E 0A 00 00 FF FF FF FF FF FF FF FF 00 00 00 00
00E93030 FE FF FF FF 01 00 00 00 B5 4E A1 49 4A B1 5E B6
00E93040 01 00 00 00 48 28 49 00 68 4E 49 00 00 00 00 00
00E93050 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
00E93060 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
00E93070 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
00E93080 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
00E93090 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
```

```
0031FDAC  00E9300C  ASCII "%d"
0031FDB0  0031FDB4
0031FDB4  0000007B
0031FDB8  0031FDFC
0031FDBC  00E911A9  RE[illegible] from ex1.00E91000 to
0031FDC0  00000001
0031FDC4  00494E68  hNI  ASCII "pNI"
0031FDC8  00492848  H(I
0031FDCC  4990B349  I|PI
0031FDD0  00000000
0031FDD4  00000000
```

図 1.13: OllyDbg: scanf() が実行された

scanf() は EAX で1を返します。これは、1つの値を正常に読み取ったことを意味します。ローカル変数に対応するスタック要素をもう一度見ると、0x7B（123）が含まれています。

その後、この値はスタックから ECX レジスタにコピーされ、printf() に渡されます：



```
00E91000 $  55             PUSH EBP
00E91001 .  8BEC           MOV EBP,ESP
00E91003 .  51             PUSH ECX
00E91004 .  68 0030E900    PUSH OFFSET 00E93000
00E91009 .  FF15 9C20E90   CALL DWORD PTR DS:[<&MSVCR100.printf>]
00E9100F .  83C4 04        ADD ESP,4
00E91012 .  8D45 FC        LEA EAX,[LOCAL.1]
00E91015 .  50             PUSH EAX
00E91016 .  68 0C30E900    PUSH OFFSET 00E9300C
00E9101B .  FF15 A420E90   CALL DWORD PTR DS:[<&MSVCR100.scanf>]
00E91021 .  83C4 08        ADD ESP,8
00E91024 .  8B4D FC        MOV ECX,DWORD PTR SS:[LOCAL.1]
00E91027 .  51             PUSH ECX
00E91028 .  68 1030E900    PUSH OFFSET 00E93010
00E9102D .  FF15 9C20E90   CALL DWORD PTR DS:[<&MSVCR100.printf>]
00E91033 .  83C4 08        ADD ESP,8
Stack [0031FDB0]=0031FDB4
ECX=0000007B (decimal 123.)
```

```
Registers (FPU)
EAX 00000001
ECX 0000007B
EDX 6E4945D0 MSVCR100.__badioinfo
EBX 00000000
ESP 0031FDB4
EBP 0031FDB8
ESI 00000001
EDI 00E93398 ex1.00E93398

EIP 00E91027 ex1.00E91027

C 0  ES 002B 32bit 0(FFFFFFFF)
P 1  CS 0023 32bit 0(FFFFFFFF)
A 1  SS 002B 32bit 0(FFFFFFFF)
Z 0  DS 002B 32bit 0(FFFFFFFF)
S 0  FS 0053 32bit 7EFDD000(FFF)
T 0  GS 002B 32bit 0(FFFFFFFF)
D 0
O 0  LastErr 00000000 ERROR_SUCCESS
EFL 00000216 (NO,NB,NE,A,NS,PE,GE,G)
```

```
Address   Hex dump
00E93000  45 6E 74 65 72 20 58 3A 0A 00 00 00 25 64 00 00
00E93010  59 6F 75 20 65 6E 74 65 72 65 64 20 25 64 2E 2E
00E93020  2E 0A 00 00 FF FF FF FF FF FF FF FF 00 00 00 00
00E93030  FE FF FF FF 01 00 00 00 B5 4E A1 49 4A B1 5E B6
00E93040  01 00 00 00 48 28 49 00 68 4E 49 00 00 00 00 00
00E93050  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
00E93060  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
00E93070  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
00E93080  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
00E93090  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
```

```
0031FDB4  0000007B  {
0031FDB8  0031FDFC
0031FDBC  00E911A9          RETURN from ex1.00E91000 to
0031FDC0  00000001  0
0031FDC4  00494E68  hNI     ASCII "pNI
0031FDC8  00492848  H(I
0031FDCC  4990B349  I|PI
0031FDD0  00000000
0031FDD4  00000000
0031FDD8  7EFDE000
0031FDDC  00000000
```

図 1.14: OllyDbg: printf() に渡す値を準備する

GCC

Linux上のGCC 4.4.1でこのコードをコンパイルしようとしましょう。

```
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は printf() 呼び出しを puts() の呼び出しで置き換えました。この理由は、(1.5.4 on page 20) で説明されました。

MSVCの例のように、引数は MOV 命令を使用してスタックに配置されます。

ところで

ところで、この単純な例は、コンパイラが C/C++ ブロックの式のリストを命令の連続したリストに変換するという事実のデモンストレーションです。C/C++ の式の間には何もないので、結果のマシンコードには、ある式から次の式への制御フローの間には何もありません。

x64

ここの画像は、スタックではなくレジスタが引数の受け渡しに使用されるという違いと似ています。

MSVC

Listing 1.68: 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

        ; return 0
        xor     eax, eax
        add     rsp, 56
        ret     0
main    ENDP
_TEXT   ENDS
```

GCC

Listing 1.69: 最適化 GCC 4.4.6 x64

```
.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

        ; return 0
        xor     eax, eax
        add     rsp, 24
        ret
```

ARM

最適化 **Keil 6/2013 (Thumbモード)**

```
.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}
```

`scanf()` がitemを読み込むためには、*int* へのparameter.pointerが必要です。*int* は32ビットなので、メモリのどこかに格納するには4バイトが必要で、32ビットのレジスタに正確に収まります。ローカル変数 `x` の場所がスタックに割り当てられ、IDA の名前は *var_8* です。ただし、SP（stack pointer）がすでにその領域を指しているため、その領域を直接割り当てることはできません。

したがって、SPの値は `R1` レジスタにコピーされ、フォーマット文字列とともに `scanf()` に渡されます。その後、`LDR` 命令の助けを借りて、この値はスタックから `R1` レジスタに移動され、`printf()` に渡されます。

ARM64

Listing 1.70: 非最適化 GCC 4.9.1 ARM64

```
.LC0:
        .string "Enter X:"
.LC1:
        .string "%d"
.LC2:
        .string "You entered %d...\n"
scanf_main:
; subtract 32 from SP, then save FP and LR in stack frame:
        stp     x29, x30, [sp, -32]!
; set stack frame (FP=SP)
        add     x29, sp, 0
; load pointer to the "Enter X:" string:
        adrp    x0, .LC0
        add     x0, x0, :lo12:.LC0
; X0=pointer to the "Enter X:" string
; print it:
        bl      puts
; load pointer to the "%d" string:
        adrp    x0, .LC1
        add     x0, x0, :lo12:.LC1
; find a space in stack frame for "x" variable (X1=FP+28):
        add     x1, x29, 28
; X1=address of "x" variable
; pass the address to scanf() and call it:
        bl      __isoc99_scanf
; load 32-bit value from the variable in stack frame:
        ldr     w1, [x29,28]
; W1=x
; load pointer to the "You entered %d...\n" string
```

```
30  ; printf() will take text string from X0 and "x" variable from X1 (or W1)
31          adrp    x0, .LC2
32          add     x0, x0, :lo12:.LC2
33          bl      printf
34  ; return 0
35          mov     w0, 0
36  ; restore FP and LR, then add 32 to SP:
37          ldp     x29, x30, [sp], 32
38          ret
```

スタックフレームには32バイトが割り当てられており、必要なサイズよりも大きくなっています。たぶんメモリのアラインメントの問題でしょうか？最も興味深いのはスタックフレーム内の x 変数のためのスペースを見つけることです（22行目）。なぜ28なのでしょう？何らかの理由で、コンパイラは、この変数をスタックフレームの最後に置きます。アドレスは `scanf()` に渡され、`scanf()` はユーザ入力値をそのアドレスのメモリに格納するだけです。これは *int* 型の32ビット値です。値は27行目から取得され、`printf()` に渡されます。

MIPS

ローカルスタック内の場所は x 変数に割り当てられ、$\$sp+24$ と呼ばれます。

そのアドレスは `scanf()` に渡され、ユーザー入力値は INCLW（「Load Word」）を使用してロードされます。そしてそれから `printf()` に渡されます。

Listing 1.71: 最適化 GCC 4.4.5 (アセンブリ出力)

```
$LC0:
        .ascii  "Enter X:\000"
$LC1:
        .ascii  "%d\000"
$LC2:
        .ascii  "You entered %d...\012\000"
main:
; 関数プロローグ:
        lui     $28,%hi(__gnu_local_gp)
        addiu   $sp,$sp,-40
        addiu   $28,$28,%lo(__gnu_local_gp)
        sw      $31,36($sp)
; puts()を呼び出す:
        lw      $25,%call16(puts)($28)
        lui     $4,%hi($LC0)
        jalr    $25
        addiu   $4,$4,%lo($LC0) ; branch delay slot
; scanf()を呼び出す:
        lw      $28,16($sp)
        lui     $4,%hi($LC1)
        lw      $25,%call16(__isoc99_scanf)($28)
; scanf()の2番目の引数に $a1=$sp+24 を設定する:
        addiu   $5,$sp,24
        jalr    $25
        addiu   $4,$4,%lo($LC1) ; branch delay slot

; printf()を呼び出す:
        lw      $28,16($sp)
; printf()の2番目の引数を設定する,
; アドレス$sp+24にwordをロードする:
        lw      $5,24($sp)
        lw      $25,%call16(printf)($28)
        lui     $4,%hi($LC2)
        jalr    $25
        addiu   $4,$4,%lo($LC2) ; branch delay slot

; 関数エピローグ:
        lw      $31,36($sp)
; 戻り値に0を設定する:
        move    $2,$0
; return:
        j       $31
        addiu   $sp,$sp,40      ; branch delay slot
```

IDAはスタックレイアウトを次のように表示します。

Listing 1.72: 最適化 GCC 4.4.5 (IDA)

```
.text:00000000 main:
.text:00000000
.text:00000000 var_18   = -0x18
.text:00000000 var_10   = -0x10
.text:00000000 var_4    = -4
.text:00000000
; 関数プロローグ:
.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)
; puts()を呼び出す:
.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:" ; branch delay slot
; scanf()を呼び出す:
.text:00000024          lw      $gp, 0x28+var_18($sp)
.text:00000028          lui     $a0, ($LC1 >> 16)  # "%d"
.text:0000002C          lw      $t9, (__isoc99_scanf & 0xFFFF)($gp)
; scanf()の2番目の引数に $a1=$sp+24 を設定する:
.text:00000030          addiu   $a1, $sp, 0x28+var_10
.text:00000034          jalr    $t9  ; branch delay slot
.text:00000038          la      $a0, ($LC1 & 0xFFFF)  # "%d"
; printf()を呼び出す:
.text:0000003C          lw      $gp, 0x28+var_18($sp)
; printf()の2番目の引数を設定する,
; アドレス$sp+24にwordをロードする:
.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" ; branch delay ↙
    ↪ slot
; 関数エピローグ:
.text:00000054          lw      $ra, 0x28+var_4($sp)
; 戻り値に0を設定する:
.text:00000058          move    $v0, $zero
; return:
.text:0000005C          jr      $ra
.text:00000060          addiu   $sp, 0x28 ; branch delay slot
```

第1.9.2節一般的な間違い

x へのポインタではなく、*x* の値を渡すのは極めて一般的な間違い（および/またはタイプミス）です。

```
#include <stdio.h>

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

        scanf ("%d", x); // BUG

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

        return 0;
};
```

では、何が起こるでしょうか？*x* は初期化されておらず、ローカルスタックからのランダムノイズを含んでいます。`scanf()` が呼び出されると、ユーザーから文字列を受け取り、数値に解析し、*x* に書き込んでメモリ内のア

ドレスとして扱います。しかしランダムなノイズがあるので、scanf() はランダムなアドレスに書き込もうとします。おそらく、プロセスがクラッシュするでしょう。

興味深いことに、デバッグビルドのいくつかの CRT ライブラリは、視覚的に特徴的なパターンを 0xCCCCCCCC や 0x0BADF00D のように割り当てられたメモリに入れています。この場合、*x* は0xCCCCCCCCを含むことができ、scanf() はアドレス0xCCCCCCCCに書き込みを試みます。また、プロセス内の何かがアドレス0xCCCCCCCCに書き込もうとすると、初期化されていない変数（またはポインタ）が事前初期化なしで使用されることがわかります。これは、新しく割り当てられたメモリがちょうどクリアされた場合よりも優れています。

第1.9.3節グローバル変数

前の例の *x* 変数がローカルではなく、グローバル変数であればどうでしょうか？それから、関数本体からだけでなく、どの時点からでもアクセスできるようになりました。グローバル変数はanti-patternと見なされますが、実験のために行ってみましょう。

```
#include <stdio.h>

// now x is global variable
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
```

この場合、*x* 変数は _DATA セグメントに定義され、ローカルスタックにはメモリは割り当てられません。スタックからではなく、直接アクセスされます。初期化されていないグローバル変数は、実行可能ファイルにスペースを入れません（なぜ、最初に変数をゼロに設定する必要があるのでしょうか?)。しかし、誰かが自分のアドレスにアクセスすると、OSは0で初期化されたブロック[70]を割り当てます。

変数に明示的に値を割り当てましょう：

```
int x=10; // default value
```

以下を得ます。

```
_DATA   SEGMENT
_x      DD      0aH

...
```

ここでは、この変数のDWORDタイプの値 `0xA`（DDはDWORD = 32ビットを表します）が表示されます。

IDA にコンパイルされた.exeを開くと、_DATA セグメントの先頭に *x* 変数が配置されていて、その後にテキスト文字列が表示されます。

x の値が設定されていない前の例のコンパイル済み.exeを IDA で開くと、次のように表示されます。

Listing 1.73: 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
```

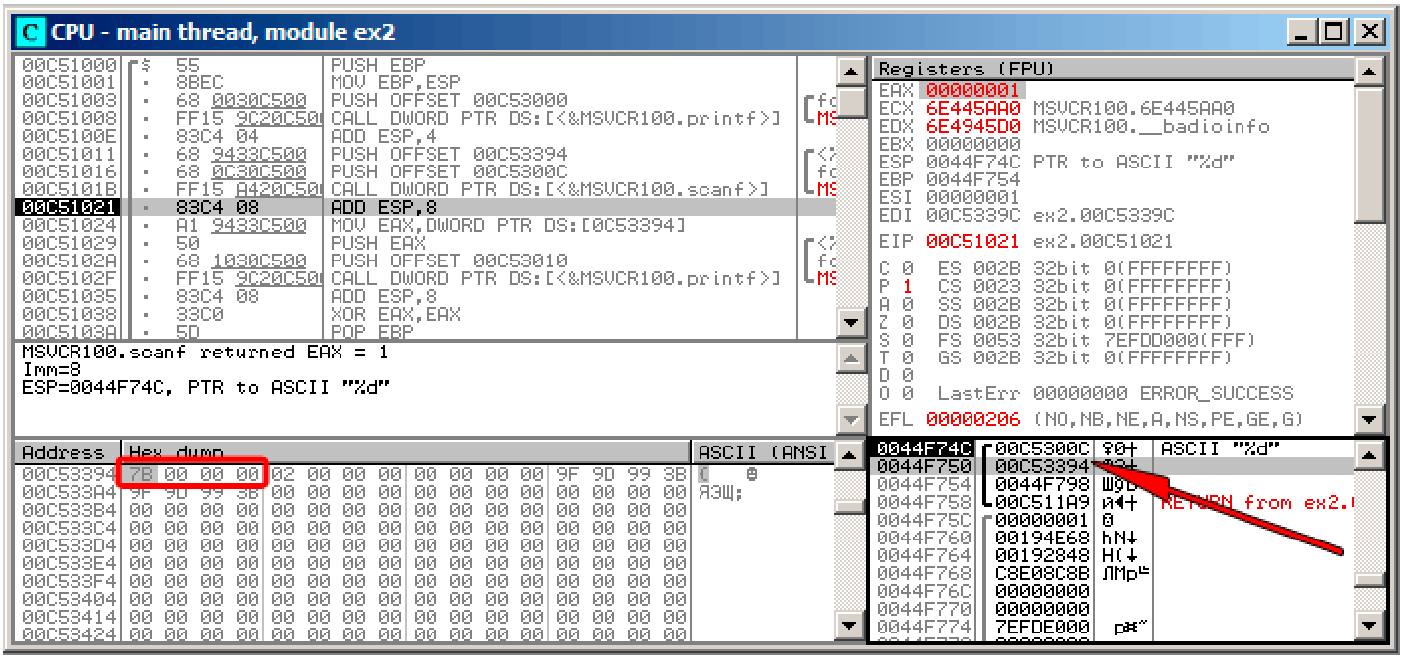
_x に? がマークされていると、残りの変数は初期化する必要はありません。これは、メモリに.exeをロードした後、これらすべての変数のための領域が割り当てられ、0で満たされる [*ISO/IEC 9899:TC3 (C C99 standard)*, (2007)6.7.8p10] ことを意味します。しかし、.exeファイルでは、これらの初期化されていない変数は何も占有しません。これは、例えば、大きな配列の場合に便利です。

---

[70] これがVM[71]の動作です

MSVC: x86 + OllyDbg

ここではさらに単純です。



```
00C51000 ┌$   55            PUSH EBP
00C51001 │·   8BEC          MOV EBP,ESP
00C51003 │·   68 0030C500   PUSH OFFSET 00C53000
00C51008 │·   FF15 9C20C50  CALL DWORD PTR DS:[<&MSVCR100.printf>]
00C5100E │·   83C4 04       ADD ESP,4
00C51011 │·   68 9433C500   PUSH OFFSET 00C53394
00C51016 │·   68 0C30C500   PUSH OFFSET 00C5300C
00C5101B │·   FF15 A420C50  CALL DWORD PTR DS:[<&MSVCR100.scanf>]
00C51021 │·   83C4 08       ADD ESP,8
00C51024 │·   A1 9433C500   MOV EAX,DWORD PTR DS:[0C53394]
00C51029 │·   50            PUSH EAX
00C5102A │·   68 1030C500   PUSH OFFSET 00C53010
00C5102F │·   FF15 9C20C50  CALL DWORD PTR DS:[<&MSVCR100.printf>]
00C51035 │·   83C4 08       ADD ESP,8
00C51038 │·   33C0          XOR EAX,EAX
00C5103A │·   5D            POP EBP
```

```
MSVCR100.scanf returned EAX = 1
Imm=8
ESP=0044F74C, PTR to ASCII "%d"
```

```
Registers (FPU)
EAX 00000001
ECX 6E445AA0 MSVCR100.6E445AA0
EDX 6E4945D0 MSVCR100.__badioinfo
EBX 00000000
ESP 0044F74C PTR to ASCII "%d"
EBP 0044F754
ESI 00000001
EDI 00C5339C ex2.00C5339C

EIP 00C51021 ex2.00C51021

C 0  ES 002B 32bit 0(FFFFFFFF)
P 1  CS 0023 32bit 0(FFFFFFFF)
A 0  SS 002B 32bit 0(FFFFFFFF)
Z 0  DS 002B 32bit 0(FFFFFFFF)
S 0  FS 0053 32bit 7EFDD000(FFF)
T 0  GS 002B 32bit 0(FFFFFFFF)
D 0
O 0  LastErr 00000000 ERROR_SUCCESS

EFL 00000206 (NO,NB,NE,A,NS,PE,GE,G)
```

```
Address  Hex dump                                         ASCII (ANSI
00C53394 7B 00 00 00 02 00 00 00 00 00 00 00 9F 9D 99 3B  {     ☻
00C533A4 9F 9D 99 3B 00 00 00 00 00 00 00 00 00 00 00 00  ЯЭЩ;
00C533B4 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
00C533C4 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
00C533D4 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
00C533E4 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
00C533F4 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
00C53404 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
00C53414 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
00C53424 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
```

```
0044F74C ┌00C5300C ♀0+   ASCII "%d"
0044F750 │00C53394
0044F754 │0044F798 Ш9D
0044F758 └00C511A9 И◄+  RETURN from ex2.
0044F75C ┌00000001 ☺
0044F760 │00194E68 hN↓
0044F764 │00192848 H(↓
0044F768 │C8E08C8B ЛМp╘
0044F76C │00000000
0044F770 │00000000
0044F774 │7EFDE000  p¤˝
```

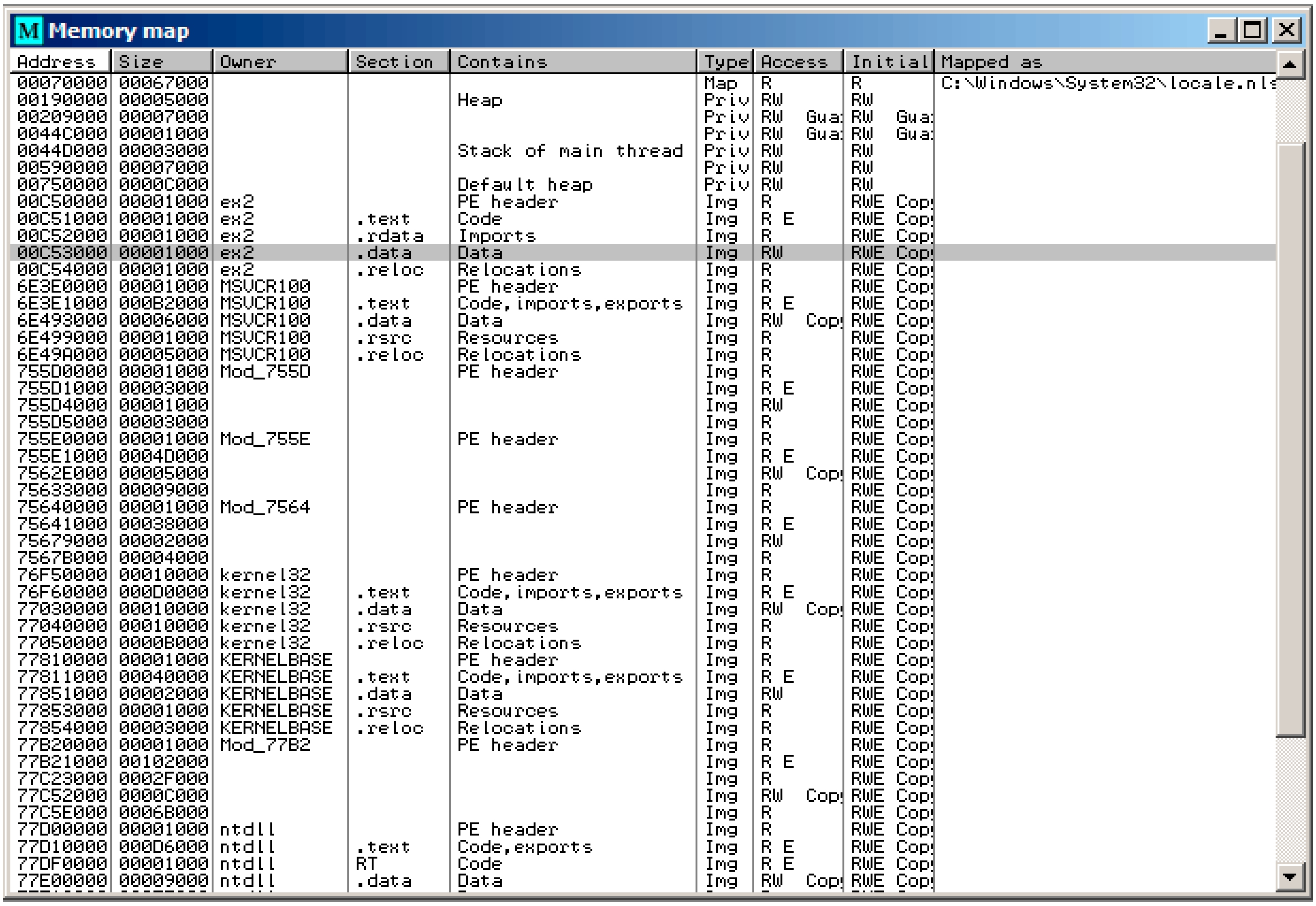
図 1.15: OllyDbg: after `scanf()` execution

変数はデータセグメントにあります。PUSH 命令（x のアドレスを押す）が実行されると、アドレスがスタックウィンドウに表示されます。その行を右クリックし、「ダンプに従う」を選択します。変数は、左側のメモリウィンドウに表示されます。コンソールに123を入力すると、メモリウィンドウに `0x7B` が表示されます（ハイライトされたスクリーンショット領域を参照）。

しかし､最初のバイトはなぜ7Bでしょうか？論理的に考えると､`00 00 00 7B` のはずです｡この原因はendiannessと呼ばれるもので、x86はリトルエンディアンを使用します。これは、最下位バイトが最初に書き込まれ、最上位バイトが最後に書き込まれることを意味します。これについての詳細：**??** on page ?? この例では、32ビットの値がこのメモリアドレスから EAX にロードされ、`printf()` に渡されます。

x のメモリアドレスは `0x00C53394` です。

OllyDbg では、プロセスメモリマップ（Alt-M）を見ることができ、このアドレスはプログラムの`.data` PEセグメント内にあることがわかります。



```
Address  Size     Owner      Section  Contains                Type Access   Initial  Mapped as
00070000 00067000                                              Map  R        R        C:\Windows\System32\locale.nl
00190000 00005000                     Heap                    Priv RW       RW
00209000 00007000                                              Priv RW  Gua  RW  Gua
0044C000 00001000                                              Priv RW  Gua  RW  Gua
0044D000 00003000                     Stack of main thread    Priv RW       RW
00590000 00007000                                              Priv RW       RW
00750000 0000C000                     Default heap            Priv RW       RW
00C50000 00001000 ex2                 PE header               Img  R        RWE Cop
00C51000 00001000 ex2        .text    Code                    Img  R E      RWE Cop
00C52000 00001000 ex2        .rdata   Imports                 Img  R        RWE Cop
00C53000 00001000 ex2        .data    Data                    Img  RW       RWE Cop
00C54000 00001000 ex2        .reloc   Relocations             Img  R        RWE Cop
6E3E0000 00001000 MSVCR100            PE header               Img  R        RWE Cop
6E3E1000 000B2000 MSVCR100   .text    Code,imports,exports    Img  R E      RWE Cop
6E493000 00006000 MSVCR100   .data    Data                    Img  RW  Cop  RWE Cop
6E499000 00001000 MSVCR100   .rsrc    Resources               Img  R        RWE Cop
6E49A000 00005000 MSVCR100   .reloc   Relocations             Img  R        RWE Cop
755D0000 00001000 Mod_755D            PE header               Img  R        RWE Cop
755D1000 00003000                                              Img  R E      RWE Cop
755D4000 00001000                                              Img  RW       RWE Cop
755D5000 00003000                                              Img  R        RWE Cop
755E0000 00001000 Mod_755E            PE header               Img  R        RWE Cop
755E1000 0004D000                                              Img  R E      RWE Cop
7562E000 00005000                                              Img  RW  Cop  RWE Cop
75633000 00009000                                              Img  R        RWE Cop
75640000 00001000 Mod_7564            PE header               Img  R        RWE Cop
75641000 00038000                                              Img  R E      RWE Cop
75679000 00002000                                              Img  RW       RWE Cop
7567B000 00004000                                              Img  R        RWE Cop
76F50000 00010000 kernel32            PE header               Img  R        RWE Cop
76F60000 000D0000 kernel32   .text    Code,imports,exports    Img  R E      RWE Cop
77030000 00010000 kernel32   .data    Data                    Img  RW  Cop  RWE Cop
77040000 00010000 kernel32   .rsrc    Resources               Img  R        RWE Cop
77050000 0000B000 kernel32   .reloc   Relocations             Img  R        RWE Cop
77810000 00001000 KERNELBASE          PE header               Img  R        RWE Cop
77811000 00040000 KERNELBASE .text    Code,imports,exports    Img  R E      RWE Cop
77851000 00002000 KERNELBASE .data    Data                    Img  RW       RWE Cop
77853000 00001000 KERNELBASE .rsrc    Resources               Img  R        RWE Cop
77854000 00003000 KERNELBASE .reloc   Relocations             Img  R        RWE Cop
77B20000 00001000 Mod_77B2            PE header               Img  R        RWE Cop
77B21000 00102000                                              Img  R E      RWE Cop
77C23000 0002F000                                              Img  R        RWE Cop
77C52000 0000C000                                              Img  RW  Cop  RWE Cop
77C5E000 0006B000                                              Img  R        RWE Cop
77D00000 00001000 ntdll               PE header               Img  R        RWE Cop
77D10000 000D6000 ntdll      .text    Code,exports            Img  R E      RWE Cop
77DF0000 00001000 ntdll      RT       Code                    Img  R E      RWE Cop
77E00000 00009000 ntdll      .data    Data                    Img  RW  Cop  RWE Cop
```

図 1.16: OllyDbg: process memory map

GCC: x86

Linuxの画像はほぼ同じですが、初期化されていない変数は `_bss` セグメントにあります。ELF[72]ファイルでは、このセグメントには次の属性があります。

```
; Segment type: Uninitialized
; Segment permissions: Read/Write
```

ただし、変数をある値で初期化してください。10の場合、次の属性を持つ `_data` セグメントに配置されます。

```
; Segment type: Pure data
; Segment permissions: Read/Write
```

MSVC: x64

Listing 1.74: 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
```

---

[72] Linuxを含め*NIXシステムで広く使用される実行ファイルフォーマット

```
_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

        ; return 0
        xor     eax, eax

        add     rsp, 40
        ret     0
main    ENDP
_TEXT   ENDS
```

コードはx86とほとんど同じです。x 変数のアドレスは、LEA 命令を使用して `scanf()` に渡され、変数の値は MOV 命令を使用して2番目の `printf()` に渡されることに注意してください。`DWORD PTR` はアセンブリ言語の一部であり（マシンコードと無関係）、可変データサイズが32ビットであり、MOV 命令がそれに応じてエンコードされなければならないことを示します。

ARM: 最適化 Keil 6/2013 (Thumbモード)

Listing 1.75: 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
```

したがって、x 変数は現在グローバルであり、このために別のセグメント、つまりデータセグメント（*.data*）に配置されています。テキスト文字列がコードセグメント（*.text*）にあり、x がここにあるのはなぜでしょうか？これは変数なので、定義上、その値は変更される可能性があります。さらに、頻繁に変更される可能性があります。テキスト文字列は定数型ですが、変更されないため、*.text* セグメントに配置されます。

コードセグメントは、時にはROM[73]チップに配置されることがあります。（ここでは、組み込み電子機器を扱います。メモリ不足が普通です）変更可能な変数はRAMに配置されます。

ROMを持っているときは、定数変数をRAMに格納するのはそれほど経済的ではありません。

さらに、RAMの定数変数は初期化する必要があります。これは、電源投入後、明らかにRAMにランダム情報が含まれているためです。

次に、コードセグメント内の x（`off_2C`）変数へのポインターが表示され、変数を使用するすべての操作はこのポインターを介して行われます。

これは、x 変数がこの特定のコードフラグメントから離れた場所に配置される可能性があるため、そのアドレスをコードのすぐ近くに保存する必要があるためです。

Thumbモードの LDR 命令は、その位置から1020バイトの範囲内の変数と、±4095 バイトの範囲のARMモードの変数からのみアドレス可能です。

したがって、x 変数のアドレスは、リンカがコードの近くのどこかに変数を格納できる保証がないため、近い場所に配置する必要があります。外部メモリチップでもうまくいくかもしれません！

もう1つ：変数が *const* として宣言されている場合、Keilコンパイラは`.constdata` セグメントにそれを割り当てます。

その後、リンカはこのセグメントをROMにコードセグメントとともに配置することができます。

ARM64

Listing 1.76: 非最適化 GCC 4.9.1 ARM64

```
        .comm   x,4,4
.LC0:
        .string "Enter X:"
.LC1:
        .string "%d"
.LC2:
        .string "You entered %d...\n"
f5:
; save FP and LR in stack frame:
        stp     x29, x30, [sp, -16]!
; set stack frame (FP=SP)
        add     x29, sp, 0
; load pointer to the "Enter X:" string:
        adrp    x0, .LC0
        add     x0, x0, :lo12:.LC0
        bl      puts
; load pointer to the "%d" string:
        adrp    x0, .LC1
        add     x0, x0, :lo12:.LC1
; form address of x global variable:
        adrp    x1, x
        add     x1, x1, :lo12:x
        bl      __isoc99_scanf
; form address of x global variable again:
        adrp    x0, x
        add     x0, x0, :lo12:x
; load value from memory at this address:
        ldr     w1, [x0]
; load pointer to the "You entered %d...\n" string:
        adrp    x0, .LC2
        add     x0, x0, :lo12:.LC2
        bl      printf
; return 0
        mov     w0, 0
; restore FP and LR:
```

---

[73] Japanese text placeholder

```
36          ldp     x29, x30, [sp], 16
37          ret
```

この場合、x 変数はグローバルとして宣言され、そのアドレスは ADRP/ADD 命令ペア（21行目と25行目）を使用して計算されます。

MIPS

初期化されていないグローバル変数

だから今 x 変数はグローバルです。オブジェクトファイルではなく実行ファイルにコンパイルし、IDA にロードしてみましょう。IDAは、*.sbss* ELFセクションに x 変数を表示します（25ページの「グローバルポインタ」1.5.5 on page 24を覚えておいてください）。これは変数が最初に初期化されていないためです。

Listing 1.77: 最適化 GCC 4.4.5 (IDA)

```
.text:004006C0 main:
.text:004006C0
.text:004006C0 var_10          = -0x10
.text:004006C0 var_4           = -4
.text:004006C0
; 関数プロローグ:
.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)
; puts()を呼び出す:
.text:004006D4                 la      $t9, puts
.text:004006D8                 lui     $a0, 0x40
.text:004006DC                 jalr    $t9 ; puts
.text:004006E0                 la      $a0, aEnterX     # "Enter X:" ; branch delay slot
; scanf()を呼び出す:
.text:004006E4                 lw      $gp, 0x20+var_10($sp)
.text:004006E8                 lui     $a0, 0x40
.text:004006EC                 la      $t9, __isoc99_scanf
; xのアドレスを準備する:
.text:004006F0                 la      $a1, x
.text:004006F4                 jalr    $t9 ; __isoc99_scanf
.text:004006F8                 la      $a0, aD          # "%d"       ; branch delay slot
; printf()を呼び出す:
.text:004006FC                 lw      $gp, 0x20+var_10($sp)
.text:00400700                 lui     $a0, 0x40
; xのアドレスを取得する:
.text:00400704                 la      $v0, x
.text:00400708                 la      $t9, printf
; 変数xから値をロードして$a1にてprintf()に値を渡す:
.text:0040070C                 lw      $a1, (x - 0x41099C)($v0)
.text:00400710                 jalr    $t9 ; printf
.text:00400714                 la      $a0, aYouEnteredD___  # "You entered %d...\n" ; branch ↙
    ↳ delay slot
; 関数エピローグ:
.text:00400718                 lw      $ra, 0x20+var_4($sp)
.text:0040071C                 move    $v0, $zero
.text:00400720                 jr      $ra
.text:00400724                 addiu   $sp, 0x20  ; branch delay slot

...

.sbss:0041099C  # Segment type: Uninitialized
.sbss:0041099C                 .sbss
.sbss:0041099C                 .globl x
.sbss:0041099C x:              .space 4
.sbss:0041099C
```

IDAは情報量を減らすため、objdumpを使用してリスティングを行い、コメントします。

Listing 1.78: 最適化 GCC 4.4.5 (objdump)

```
 1 004006c0 <main>:
 2 ; 関数プロローグ:
 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 ; puts()を呼び出す:
 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 ; branch delay slot
13 ; scanf()を呼び出す:
14   4006e4:       8fbc0010        lw      gp,16(sp)
15   4006e8:       3c040040        lui     a0,0x40
16   4006ec:       8f998038        lw      t9,-32712(gp)
17 ; xのアドレスを準備する:
18   4006f0:       8f858044        lw      a1,-32700(gp)
19   4006f4:       0320f809        jalr    t9
20   4006f8:       248408fc        addiu   a0,a0,2300 ; branch delay slot
21 ; printf()を呼び出す:
22   4006fc:       8fbc0010        lw      gp,16(sp)
23   400700:       3c040040        lui     a0,0x40
24 ; xのアドレスを取得する:
25   400704:       8f828044        lw      v0,-32700(gp)
26   400708:       8f99803c        lw      t9,-32708(gp)
27 ; 変数xから値をロードして$a1にてprintf()に値を渡す:
28   40070c:       8c450000        lw      a1,0(v0)
29   400710:       0320f809        jalr    t9
30   400714:       24840900        addiu   a0,a0,2304 ; branch delay slot
31 ; 関数エピローグ:
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   ; branch delay slot
36 ; 次の関数の開始アドレスが16バイト境界になるようにNOPで埋める:
37   400728:       00200825        move    at,at
38   40072c:       00200825        move    at,at
```

今度は x 変数アドレスがGPを使って64KiBのデータバッファから読み込まれ、負のオフセットが加えられていることがわかります（18行目）。さらに、この例（puts()、scanf()、printf()）で使用されている3つの外部関数のアドレスもGPを使用して64KiBグローバルデータバッファから読み込まれます（9,16,26行目）。GPはバッファの中央を指しています。このようなオフセットは、3つの関数のアドレスと x 変数のアドレスがすべてそのバッファの先頭に格納されていることを示しています。私たちの例は非常に小さいので、それは理にかなっています。

言及する価値がある別のことは、次の関数の開始を16バイトの境界に合わせるために、関数が2つのNOP（MOVE $AT,$AT、アイドル命令）で終了することです。

初期化されたグローバル変数

x 変数にデフォルト値を与えることで、この例を変更しましょう。

```
int x=10; // default value
```

IDAは x 変数が.dataセクションに存在することを示しています：

Listing 1.79: 最適化 GCC 4.4.5 (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)
; アドレスxの高ビットを準備する:
.text:004006CC                 lui     $s0, 0x41
.text:004006D0                 la      $t9, __isoc99_scanf
.text:004006D4                 lui     $a0, 0x40
; アドレスxの低ビットを準備する:
.text:004006D8                 addiu   $a1, $s0, (x - 0x410000)
; アドレスxは$a1にあります
.text:004006DC                 jalr    $t9 ; __isoc99_scanf
.text:004006E0                 la      $a0, aD          # "%d"
.text:004006E4                 lw      $gp, 0x20+var_10($sp)
; メモリからwordを取得する:
.text:004006E8                 lw      $a1, x
; xの値は$a1にあります
.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
```

.sdataにしたら？これはおそらくいくつかのGCCオプションに依存するのでしょうか？

それにもかかわらず、x は一般的なメモリ領域である.dataにあり、ここで変数を扱う方法を見てみることができます。

変数のアドレスは、命令のペアを使用して構成する必要があります。

私たちの場合、それらは `LUI` （「Load Upper Immediate」）と `ADDIU` （「Add Immediate Unsigned Word」）です。

厳密な検査のためのobjdumpリストもあります：

Listing 1.80: 最適化 GCC 4.4.5 (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)
; アドレスxの高ビットを準備する:
  4006cc:       3c100041        lui     s0,0x41
  4006d0:       8f998038        lw      t9,-32712(gp)
  4006d4:       3c040040        lui     a0,0x40
; アドレスxの低ビットに加える:
  4006d8:       26050920        addiu   a1,s0,2336
; アドレスxは$a1にあります
```

```
  4006dc:       0320f809        jalr    t9
  4006e0:       248408dc        addiu   a0,a0,2268
  4006e4:       8fbc0010        lw      gp,16(sp)
; アドレスxの高ビットは$s0にあります:
; アドレスxの低ビットに加えて、メモリからwordをロードする:
  4006e8:       8e050920        lw      a1,2336(s0)
; xの値は$a1にあります
  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
```

アドレスは LUI と ADDIU を使用して形成されていますが、アドレスの上位部分はまだ $S0 レジスタにあり、LW（「Load Word」）命令でオフセットをエンコードすることができます。変数から値をロードして printf() に渡すには十分です。

一時的なデータを保持するレジスタの先頭にはT-が付いていますが、ここでは接頭辞S-が付いています。その内容は他の関数で使用する前に保持しておく必要があります。

そのため、$S0 の値は0x4006ccのアドレスに設定されており、scanf() 呼び出し後に0x4006e8番地で再び使用されています。scanf() 関数は値を変更しません。

第1.9.4節scanf()

前述のように、今日 scanf() を使用するのはちょっと古めかしいです。しかし、必要ならば、scanf() がエラーなく正しく終了するかどうかを確認する必要があります。

```
#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;
};
```

標準では、scanf()[74]関数は正常に読み取られたフィールドの数を返します。

私たちの場合、すべてがうまく行き、ユーザーが数字を入力した場合、scanf() は1を返し、エラー（またはEOF[75]）では0を返します。

scanf() の戻り値をチェックするためのCコードを追加し、エラーの場合にはエラーメッセージを出力してみましょう。

期待どおりに動作します。

```
C:\...>ex3.exe
Enter X:
123
You entered 123...

C:\...>ex3.exe
Enter X:
ouch
What you entered? Huh?
```

---

[74] scanf, wscanf: MSDN

[75] End of File

MSVC: x86

アセンブリ出力（MSVC 2010）の内容は次のとおりです。

```
        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
```

caller 関数（`main()`）は callee 関数（`scanf()`）の結果を必要とするため、呼び出し先は EAX レジスタに返します。

我々は、`CMP EAX, 1` (*CoMPare*) の指示によりそれをチェックします。つまり、EAX レジスタの値と1を比較します。

JNE 条件ジャンプが CMP 命令の後に続きます。JNE は *Jump if Not Equal* の略です。

したがって、EAX レジスタの値が1に等しくない場合、CPUは JNE オペランドに記述されているアドレス（この場合は `$LN2@main`）に実行を渡します。このアドレスに制御を渡すと、CPUは `printf()` を引数 `What you entered? Huh?` で実行します。しかし、すべてがうまくいけば、条件付きジャンプは取られず、別の `printf()` 呼び出しが`'You entered %d...'` と `x` の値を引数にとって実行されます。

この場合、2番目の `printf()` は実行されないため、その前に JMP があります（無条件ジャンプ）。2番目の `printf()` の後、戻り値0 を実装する `XOR EAX, EAX` 命令の直前に制御を渡します。

したがって、ある値を別の値と比較することは、通常、CMP/Jcc 命令ペアによって実装されると言えます *cc* は条件コードです。CMP は2つの値を比較し、プロセッサフラグ[76]を設定します。Jcc はこれらのフラグをチェックし、指定されたアドレスに制御を渡すかどうかを決定します。

これは逆説的に聞こえるかもしれませんが、CMP 命令は実際には SUB （減算）です。すべての算術命令は、CMP だけでなくプロセッサフラグを設定します。1と1を比較し、$1-1$ が0であるため、ZFフラグが設定されます（最後の結果が0であることを意味します）。オペランドが等しい場合を除いて、ZF は設定できません。JNE は ZF フラグのみをチェックし、設定されていない場合にジャンプします。JNEは実際にはJNZ（*Jump if Not Zero*）の同義語です。アセンブラは、JNE命令とJNZ命令の両方を同じオペコードに変換します。したがって、CMP 命令は SUB 命令で置き換えることができ、SUB が最初のオペランドの値を変更するという違いを除けば、ほとんどすべてが問題ありません。CMP は結果を保存しない *SUB* ですが、フラグに影響します。

MSVC: x86: IDA

IDAを実行してIDAを実行しようとします。ところで、初心者の方は、MSVCで/MD オプションを使用することをお勧めします。つまり、これらの標準関数はすべて実行可能ファイルにリンクされず、代わりに `MSVCR*.DLL` ファイルからインポートされます。したがって、どの標準関数が使用され、どこでどこが使用されているのかが分かりやすくなります。

IDA のコードを分析する際には、自分自身（と他者）のためにノートを残すことが非常に役に立ちます。例えば、この例を分析すると、エラーが発生した場合に JNZ がトリガーされることがわかります。カーソルをラベルに移動して「n」を押し、「エラー」に名前を変更することができます。別のラベルを作成し、「終了」にします。以下が私の環境での結果です。

```
.text:00401000 _main proc near
.text:00401000
.text:00401000 var_4 = dword ptr -4
```

---

[76] x86フラグは以下を参照: wikipedia

```
.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
```

これで、コードを少し理解しやすくなりました。しかし、すべての命令についてコメントするのは良い考えではありません。

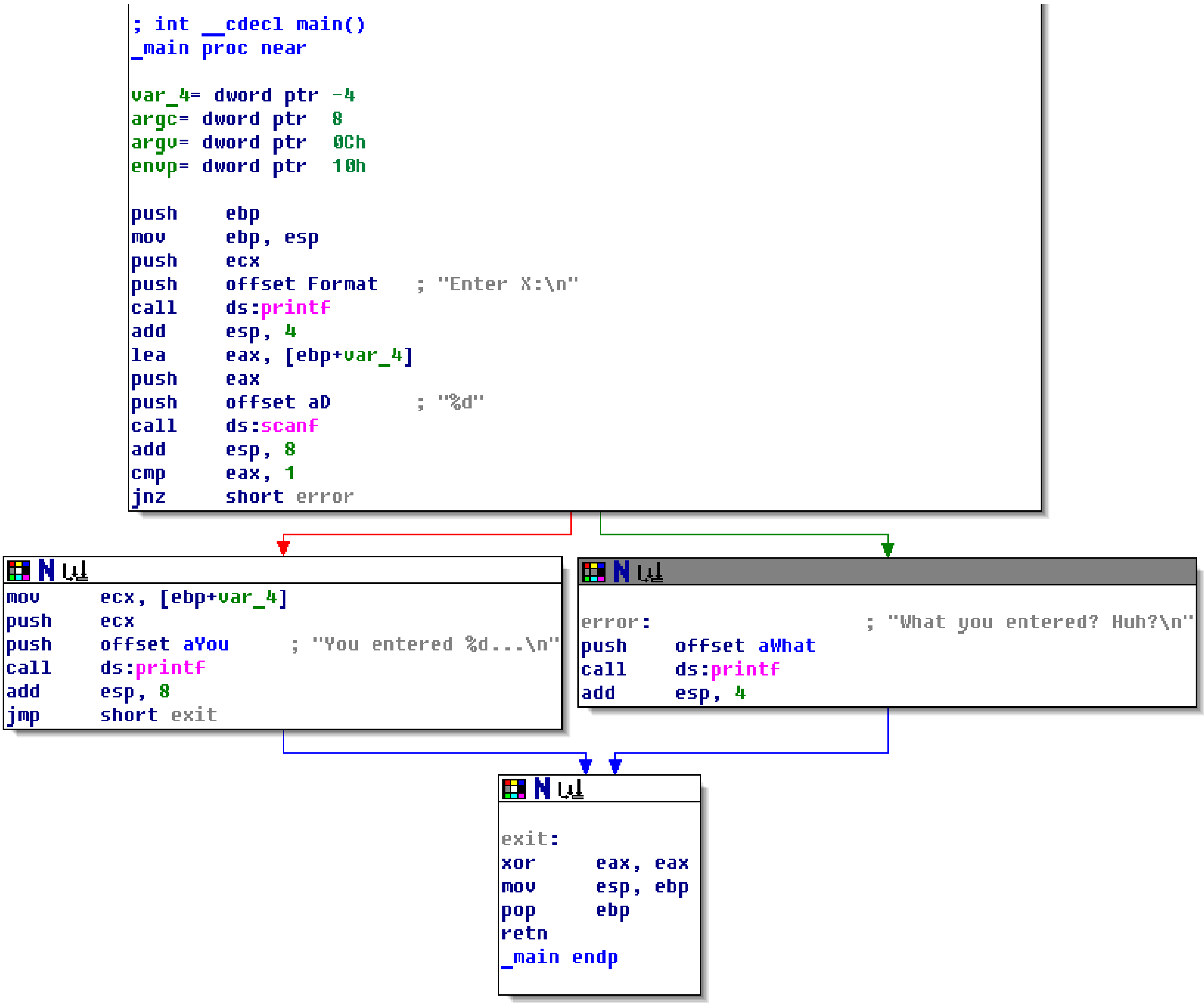
また、IDA の関数の一部を隠すこともできます。ブロックをマークするには、「–」を数値パッドに入力し、代わりに表示するテキストを入力します。

2つのブロックを隠して名前を付けましょう。

```
.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
```

以前に折りたたまれた部分を展開するには、数値パッドで「+」を使用します。

「スペース」を押すと、IDA が関数をグラフとして表示するのを見ることができます。



```
; int __cdecl main()
_main proc near

var_4= dword ptr -4
argc= dword ptr  8
argv= dword ptr  0Ch
envp= dword ptr  10h

push    ebp
mov     ebp, esp
push    ecx
push    offset Format   ; "Enter X:\n"
call    ds:printf
add     esp, 4
lea     eax, [ebp+var_4]
push    eax
push    offset aD       ; "%d"
call    ds:scanf
add     esp, 8
cmp     eax, 1
jnz     short error
```

```
mov     ecx, [ebp+var_4]
push    ecx
push    offset aYou     ; "You entered %d...\n"
call    ds:printf
add     esp, 8
jmp     short exit
```

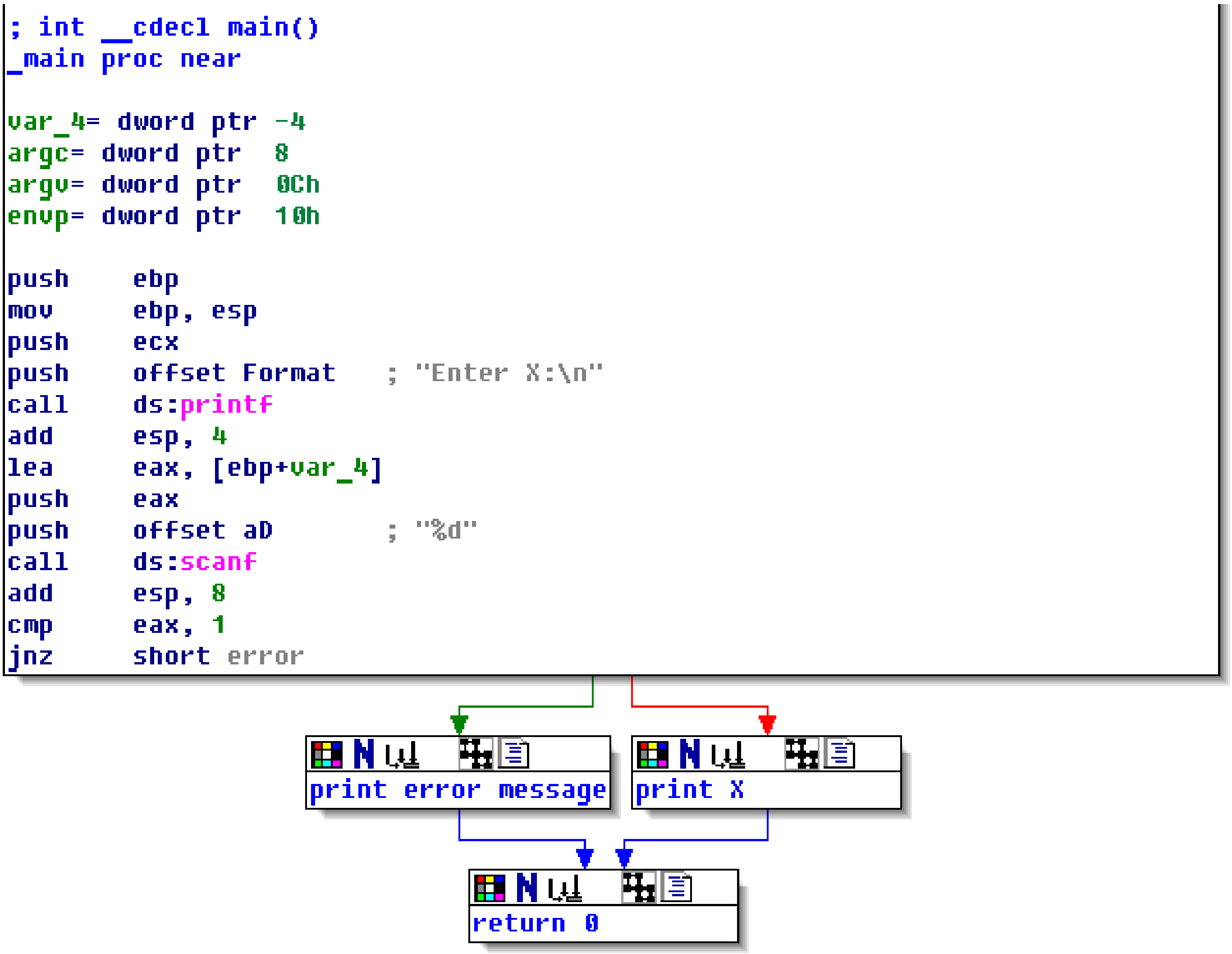
```
error:                  ; "What you entered? Huh?\n"
push    offset aWhat
call    ds:printf
add     esp, 4
```

```
exit:
xor     eax, eax
mov     esp, ebp
pop     ebp
retn
_main endp
```

図 1.17: Graph mode in IDA

各条件ジャンプの後、緑と赤の2つの矢印があります。緑の矢印は、ジャンプがトリガされた場合に実行されるブロックを指し、そうでない場合は赤を指します。

このモードでノードを折りたたみ、名前を付けることもできます（[q グループノード)。3つのブロックでやってみましょう。



```
; int __cdecl main()
_main proc near

var_4= dword ptr -4
argc= dword ptr  8
argv= dword ptr  0Ch
envp= dword ptr  10h

push    ebp
mov     ebp, esp
push    ecx
push    offset Format   ; "Enter X:\n"
call    ds:printf
add     esp, 4
lea     eax, [ebp+var_4]
push    eax
push    offset aD       ; "%d"
call    ds:scanf
add     esp, 8
cmp     eax, 1
jnz     short error
```

```
print error message
```

```
print X
```

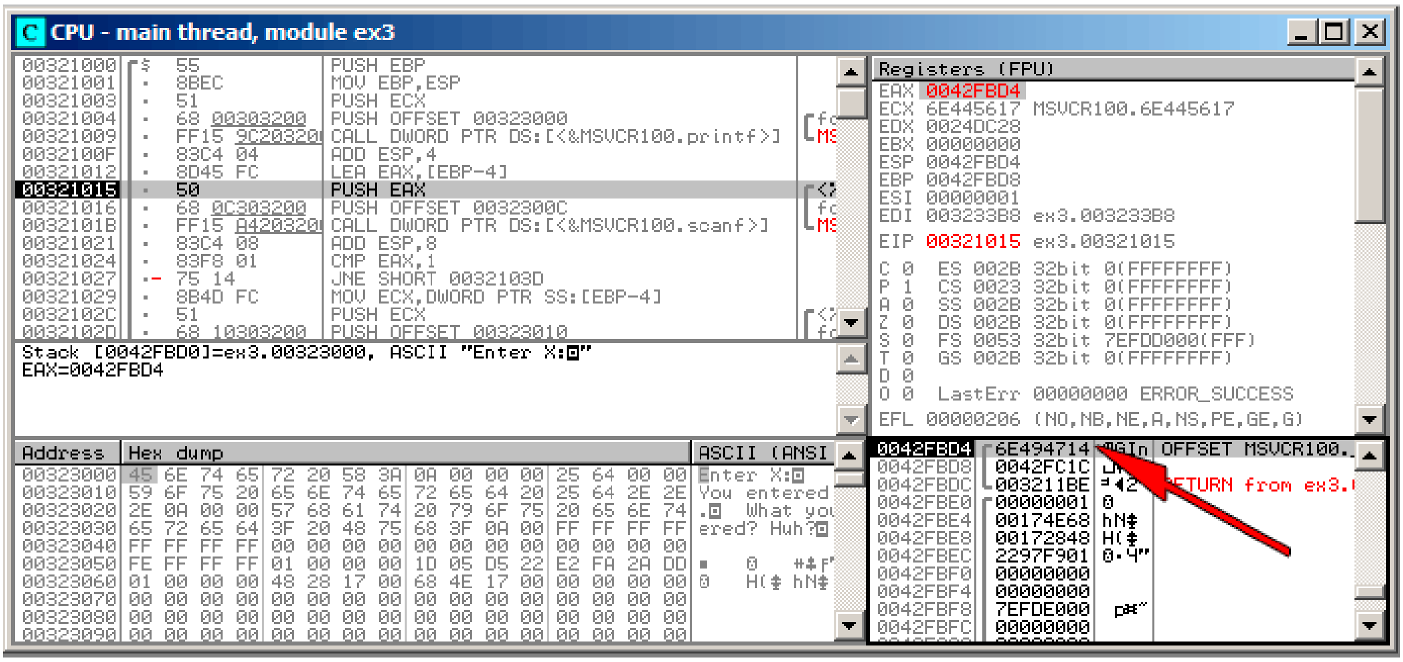
```
return 0
```

図 1.18: Graph mode in IDA with 3 nodes folded

それは非常に便利です。リバースエンジニアの仕事（および他の研究者の仕事）の非常に重要な部分は、彼らが扱う情報の量を減らすことであると言えます。

MSVC: x86 + OllyDbg

OllyDbg でプログラムをハックしようとして、scanf() が常にエラーなく動作するようにしましょう。ローカル変数のアドレスが scanf() に渡されると、変数には最初にいくつかのランダムなガベージが含まれます。この場合、0x6E494714 です。



```
00321000  $  55           PUSH EBP
00321001  .  8BEC         MOV EBP,ESP
00321003  .  51           PUSH ECX
00321004  .  68 00303200  PUSH OFFSET 00323000
00321009  .  FF15 9C20320 CALL DWORD PTR DS:[<&MSVCR100.printf>]
0032100F  .  83C4 04      ADD ESP,4
00321012  .  8D45 FC      LEA EAX,[EBP-4]
00321015  .  50           PUSH EAX
00321016  .  68 0C303200  PUSH OFFSET 0032300C
0032101B  .  FF15 A420320 CALL DWORD PTR DS:[<&MSVCR100.scanf>]
00321021  .  83C4 08      ADD ESP,8
00321024  .  83F8 01      CMP EAX,1
00321027  .- 75 14        JNE SHORT 0032103D
00321029  .  8B4D FC      MOV ECX,DWORD PTR SS:[EBP-4]
0032102C  .  51           PUSH ECX
0032102D  .  68 10303200  PUSH OFFSET 00323010
```

```
Stack [0042FBD0]=ex3.00323000, ASCII "Enter X:◘"
EAX=0042FBD4
```

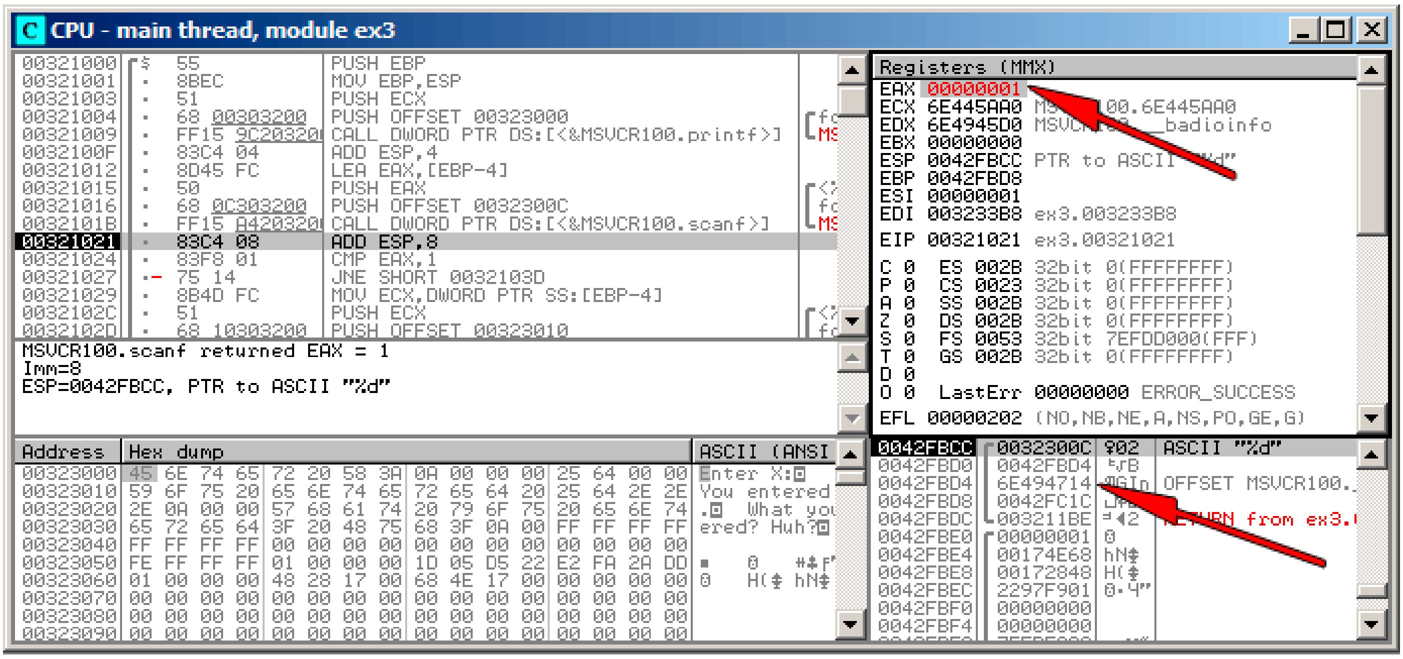
```
Registers (FPU)
EAX 0042FBD4
ECX 6E445617 MSVCR100.6E445617
EDX 0024DC28
EBX 00000000
ESP 0042FBD4
EBP 0042FBD8
ESI 00000001
EDI 003233B8 ex3.003233B8

EIP 00321015 ex3.00321015

C 0  ES 002B 32bit 0(FFFFFFFF)
P 1  CS 0023 32bit 0(FFFFFFFF)
A 0  SS 002B 32bit 0(FFFFFFFF)
Z 0  DS 002B 32bit 0(FFFFFFFF)
S 0  FS 0053 32bit 7EFDD000(FFF)
T 0  GS 002B 32bit 0(FFFFFFFF)
D 0
O 0  LastErr 00000000 ERROR_SUCCESS

EFL 00000206 (NO,NB,NE,A,NS,PE,GE,G)
```

```
Address   Hex dump                                         ASCII (ANSI
00323000  45 6E 74 65 72 20 58 3A 0A 00 00 00 25 64 00 00  Enter X:◘
00323010  59 6F 75 20 65 6E 74 65 72 65 64 20 25 64 2E 2E  You entered
00323020  2E 0A 00 00 57 68 61 74 20 79 6F 75 20 65 6E 74  .◘  What you
00323030  65 72 65 64 3F 20 48 75 68 3F 0A 00 FF FF FF FF  ered? Huh?◘
00323040  FF FF FF FF 00 00 00 00 00 00 00 00 00 00 00 00
00323050  FE FF FF FF 01 00 00 00 1D 05 D5 22 E2 FA 2A DD  ■   0   #‡Γ
00323060  01 00 00 00 48 28 17 00 68 4E 17 00 00 00 00 00  0   H(‡ hN‡
00323070  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
00323080  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
00323090  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
```

```
0042FBD4  6E494714  ▯GIn  OFFSET MSVCR100._
0042FBD8  0042FC1C
0042FBDC  003211BE        RETURN from ex3.
0042FBE0  00000001  0
0042FBE4  00174E68  hN‡
0042FBE8  00172848  H(‡
0042FBEC  2297F901  0·ч"
0042FBF0  00000000
0042FBF4  00000000
0042FBF8  7EFDE000   pæ˝
0042FBFC  00000000
```

図 1.19: OllyDbg: passing variable address into scanf()

scanf() が実行されている間、コンソールでは、「asdasd」のように、数字ではないものを入力します。scanf() は、エラーが発生したことを示す EAX が0で終了します。



```
00321000  $   55             PUSH EBP
00321001  .   8BEC           MOV EBP,ESP
00321003  .   51             PUSH ECX
00321004  .   68 00303200    PUSH OFFSET 00323000
00321009  .   FF15 9C20320   CALL DWORD PTR DS:[<&MSVCR100.printf>]
0032100F  .   83C4 04        ADD ESP,4
00321012  .   8D45 FC        LEA EAX,[EBP-4]
00321015  .   50             PUSH EAX
00321016  .   68 0C303200    PUSH OFFSET 0032300C
0032101B  .   FF15 A420320   CALL DWORD PTR DS:[<&MSVCR100.scanf>]
00321021  .   83C4 08        ADD ESP,8
00321024  .   83F8 01        CMP EAX,1
00321027  .-  75 14          JNE SHORT 0032103D
00321029  .   8B4D FC        MOV ECX,DWORD PTR SS:[EBP-4]
0032102C  .   51             PUSH ECX
0032102D  .   68 10303200    PUSH OFFSET 00323010
```

```
MSVCR100.scanf returned EAX = 1
Imm=8
ESP=0042FBCC, PTR to ASCII "%d"
```

```
Registers (MMX)
EAX 00000001
ECX 6E445AA0
EDX 6E4945D0 MSVC      __badioinfo
EBX 00000000
ESP 0042FBCC PTR to ASCII
EBP 0042FBD8
ESI 00000001
EDI 003233B8 ex3.003233B8

EIP 00321021 ex3.00321021

C 0  ES 002B 32bit 0(FFFFFFFF)
P 0  CS 0023 32bit 0(FFFFFFFF)
A 0  SS 002B 32bit 0(FFFFFFFF)
Z 0  DS 002B 32bit 0(FFFFFFFF)
S 0  FS 0053 32bit 7EFDD000(FFF)
T 0  GS 002B 32bit 0(FFFFFFFF)
D 0
O 0  LastErr 00000000 ERROR_SUCCESS

EFL 00000202 (NO,NB,NE,A,NS,PO,GE,G)
```

```
Address   Hex dump                                         ASCII (ANSI
00323000  45 6E 74 65 72 20 58 3A 0A 00 00 00 25 64 00 00  Enter X:
00323010  59 6F 75 20 65 6E 74 65 72 65 64 20 25 64 2E 2E  You entered
00323020  2E 0A 00 00 57 68 61 74 20 79 6F 75 20 65 6E 74  .  What you
00323030  65 72 65 64 3F 20 48 75 68 3F 0A 00 FF FF FF FF  ered? Huh?
00323040  FF FF FF FF 00 00 00 00 00 00 00 00 00 00 00 00
00323050  FE FF FF FF 01 00 00 00 1D 05 D5 22 E2 FA 2A DD
00323060  01 00 00 00 48 28 17 00 68 4E 17 00 00 00 00 00
00323070  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
00323080  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
00323090  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
```

```
0042FBCC  0032300C  ASCII "%d"
0042FBD0  0042FBD4
0042FBD4  6E494714       OFFSET MSVCR100._
0042FBD8  0042FC1C
0042FBDC  003211BE       from ex3.
0042FBE0  00000001
0042FBE4  00174E68
0042FBE8  00172848
0042FBEC  2297F901
0042FBF0  00000000
0042FBF4  00000000
```

図 1.20: OllyDbg: scanf() returning error

また、スタック内のローカル変数をチェックし、変更されていないことに注意してください。実際、scanf() は何を書いていますか？ゼロを返す以外は何もしませんでした。

私たちのプログラムを「ハックする」ようにしましょう。EAX を右クリックし、オプションの中に「Set to 1」があります。これが必要なものです。

EAX には1があるので、以下のチェックを意図どおりに実行し、printf() は変数の値をスタックに出力します。

プログラム（F9）を実行すると、コンソールウィンドウで次のように表示されます。

Listing 1.81: console window

```
Enter X:
asdasd
You entered 1850296084...
```

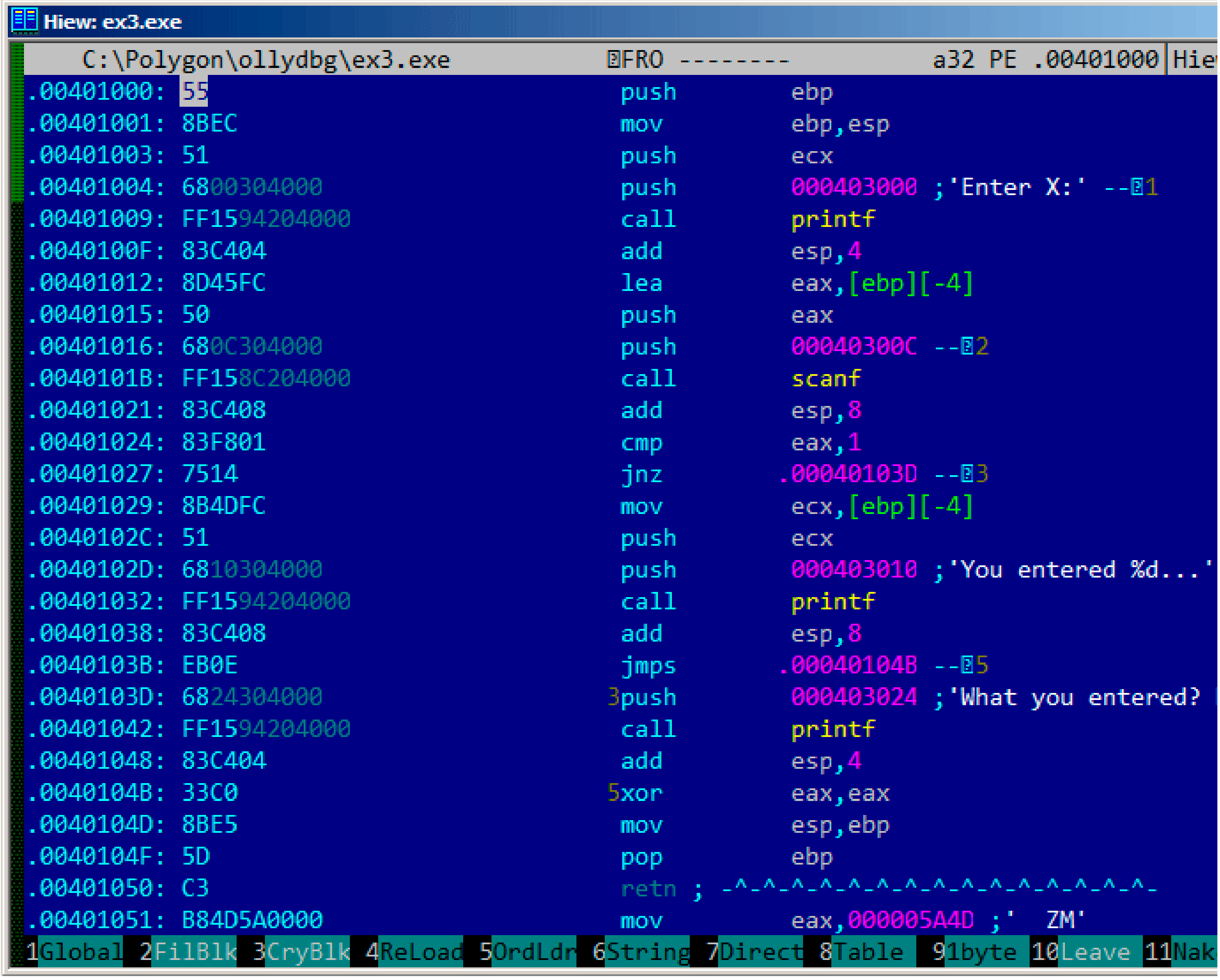
実際、1850296084はスタック（0x6E494714）の数値を10進表現したものです！

MSVC: x86 + Hiew

これは、実行可能ファイルのパッチ適用の簡単な例としても使用できます。実行可能ファイルにパッチを適用して、入力内容にかかわらずプログラムが常に入力を出力するようにすることがあります。

実行可能ファイルが外部の `MSVCR*.DLL`（つまり/MD オプション付き）[77] に対してコンパイルされていると仮定すると、`.text` セクションの先頭に `main()` 関数があります。Hiewで実行可能ファイルを開き、`.text` セクションの先頭を見つけましょう（Enter、F8、F6、Enter、Enter)。

以下のように見えます。



```
Hiew: ex3.exe
   C:\Polygon\ollydbg\ex3.exe                 ⊡FRO --------          a32 PE .00401000 Hie
.00401000: 55                                  push        ebp
.00401001: 8BEC                                mov         ebp,esp
.00401003: 51                                  push        ecx
.00401004: 6800304000                          push        000403000 ;'Enter X:' --⊡1
.00401009: FF1594204000                        call        printf
.0040100F: 83C404                              add         esp,4
.00401012: 8D45FC                              lea         eax,[ebp][-4]
.00401015: 50                                  push        eax
.00401016: 680C304000                          push        00040300C --⊡2
.0040101B: FF158C204000                        call        scanf
.00401021: 83C408                              add         esp,8
.00401024: 83F801                              cmp         eax,1
.00401027: 7514                                jnz        .00040103D --⊡3
.00401029: 8B4DFC                              mov         ecx,[ebp][-4]
.0040102C: 51                                  push        ecx
.0040102D: 6810304000                          push        000403010 ;'You entered %d...'
.00401032: FF1594204000                        call        printf
.00401038: 83C408                              add         esp,8
.0040103B: EB0E                                jmps       .00040104B --⊡5
.0040103D: 6824304000                         3push        000403024 ;'What you entered?
.00401042: FF1594204000                        call        printf
.00401048: 83C404                              add         esp,4
.0040104B: 33C0                               5xor         eax,eax
.0040104D: 8BE5                                mov         esp,ebp
.0040104F: 5D                                  pop         ebp
.00401050: C3                                  retn ; -^-^-^-^-^-^-^-^-^-^-^-^-^-^-^-
.00401051: B84D5A0000                          mov         eax,000005A4D ;'  ZM'
 1Global  2FilBlk  3CryBlk  4ReLoad  5OrdLdr  6String  7Direct  8Table   91byte  10Leave  11Nak
```

図 1.21: Hiew: main() function

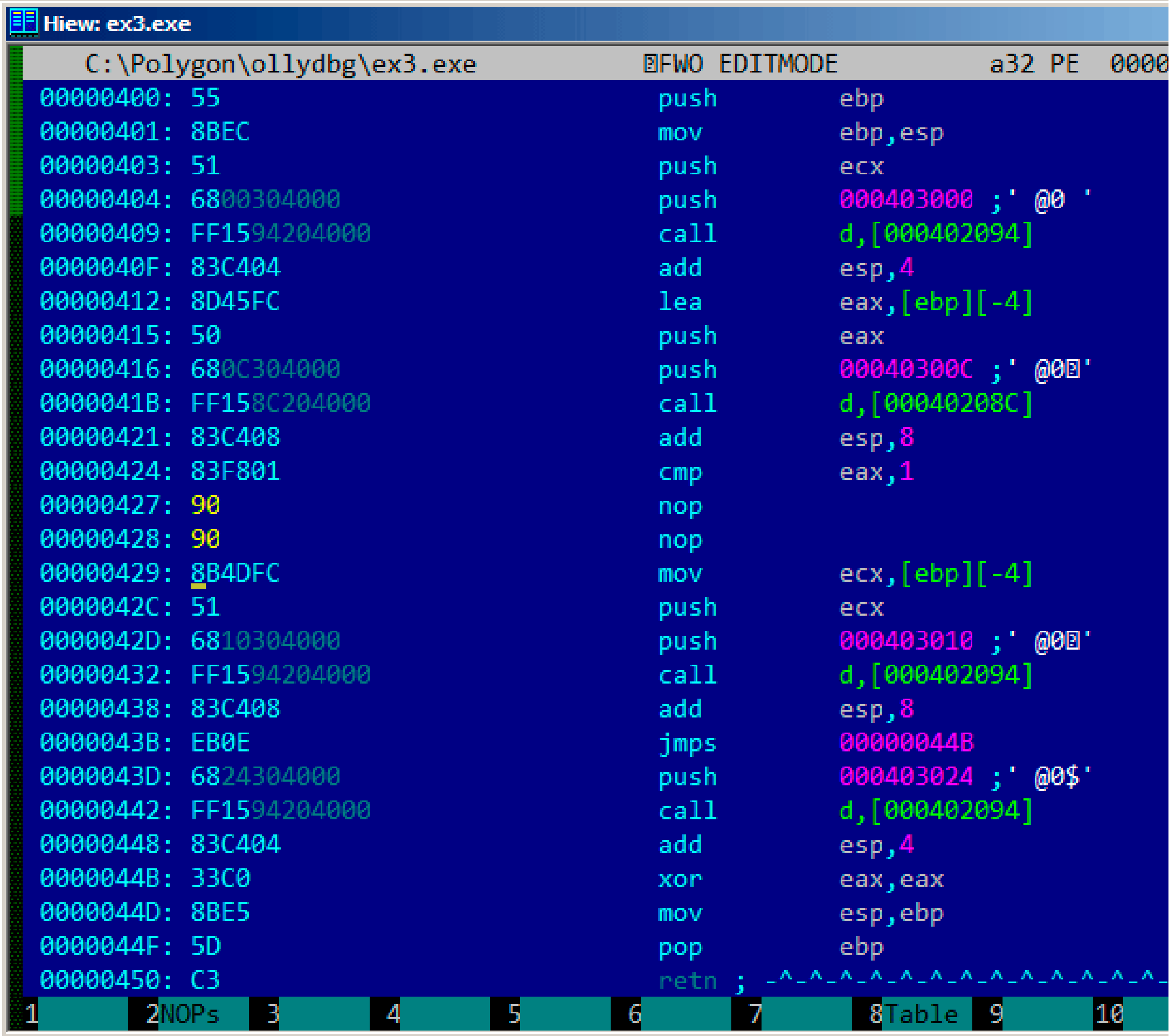
HiewはASCIIZ[78]文字列を検索し、インポートされた関数の名前と同様に表示します。

---

[77] 「ダイナミックリンク」とも呼ばれる

[78] ASCII Zero ( )

カーソルを.00401027 番地（ここでバイパスする JNZ 命令がある場所）に移動し、F3を押し、「9090」(2つのNOPを意味する）と入力します。



```
    C:\Polygon\ollydbg\ex3.exe                  □FWO EDITMODE          a32 PE  0000
 00000400: 55                                   push      ebp
 00000401: 8BEC                                 mov       ebp,esp
 00000403: 51                                   push      ecx
 00000404: 6800304000                           push      000403000 ;' @0 '
 00000409: FF1594204000                         call      d,[000402094]
 0000040F: 83C404                               add       esp,4
 00000412: 8D45FC                               lea       eax,[ebp][-4]
 00000415: 50                                   push      eax
 00000416: 680C304000                           push      00040300C ;' @0□'
 0000041B: FF158C204000                         call      d,[00040208C]
 00000421: 83C408                               add       esp,8
 00000424: 83F801                               cmp       eax,1
 00000427: 90                                   nop
 00000428: 90                                   nop
 00000429: 8B4DFC                               mov       ecx,[ebp][-4]
 0000042C: 51                                   push      ecx
 0000042D: 6810304000                           push      000403010 ;' @0□'
 00000432: FF1594204000                         call      d,[000402094]
 00000438: 83C408                               add       esp,8
 0000043B: EB0E                                 jmps      00000044B
 0000043D: 6824304000                           push      000403024 ;' @0$'
 00000442: FF1594204000                         call      d,[000402094]
 00000448: 83C404                               add       esp,4
 0000044B: 33C0                                 xor       eax,eax
 0000044D: 8BE5                                 mov       esp,ebp
 0000044F: 5D                                   pop       ebp
 00000450: C3                                   retn ; -^-^-^-^-^-^-^-^-^-^-^-^-^-
 1        2NOPs   3        4        5        6        7        8Table  9        10
```

図 1.22: Hiew: replacing JNZ with two NOPs

その後、F9（更新）を押します。これで、実行可能ファイルがディスクに保存されます。私たちが望むように動作します。

2つのNOPはおそらく最も美しいアプローチではありません。この命令をパッチする別の方法は、第2オペコードバイトに0を書き込むことであり（jump offset)、JNZ は常に次の命令にジャンプします。

また、最初のバイトを EB で置き換え、2番目のバイト（jump offset ）には触れないでください。私たちは常に無条件のジャンプを得るでしょう。この場合、エラーメッセージは入力に関係なく毎回表示されます。

MSVC: x64

MSVC: x64

ここではx86-64の32ビットである *int* 型変数について説明しているので、ここではレジスタの32ビット部分（E-を前に付ける）も同様に使用されています。ただし、ポインタを使用している間は、64ビットのレジスタ部分が使用され、先頭に R-が付きます。

Listing 1.82: MSVC 2012 x64

```
_DATA   SEGMENT
$SG2924 DB      'Enter X:', 0aH, 00H
$SG2926 DB      '%d', 00H
$SG2927 DB      'You entered %d...', 0aH, 00H
$SG2929 DB      'What you entered? Huh?', 0aH, 00H
_DATA   ENDS

_TEXT   SEGMENT
x$ = 32
main    PROC
$LN5:
        sub     rsp, 56
        lea     rcx, OFFSET FLAT:$SG2924 ; 'Enter X:'
        call    printf
        lea     rdx, QWORD PTR x$[rsp]
        lea     rcx, OFFSET FLAT:$SG2926 ; '%d'
        call    scanf
        cmp     eax, 1
        jne     SHORT $LN2@main
        mov     edx, DWORD PTR x$[rsp]
        lea     rcx, OFFSET FLAT:$SG2927 ; 'You entered %d...'
        call    printf
        jmp     SHORT $LN1@main
$LN2@main:
        lea     rcx, OFFSET FLAT:$SG2929 ; 'What you entered? Huh?'
        call    printf
$LN1@main:
        ; 0をリターン
        xor     eax, eax
        add     rsp, 56
        ret     0
main    ENDP
_TEXT   ENDS
END
```

ARM

ARM: 最適化 Keil 6/2013 (Thumbモード)

Listing 1.83: 最適化 Keil 6/2013 (Thumbモード)

```
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
```

ここでの新しい命令は CMP とBEQ[79]です。

CMP は同じ名前のx86命令に似ていますが、他の引数から引数の1つを減算し、必要に応じて条件フラグを更新します。

オペランドが互いに等しい場合、または最後の計算の結果が0の場合、またはZフラグが1の場合、BEQは別のアドレスにジャンプします。これはx86では JZ として動作します。

それ以外はすべてシンプルです。実行フローが2つの分岐に分岐した後、関数の戻り値として0が R0 に書き込まれた時点で分岐が収束し、関数が終了します。

ARM64

Listing 1.84: 非最適化 GCC 4.9.1 ARM64

```
.LC0:
        .string "Enter X:"
.LC1:
        .string "%d"
.LC2:
        .string "You entered %d...\n"
.LC3:
        .string "What you entered? Huh?"
f6:
; スタックフレームにFPとLRと保存
        stp     x29, x30, [sp, -32]!
; スタックフレームを設定(FP=SP)
        add     x29, sp, 0
; "Enter X:"文字列へのポインタをロード
        adrp    x0, .LC0
        add     x0, x0, :lo12:.LC0
        bl      puts
; "%d"文字列へのポインタをロード
        adrp    x0, .LC1
        add     x0, x0, :lo12:.LC1
; ローカルスタックにある変数xのアドレスを計算
        add     x1, x29, 28
        bl      __isoc99_scanf
; W0にscanf()の戻り値が入っている
; チェックする
        cmp     w0, 1
; BNEはイコールでない場合に分岐する
; だから、W0<>0の場合、L2にジャンプする
        bne     .L2
; W0=1の場合、エラーなし
; ローカルスタックからxの値をロードする
        ldr     w1, [x29,28]
; "You entered %d...\n"文字列へのポインタをロードする
        adrp    x0, .LC2
        add     x0, x0, :lo12:.LC2
        bl      printf
; "What you entered? Huh?"文字列を表示するコードをスキップする
        b       .L3
.L2:
; "What you entered? Huh?"文字列へのポインタをロードする
        adrp    x0, .LC3
        add     x0, x0, :lo12:.LC3
        bl      puts
.L3:
; 0をリターン
        mov     w0, 0
; FPとLRを元に戻す:
        ldp     x29, x30, [sp], 32
        ret
```

この場合のコードフローは、CMP/BNE （Branch if Not Equal）命令のペアを使用して分岐します。

---

[79] (PowerPC, ARM) Branch if Equal

MIPS

Listing 1.85: 最適化 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
```

`scanf()` は、その作業の結果をレジスタ $V0 に返します。アドレス0x004006E4は、$V0 の値と $V1（1は $V1 以前の0x004006DCに格納されています）と比較することでチェックされます。BEQ は「Branch Equal」の略です。2つの値が等しい場合（すなわち、成功した場合)、アドレス0x0040070Cにジャンプします。

練習問題

見てきたように、JNE/JNZ 命令は JE/JZ 命令に簡単に置き換えることができます（BNE by BEQ またはその逆)。しかし、基本ブロックも入れ替える必要があります。いくつかの例でこれを試してください。

第1.9.5節練習問題

- http://challenges.re/53

第1.10節渡された引数にアクセスする

さて、callerの関数が引数をcallee側にスタック経由で渡していることが分かりました。しかし、callee関数はどうやって引数にアクセスするのでしょうか？

Listing 1.86: simple example

```
#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;
};
```

第1.10.1節x86

MSVC

コンパイルして得られるものを次に示します（MSVC 2010 Express）。

Listing 1.87: MSVC 2010 Express

```
_TEXT   SEGMENT
_a$ = 8         ; size = 4
_b$ = 12        ; size = 4
_c$ = 16        ; size = 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 ; 3番目の引数
        push    2 ; 2番目の引数
        push    1 ; 1番目の引数
        call    _f
        add     esp, 12
        push    eax
        push    OFFSET $SG2463 ; '%d', 0aH, 00H
        call    _printf
        add     esp, 8
        ; 0をリターン
        xor     eax, eax
        pop     ebp
        ret     0
_main   ENDP
```

`main()` 関数は3つの数値をスタックにプッシュし、`f(int,int,int)` を呼び出すことがわかります。

`f()` 内の引数アクセスは、ローカル変数と同じ方法で `_a$ = 8` のようなマクロの助けを借りて構成されますが、正のオフセット（プラスで扱われます）を持ちます。したがって、`_a$` マクロを EBP レジスタの値に追加することによってstack frameの外側を処理しています。

次に、a の値が EAX に格納されます。IMUL 命令実行後、EAX の値は EAX の値と _b の内容のproductです。

その後、ADD は `_c` の値を EAX に追加します。

EAX の値は移動する必要はありません。すでに存在している必要があります。callerに戻ると、EAX 値をとり、`printf()` の引数として使用します。

MSVC + OllyDbg

これを OllyDbg で説明しましょう。最初の引数（最初の引数）を使用する `f()` の最初の命令をトレースすると、EBP が赤い四角でマークされたstack frameを指していることがわかります。

stack frameの最初の要素は EBP のセーブされた値であり、2番目の要素はRAであり、3番目の要素は最初の関数の引数であり、2番目と3番目の要素です。

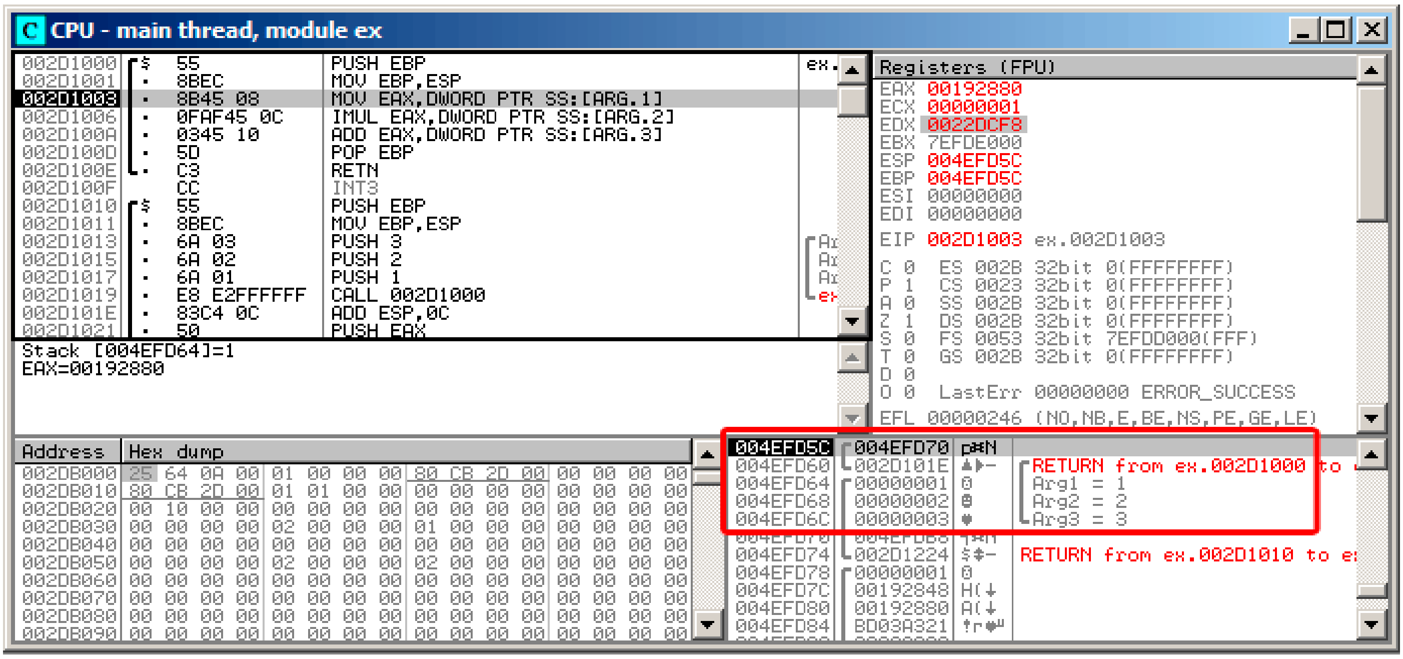
最初の関数引数にアクセスするには、EBP にちょうど8（2つの32ビットワード）を追加する必要があります。

OllyDbg はこれを知っているので、

「RETURN from」や 「Arg1 = ...」などのスタック要素にコメントを追加しました。

注意：関数の引数は、関数のスタックフレームのメンバーではなく、むしろcaller関数のスタックフレームのメンバーです。

したがって、OllyDbg は別のスタックフレームのメンバーとして 「Arg」要素をマークしました。



```
002D1000 ┌$  55          PUSH EBP
002D1001 │.  8BEC        MOV EBP,ESP
002D1003 │.  8B45 08     MOV EAX,DWORD PTR SS:[ARG.1]
002D1006 │.  0FAF45 0C   IMUL EAX,DWORD PTR SS:[ARG.2]
002D100A │.  0345 10     ADD EAX,DWORD PTR SS:[ARG.3]
002D100D │.  5D          POP EBP
002D100E └.  C3          RETN
002D100F     CC          INT3
002D1010 ┌$  55          PUSH EBP
002D1011 │.  8BEC        MOV EBP,ESP
002D1013 │.  6A 03       PUSH 3
002D1015 │.  6A 02       PUSH 2
002D1017 │.  6A 01       PUSH 1
002D1019 │.  E8 E2FFFFFF CALL 002D1000
002D101E │.  83C4 0C     ADD ESP,0C
002D1021 │.  50          PUSH EAX
Stack [004EFD64]=1
EAX=00192880
```

```
Registers (FPU)
EAX 00192880
ECX 00000001
EDX 0022DCF8
EBX 7EFDE000
ESP 004EFD5C
EBP 004EFD5C
ESI 00000000
EDI 00000000

EIP 002D1003 ex.002D1003

C 0  ES 002B 32bit 0(FFFFFFFF)
P 1  CS 0023 32bit 0(FFFFFFFF)
A 0  SS 002B 32bit 0(FFFFFFFF)
Z 1  DS 002B 32bit 0(FFFFFFFF)
S 0  FS 0053 32bit 7EFDD000(FFF)
T 0  GS 002B 32bit 0(FFFFFFFF)
D 0
O 0  LastErr 00000000 ERROR_SUCCESS

EFL 00000246 (NO,NB,E,BE,NS,PE,GE,LE)
```

```
Address  Hex dump
002DB000 25 64 0A 00 01 00 00 00 80 CB 2D 00 00 00 00 00
002DB010 80 CB 2D 00 01 01 00 00 00 00 00 00 00 00 00 00
002DB020 00 10 00 00 00 00 00 00 00 00 00 00 00 00 00 00
002DB030 00 00 00 00 02 00 00 00 01 00 00 00 00 00 00 00
002DB040 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
002DB050 00 00 00 00 02 00 00 00 02 00 00 00 00 00 00 00
002DB060 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
002DB070 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
002DB080 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
002DB090 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
```

```
004EFD5C ┌004EFD70 p≠N
004EFD60 └002D101E ▲▶-  ┌RETURN from ex.002D1000 to
004EFD64 ┌00000001 ☺    │Arg1 = 1
004EFD68 │00000002 ☻    │Arg2 = 2
004EFD6C │00000003 ♥    └Arg3 = 3
004EFD74 └002D1224 $‡-  RETURN from ex.002D1010 to e
004EFD78 ┌00000001 ☺
004EFD7C │00192848 H(↓
004EFD80 │00192880 A(↓
004EFD84 │BD03A321 !r♥
```

図 1.23: OllyDbg: inside of `f()` function

GCC

GCC 4.4.1で同じものをコンパイルし、IDA の結果を見てみましょう。

Listing 1.88: 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] ; 1番目の引数
        imul    eax, [ebp+arg_4] ; 2番目の引数
        add     eax, [ebp+arg_8] ; 3番目の引数
        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 ; 3番目の引数
        mov     [esp+10h+var_C], 2 ; 2番目の引数
        mov     [esp+10h+var_10], 1 ; 1番目の引数
        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
```

結果はほぼ同じで、以前に説明したいくつかの小さな違いがあります。

stack pointerは2つの関数呼び出し（fとprintf）の後にセットバックされません。最後から2番目の LEAVE 命令（**??** on page ??）命令が最後にこれを処理するためです。

第1.10.2節x64

この話はx86-64では少し違っています。関数の引数（最初の4つまたは最初の6つ）はレジスタに渡されます。つまり、calleeはレジスタからレジスタを読み込みます。

MSVC

最適化 MSVC:

Listing 1.89: 最適化 MSVC 2012 x64

```
$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 - 1番目の引数
        ; EDX - 2番目の引数
        ; R8D - 3番目の引数
        imul    ecx, edx
        lea     eax, DWORD PTR [r8+rcx]
        ret     0
f       ENDP
```

見てわかるように、コンパクトな関数 `f()` はすべての引数をレジスタから取ります。

ここでの LEA 命令は加算に使用され、明らかにコンパイラは ADD よりも速いと考えました。

LEA は、第1および第3の `f()` 引数を準備するために `main()` 関数でも使用されます。コンパイラは、MOV 命令を使用してレジスタに値をロードする通常の方法よりも速く動作すると判断する必要があります。

非最適化MSVCの出力を見てみましょう。

Listing 1.90: MSVC 2012 x64

```
f               proc near

; シャドースペース
arg_0           = dword ptr  8
arg_8           = dword ptr  10h
arg_10          = dword ptr  18h

                ; ECX - 1番目の引数
                ; EDX - 2番目の引数
                ; R8D - 3番目の引数
                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 ; 3番目の引数
                mov     edx, 2 ; 2番目の引数
                mov     ecx, 1 ; 1番目の引数
                call    f
                mov     edx, eax
                lea     rcx, $SG2931    ; "%d\n"
                call    printf

                ; 0をリターン
                xor     eax, eax
                add     rsp, 28h
                retn
main            endp
```

レジスタからの3つの引数は何らかの理由でスタックに保存されるため、ややこしいことになっています。

これは”シャドウスペース”と呼ばれます。[80] すべてのWin64は、そこにある4つのレジスタ値をすべて保存することができます（必須ではありません）。これは2つの理由で行われます。1）入力引数にレジスタ全体（または4つのレジスタ）を割り当てるのはあまりにも贅沢なので、スタック経由でアクセスされます。2）デバッガはブレークで関数の引数をどこに見つけるか常に認識しています。[81]

だから、大規模な関数の中には、実行中にそれらを使用したい場合、入力引数を「シャドウスペース」に保存することができますが、私たちのような小さな関数ではそうでないかもしれません。

スタックに「シャドウスペース」を割り当てるのはcallerの責任です。

GCC

最適化 GCCはまあまあわかりやすいコードを生成します。

Listing 1.91: 最適化 GCC 4.4.6 x64

```
f:
        ; EDI - 1番目の引数
        ; ESI - 2番目の引数
        ; EDX - 3番目の引数
```

---

[80] MSDN

[81] MSDN

```
        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  ; 渡されたベクトルレジスタの数
        call    printf
        xor     eax, eax
        add     rsp, 8
        ret
```

非最適化 GCC:

Listing 1.92: GCC 4.4.6 x64

```
f:
        ; EDI - 1番目の引数
        ; ESI - 2番目の引数
        ; EDX - 3番目の引数
        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  ; 渡されたベクトルレジスタの数
        call    printf
        mov     eax, 0
        leave
        ret
```

System V *NIX ([Michael Matz, Jan Hubicka, Andreas Jaeger, Mark Mitchell, *System V Application Binary Interface. AMD64 Architecture Processor Supplement*, (2013)] [82]) には「シャドースペース」の要件はありませんが、calleeはレジスタが不足している場合には引数をどこかに保存します。

GCC: intの代わりのuint64_t

私たちの例は32ビットintで動作するため、32ビットのレジスタが使用されています（E-が前に付いています）。

64ビット値を使用するためには少し変更する必要があります。

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

uint64_t f (uint64_t a, uint64_t b, uint64_t c)
```

---

[82] 以下で利用可能 https://software.intel.com/sites/default/files/article/402129/mpx-linux64-abi.pdf

```
{
        return a*b+c;
};

int main()
{
        printf ("%lld\n", f(0x1122334455667788,
                            0x1111111122222222,
                            0x3333333344444444));
        return 0;
};
```

Listing 1.93: 最適化 GCC 4.4.6 x64

```
f       proc near
        imul    rsi, rdi
        lea     rax, [rdx+rsi]
        retn
f       endp

main    proc near
        sub     rsp, 8
        mov     rdx, 3333333344444444h ; 3番目の引数
        mov     rsi, 1111111122222222h ; 2番目の引数
        mov     rdi, 1122334455667788h ; 1番目の引数
        call    f
        mov     edi, offset format ; "%lld\n"
        mov     rsi, rax
        xor     eax, eax ; 渡されたベクトルレジスタの数
        call    _printf
        xor     eax, eax
        add     rsp, 8
        retn
main    endp
```

コードは同じですが、今回はフルサイズのレジスタ（R- が前に付いています）が使用されています。

第1.10.3節ARM

非最適化 **Keil 6/2013 (ARMモード)**

```
.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}
```

`main()` 関数は他の2つの関数を呼び出し、3つの値が最初の関数に渡されます（`f()`）。

前述のように、ARMでは最初の4つの値が通常最初の4つのレジスタ（R0-R3 ）に渡されます。

`f()` 関数は、最初の3つのレジスタ（R0-R2 ）を引数として使用します。

MLA（*Multiply Accumulate*）命令は最初の2つのオペランド（R3 と R1 ）を乗算し、3番目のオペランド（R2 ）を積に加算し、その結果をゼロ関数（R0 ）に格納します。

一度に乗算と加算を同時に行うの（*Fused multiply?add*）は非常に便利な操作です。ところで、SIMD[83] にFMA命令が登場する前に、x86にそのような命令はありませんでした。

最初の `MOV R3, R0` 命令は明らかに冗長です（ここでは単一の `MLA` 命令を代わりに使用できます）。これは最適化されないコンパイルであるため、コンパイラは最適化していません。

`BX` 命令は、制御を LRレジスタに格納されているアドレスに戻し、必要に応じてプロセッサモードをThumbからARMに、またはその逆に切り替えます。これは、関数 `f()` がどのような種類のコード（ARMまたはThumb）から認識されていないため、必要な場合があります。したがって、Thumbコードから呼び出された場合、`BX` は呼び出し元の関数に制御を戻すだけでなく、プロセッサーモードをThumbに切り替えます。ARMコード ([*ARM(R) Architecture Reference Manual, ARMv7-A and ARMv7-R edition*, (2012)A2.3.2]) から関数が呼び出されているかどうかを切り替えます。

最適化 **Keil 6/2013 (ARMモード)**

```
.text:00000098              f
.text:00000098 91 20 20 E0       MLA     R0, R1, R0, R2
.text:0000009C 1E FF 2F E1       BX      LR
```

Keilコンパイラによって完全最適化モード（-O3 ）でコンパイルされた `f()` 関数があります。

`MOV` 命令は最適化され（または縮小され）、`MLA` はすべての入力レジスタを使用し、結果を `R0` に配置します。

最適化 **Keil 6/2013 (Thumbモード)**

```
.text:0000005E 48 43          MULS    R0, R1
.text:00000060 80 18          ADDS    R0, R0, R2
.text:00000062 70 47          BX      LR
```

Thumbモードでは `MLA` 命令を使用できないため、コンパイラはこれら2つの演算（乗算と加算）を別々に実行するコードを生成します。

最初に、`MULS` 命令は `R0` に `R1` を掛けて、結果をレジスタ `R0` に残します。2番目の命令（`ADDS`）は結果と `R2` を加算して結果をレジスタ `R0` に残します。

ARM64

最適化 **GCC (Linaro) 4.9**

ここでのすべては簡単です。`MADD` は乗算/加算を一緒に行う命令です（既に見た `MLA` に似ています）。3つの引数はすべて、Xレジスタの32ビット部分に渡されます。実際、引数の型は32ビット *int* です。結果は `W0` に返されます。

Listing 1.94: 最適化 GCC (Linaro) 4.9

```
f:
        madd    w0, w0, w1, w2
        ret

main:
; FPとLRをスタックフレームに保存
        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
; 0をリターン
        mov     w0, 0
```

---

[83] wikipedia

```
; FPとLRを元に戻す
        ldp     x29, x30, [sp], 16
        ret

.LC7:
        .string "%d\n"
```

すべてのデータ型を64ビット `uint64_t` に拡張してテストしましょう：

```
#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"
```

`f()` 関数は同じで、64ビットのXレジスタ全体が使用されます。長い64ビットの値は、レジスタごとにロードされます。これについては、以下で説明します：**??** on page ??

非最適化 GCC (Linaro) 4.9

非最適化コンパイラはより冗長です。

```
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
```

コードは、この関数の誰か（または何か）が W0...W2 レジスタを使用する必要がある場合に、その入力引数をローカルスタックに保存します。これにより、元の関数の引数を上書きすることが防止されます。これは、将来必要になる可能性があります。

これは、レジスタセーブエリアと呼ばれます。(([*Procedure Call Standard for the ARM 64-bit Architecture (AArch64)*, (2013)][84])) しかし、呼び出し先関数はそれらを保存する義務はありません。これは、「シャドースペース」(1.10.2 on page 99) に多少似ています。

GCC 4.9を最適化すると、なぜこの引数はコードを保存しなくなったのでしょうか？これはいくつかの追加の最適化作業を行い、関数の引数がこの先に必要ではなく、レジスタ W0...W2 が使用されないと結論付けたためです。

また、単一の MADD の代わりに MUL/ADD 命令ペアがあります。

第1.10.4節MIPS

Listing 1.95: 最適化 GCC 4.4.5

```
.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, $v0     ; 分岐遅延スロット
; 戻り値は$v0に格納される
.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)
; cを設定
.text:00000024         li      $a2, 3
; aを設定
.text:00000028         li      $a0, 1
.text:0000002C         jal     f
; bを設定
.text:00000030         li      $a1, 2            ; 分岐遅延スロット
; 結果は$v0にある
.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
; take result of f()関数の結果を取得しprintf()の2番目の引数に渡す
.text:00000048         move    $a1, $v0          ; 分岐遅延スロット
.text:0000004C         lw      $ra, 0x20+var_4($sp)
.text:00000050         move    $v0, $zero
.text:00000054         jr      $ra
.text:00000058         addiu   $sp, 0x20 ; 分岐遅延スロット
```

最初の4つの関数引数は、A-が前に付いた4つのレジスタに渡されます。

MIPSには、HIとLOの2つの特殊レジスタがあり、MULT 命令の実行中に乗算の64ビット結果が格納されます。

これらのレジスタは、MFLO および MFHI 命令を使用することによってのみアクセスできます。ここでは MFLO は乗算結果の低部分をとり、それを $V0 に格納します。そのため、乗算結果の上位32ビット部分が削除されます (HIレジスタの内容は使用されません)。確かに、ここでは32ビットの *int* データ型を扱います。

最後に、ADDU（「Add Unsigned」）は3番目の引数の値を結果に加えます。

---

[84] 以下で利用可能 http://go.yurichev.com/17287

MIPSには、ADD と ADDU の2種類の加算命令があります。それらの違いは、署名性に関連するものではなく、例外に対するものです。ADD では、オーバーフローに関する例外が発生することがあります。これは、たとえばAda PLで有用[85]であり、サポートされることもあります。ADDU は、オーバーフロー時に例外を発生させません。

C/C++ ではこれをサポートしていないため、この例では ADD の代わりに ADDU が表示されています。

32ビットの結果は $V0 に残ります。

`main()` に新しい命令があります (JAL (「Jump and Link」))。

JAL と JALR の違いは、相対オフセットが最初の命令でエンコードされる一方、JALR はレジスタに格納された絶対アドレスにジャンプすることです（「ジャンプレジスタとリンクレジスタ」)。

`f()` と `main()` の両方の関数は同じオブジェクトファイルに配置されるので、`f()` の相対アドレスは既知で固定されています。

第**1.11**節戻り値を返すことの詳細

x86では、関数の実行結果は通常 EAX レジスタに返されます。[86] バイトタイプまたは文字（*char* ）の場合は、レジスタ EAX（AL ）の最下位部分が使用されます。関数が浮動小数点数を返す場合、代わりにFPUレジスタ ST(0) が使用されます。ARMでは、結果は通常 R0 レジスタに返されます。

第**1.11.1**節***void*** を返す関数の結果を使ってみる

では、`main()` 関数の戻り値が *int* 型ではなく *void* 型であると宣言された場合はどうでしょうか？いわゆるスタートアップコードは、以下のように `main()` を呼び出しています。

```
push envp
push argv
push argc
call main
push eax
call exit
```

言い換えると：

```
exit(main(argc,argv,envp));
```

`main()` を *void* として宣言すると、明示的に（*return* 文を使って）何も返されず、`main()` の最後の EAX レジスタに格納された何かがexit() の唯一の引数になります。おそらく、あなたの関数の実行から放棄されるランダムな値があるでしょう。したがって、プログラムの終了コードは疑似乱数です。

この事実を説明することができます。ここで `main()` 関数は *void* 戻り値の型を持っていることに注意してください。

```
#include <stdio.h>

void main()
{
        printf ("Hello, world!\n");
};
```

Linuxでコンパイルしましょう。

GCC 4.8.1では、`printf()` を `puts()` に置き換えましたが (以下で見てきました：1.5.4 on page 20)、これは `printf()` のように `puts()` が出力する文字数を返すので、これは問題ありません。`main()` が終了する前に EAX がゼロになっていないことに注意してください。

これは、`main()` の最後の EAX の値に `puts()` が残した値が含まれていることを意味します。

Listing 1.96: GCC 4.8.1

```
.LC0:
        .string "Hello, world!"
main:
```

---

[85] http://go.yurichev.com/17326

[86] 参照: MSDN: Return Values (C++): MSDN

```
        push    ebp
        mov     ebp, esp
        and     esp, -16
        sub     esp, 16
        mov     DWORD PTR [esp], OFFSET FLAT:.LC0
        call    puts
        leave
        ret
```

終了ステータスを示すbashスクリプトを書いてみましょう。

Listing 1.97: tst.sh

```
#!/bin/sh
./hello_world
echo $?
```

実行してみましょう。

```
$ tst.sh
Hello, world!
14
```

14は表示された文字数です。印刷される文字の数は、`printf()` から EAX/RAX までの 「exit code」へのスリップです。

ちなみに、Hex-RaysでC++を逆コンパイルすると、あるクラスのデストラクタで終了する関数に遭遇することがよくあります。

```
...

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
```

C++標準では、デストラクタは何も返しませんが、Hex-Raysがそれを知らず、デストラクタとこの関数の両方が *int* を返すと考えると、次のような出力が出力されます。

```
...

        return CString::~CString(&Str);
}
```

第**1.11.2**節関数の戻り値を使わないとどうなる？

`printf()` は正常に出力された文字の数を返しますが、実際にはこの関数の結果はめったに使用されません。

また、値を返す関数を呼び出し、戻り値を使用しないということも可能です。

```
int f()
{
    // 最初の3つのランダム値をスキップする
    rand();
    rand();
    rand();
    // 4番目を使用する
    return rand();
};
```

rand() 関数の結果は EAX の4つのケースすべてに残されています。

しかし、最初の3つのケースでは、EAX の値は使用されていません。

第**1.11.3**節構造体を返す

戻り値が EAX レジスタに残っているという事実に戻りましょう。

そのため、古いCコンパイラでは、1つのレジスタ（通常は *int* ）に収まらないものを返す関数を作成することはできませんが、必要なら、関数の引数として渡されたポインタを介して情報を返す必要があります。

したがって、通常、関数が複数の値を返す必要がある場合は、1つだけを返し、残りのすべてのポインタを返します。

構造全体を返すことが可能になっていますが、それはまだあまり一般的ではありません。関数が大きな構造体を返さなければならない場合、呼び出し側はそれを割り当ててポインタを最初の引数を介してプログラマに透過的に渡す必要があります。これは、最初の引数に手動でポインタを渡すのとほぼ同じですが、コンパイラはそれを隠します。

小さな例

```
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;
};
```

…以下のコードが出力されます (MSVC 2010 /Ox):

```
$T3853 = 8                      ; size = 4
_a$ = 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
```

構造体へのポインタの内部渡しのマクロ名は $T3853 です。

この例は、C99言語拡張を使用して書き直すことができます。

```
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 1.98: 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
```

この関数は、あたかも構造体へのポインタが渡されたかのように、呼び出し元関数によって割り当てられた構造体のフィールドを埋めるだけです。したがって、パフォーマンス上の欠点はありません。

第**1.12**節ポインタ

第**1.12.1**節入力値の入れ替え

こんな仕事をさせてみます

```
#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()
{
        // 文字列をヒープにコピーするので、変更することができます
        char *s=strdup("string");

        // 2番目と3番目の文字をスワップする
        swap_bytes (s+1, s+2);

        printf ("%s\n", s);
};
```

見てわかるように、バイトは MOVZX を使用して ECX と EBX の下位8ビット部分にロードされます（これらのレジスタの上位部分がクリアされます)。その後、バイトがスワップバックされます。

Listing 1.99: Optimizing GCC 5.4

```
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
```

両方のバイトのアドレスは引数から取り出され、関数の実行は EDX と EAX にあります。

だから私たちはポインタを使用しています。恐らく、ポインタなしにこのタスクを解決する良い方法はありません。

第1.12.2節戻り値

ポインターは関数から値を返すためによく使用されます（scanf() 関数の呼び出し（ 1.9 on page 65)）。

たとえば、関数が2つの値を返す必要がある場合などです。

グローバル変数の例

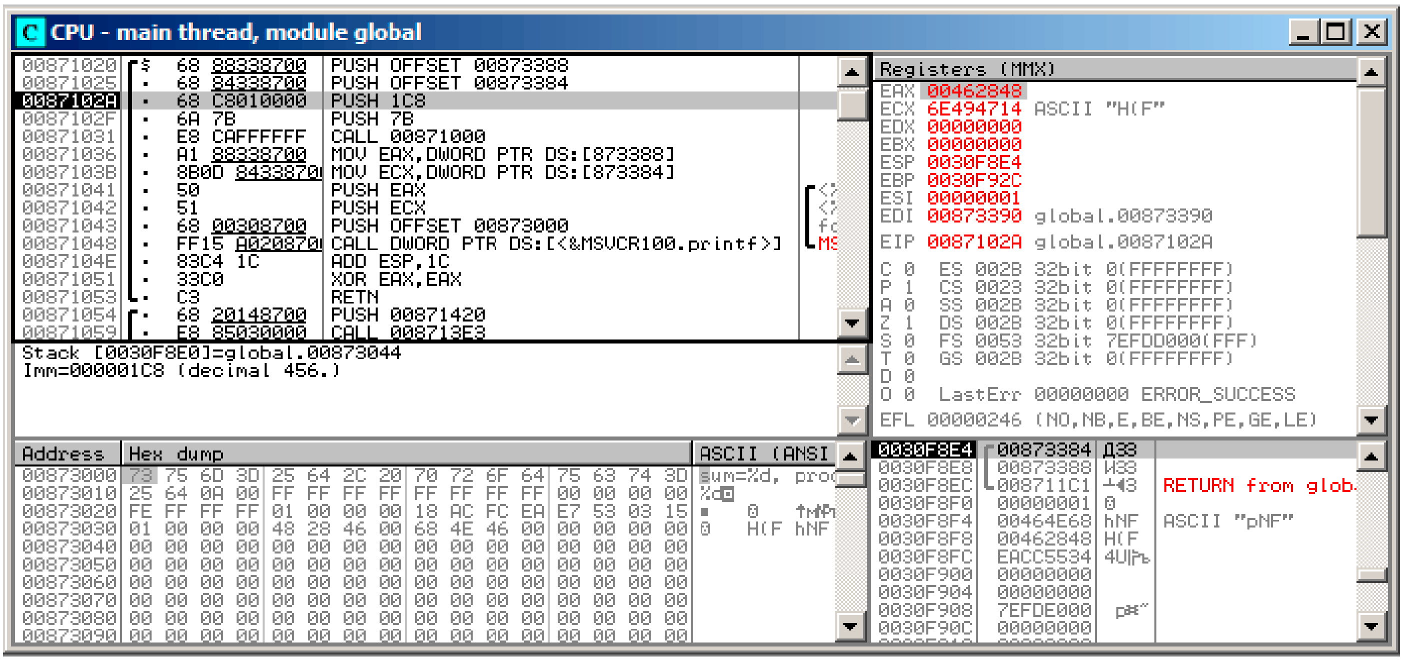
```
#include <stdio.h>

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);
};
```

次のようにコンパイルされます。

Listing 1.100: 最適化 MSVC 2010 (/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
```

OllyDbg で見てみましょう。



```
00871020 ┌$  68 88338700  PUSH OFFSET 00873388
00871025 │.  68 84338700  PUSH OFFSET 00873384
0087102A │.  68 C8010000  PUSH 1C8
0087102F │.  6A 7B        PUSH 7B
00871031 │.  E8 CAFFFFFF  CALL 00871000
00871036 │.  A1 88338700  MOV EAX,DWORD PTR DS:[873388]
0087103B │.  8B0D 8433870 MOV ECX,DWORD PTR DS:[873384]
00871041 │.  50           PUSH EAX
00871042 │.  51           PUSH ECX
00871043 │.  68 00308700  PUSH OFFSET 00873000
00871048 │.  FF15 A020870 CALL DWORD PTR DS:[<&MSVCR100.printf>]
0087104E │.  83C4 1C      ADD ESP,1C
00871051 │.  33C0         XOR EAX,EAX
00871053 └.  C3           RETN
00871054 ┌.  68 20148700  PUSH 00871420
00871059 │.  E8 85030000  CALL 008713E3
Stack [0030F8E0]=global.00873044
Imm=000001C8 (decimal 456.)
```

```
Registers (MMX)
EAX 00462848
ECX 6E494714 ASCII "H(F"
EDX 00000000
EBX 00000000
ESP 0030F8E4
EBP 0030F92C
ESI 00000001
EDI 00873390 global.00873390

EIP 0087102A global.0087102A

C 0  ES 002B 32bit 0(FFFFFFFF)
P 1  CS 0023 32bit 0(FFFFFFFF)
A 0  SS 002B 32bit 0(FFFFFFFF)
Z 1  DS 002B 32bit 0(FFFFFFFF)
S 0  FS 0053 32bit 7EFDD000(FFF)
T 0  GS 002B 32bit 0(FFFFFFFF)
D 0
O 0  LastErr 00000000 ERROR_SUCCESS

EFL 00000246 (NO,NB,E,BE,NS,PE,GE,LE)
```

```
Address  Hex dump                                         ASCII (ANSI
00873000 73 75 6D 3D 25 64 2C 20 70 72 6F 64 75 63 74 3D sum=%d, proc
00873010 25 64 0A 00 FF FF FF FF FF FF FF FF 00 00 00 00 %d
00873020 FE FF FF FF 01 00 00 00 18 AC FC EA E7 53 03 15 ■   0
00873030 01 00 00 00 48 28 46 00 68 4E 46 00 00 00 00 00 0   H(F hNF
00873040 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
00873050 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
00873060 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
00873070 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
00873080 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
00873090 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
```

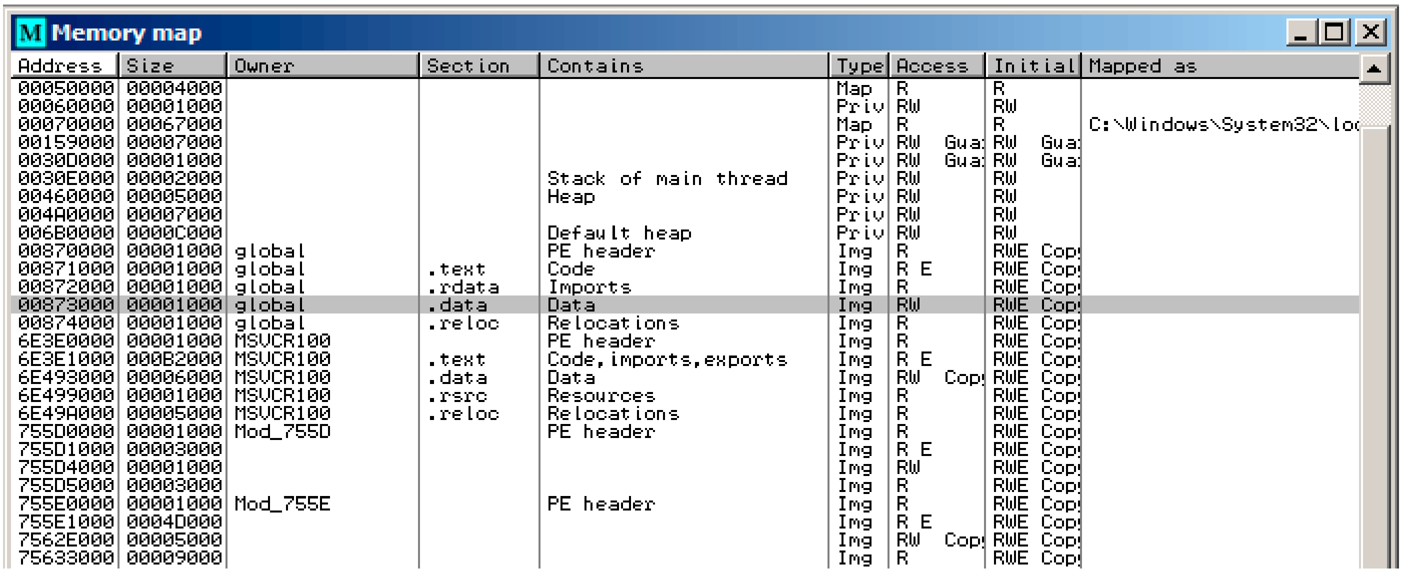
```
0030F8E4 ┌00873384 Д33
0030F8E8 │00873388 И33
0030F8EC └008711C1 ┴◄3  RETURN from glob
0030F8F0  00000001 0
0030F8F4  00464E68 hNF  ASCII "pNF"
0030F8F8  00462848 H(F
0030F8FC  EACC5534 4U╟ъ
0030F900  00000000
0030F904  00000000
0030F908  7EFDE000  pя~
0030F90C  00000000
```

図 1.24: OllyDbg: グローバル変数のアドレスは f1() に渡されます

まず、グローバル変数のアドレスが f1() に渡されます。スタック要素に「Follow in dump」をクリックすると、2つの変数に割り当てられたデータセグメント内の場所を見ることができます。

これらの変数は、初期化されていないデータ（BSSから）が実行開始前にクリアされるため、ゼロにされます。[see *ISO/IEC 9899:TC3 (C C99 standard)*, (2007) 6.7.8p10]

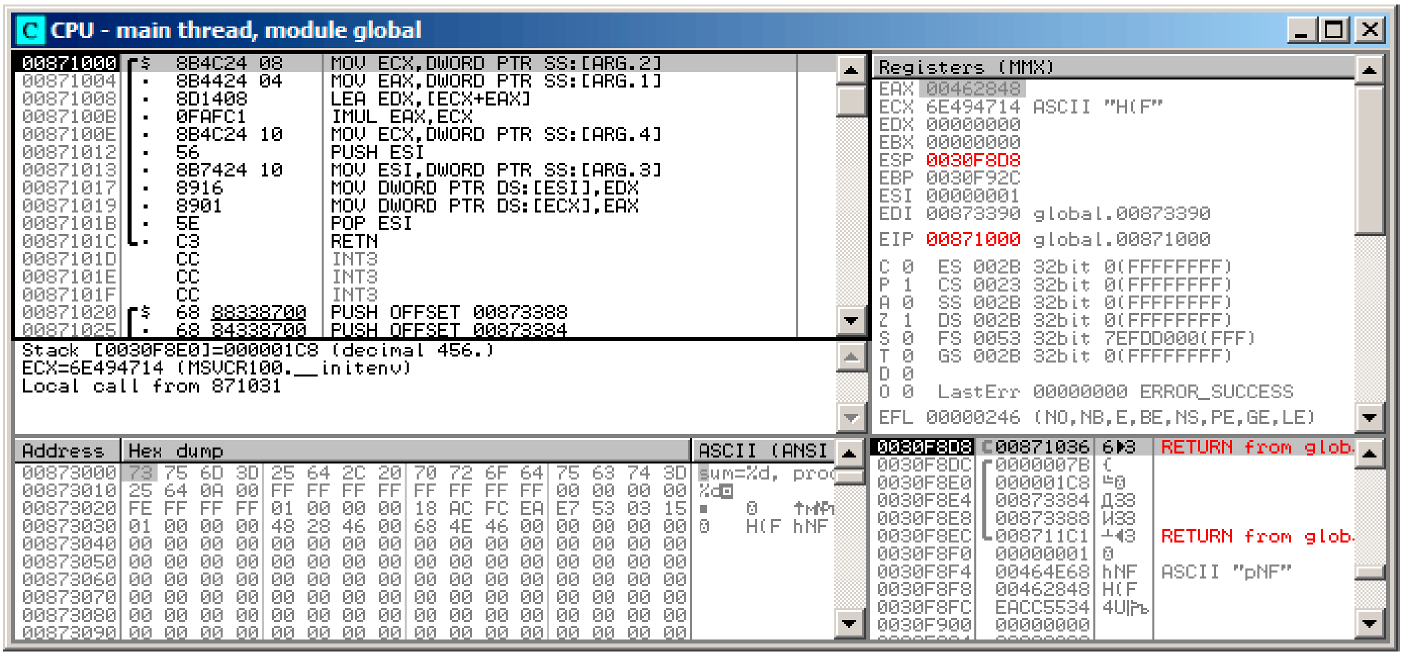
それらはデータセグメントにあり、Alt-Mを押してメモリマップを確認することで確認できます。



```
Address  Size     Owner            Section   Contains                 Type Access   Initial  Mapped as
00050000 00004000                                                     Map  R        R
00060000 00001000                                                     Priv RW       RW
00070000 00067000                                                     Map  R        R        C:\Windows\System32\lo
00159000 00007000                                                     Priv RW   Gua RW   Gua
0030D000 00001000                                                     Priv RW   Gua RW   Gua
0030E000 00002000                            Stack of main thread     Priv RW       RW
00460000 00005000                            Heap                     Priv RW       RW
004A0000 00007000                                                     Priv RW       RW
006B0000 0000C000                            Default heap             Priv RW       RW
00870000 00001000 global                     PE header                Img  R        RWE Cop
00871000 00001000 global           .text     Code                     Img  R E      RWE Cop
00872000 00001000 global           .rdata    Imports                  Img  R        RWE Cop
00873000 00001000 global           .data     Data                     Img  RW       RWE Cop
00874000 00001000 global           .reloc    Relocations              Img  R        RWE Cop
6E3E0000 00001000 MSVCR100                   PE header                Img  R        RWE Cop
6E3E1000 000B2000 MSVCR100         .text     Code,imports,exports     Img  R E      RWE Cop
6E493000 00006000 MSVCR100         .data     Data                     Img  RW   Cop RWE Cop
6E499000 00001000 MSVCR100         .rsrc     Resources                Img  R        RWE Cop
6E49A000 00005000 MSVCR100         .reloc    Relocations              Img  R        RWE Cop
755D0000 00001000 Mod_755D                   PE header                Img  R        RWE Cop
755D1000 00003000                                                     Img  R E      RWE Cop
755D4000 00001000                                                     Img  RW       RWE Cop
755D5000 00003000                                                     Img  R        RWE Cop
755E0000 00001000 Mod_755E                   PE header                Img  R        RWE Cop
755E1000 0004D000                                                     Img  R E      RWE Cop
7562E000 00005000                                                     Img  RW   Cop RWE Cop
75633000 00009000                                                     Img  R        RWE Cop
```

図 1.25: OllyDbg: メモリマップ

f1() の先頭にをトレース（F7）しましょう。



```
00871000 ┏$  8B4C24 08    MOV ECX,DWORD PTR SS:[ARG.2]
00871004 ┃.  8B4424 04    MOV EAX,DWORD PTR SS:[ARG.1]
00871008 ┃.  8D1408       LEA EDX,[ECX+EAX]
0087100B ┃.  0FAFC1       IMUL EAX,ECX
0087100E ┃.  8B4C24 10    MOV ECX,DWORD PTR SS:[ARG.4]
00871012 ┃.  56           PUSH ESI
00871013 ┃.  8B7424 10    MOV ESI,DWORD PTR SS:[ARG.3]
00871017 ┃.  8916         MOV DWORD PTR DS:[ESI],EDX
00871019 ┃.  8901         MOV DWORD PTR DS:[ECX],EAX
0087101B ┃.  5E           POP ESI
0087101C ┗.  C3           RETN
0087101D     CC           INT3
0087101E     CC           INT3
0087101F     CC           INT3
00871020 ┏$  68 88338700  PUSH OFFSET 00873388
00871025 ┃.  68 84338700  PUSH OFFSET 00873384
```

```
Stack [0030F8E0]=000001C8 (decimal 456.)
ECX=6E494714 (MSVCR100.__initenv)
Local call from 871031
```

```
Registers (MMX)
EAX 00462848
ECX 6E494714 ASCII "H(F"
EDX 00000000
EBX 00000000
ESP 0030F8D8
EBP 0030F92C
ESI 00000001
EDI 00873390 global.00873390

EIP 00871000 global.00871000

C 0  ES 002B 32bit 0(FFFFFFFF)
P 1  CS 0023 32bit 0(FFFFFFFF)
A 0  SS 002B 32bit 0(FFFFFFFF)
Z 1  DS 002B 32bit 0(FFFFFFFF)
S 0  FS 0053 32bit 7EFDD000(FFF)
T 0  GS 002B 32bit 0(FFFFFFFF)
D 0
O 0  LastErr 00000000 ERROR_SUCCESS

EFL 00000246 (NO,NB,E,BE,NS,PE,GE,LE)
```

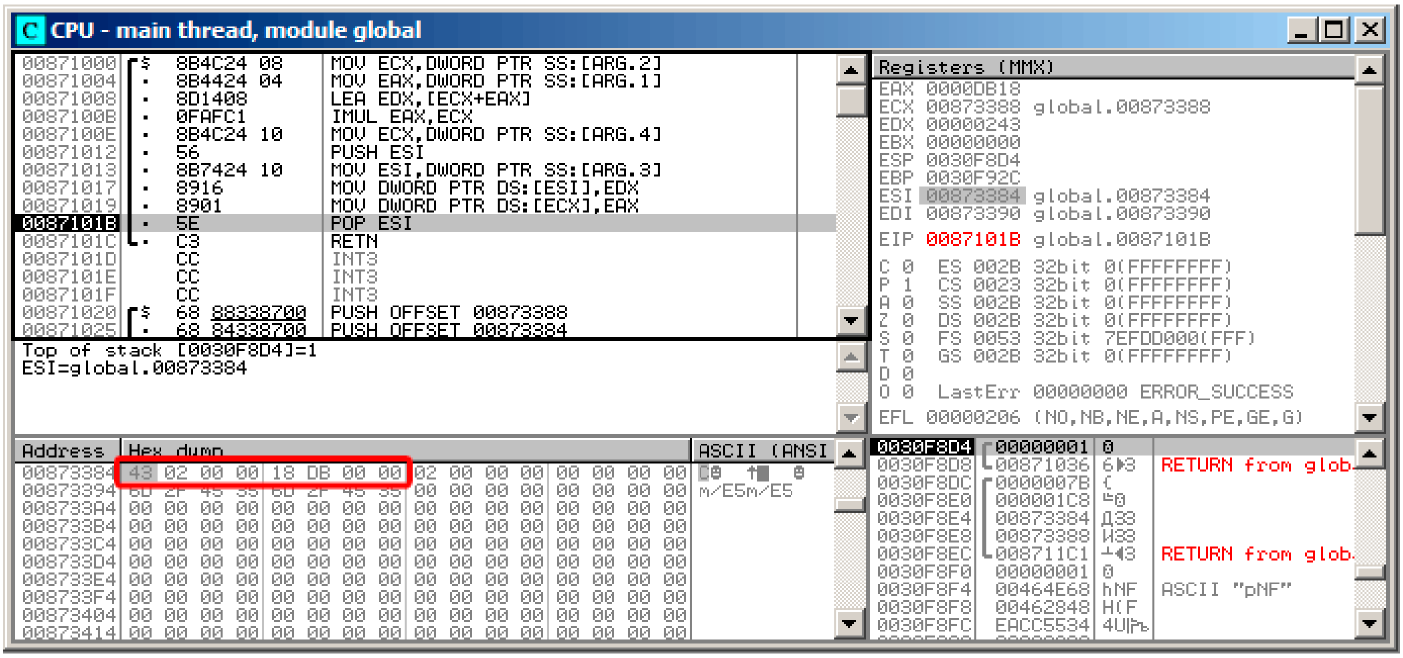
```
Address  Hex dump                                         ASCII (ANSI
00873000 73 75 6D 3D|25 64 2C 20|70 72 6F 64|75 63 74 3D sum=%d, proc
00873010 25 64 0A 00|FF FF FF FF|FF FF FF FF|00 00 00 00 %d◘
00873020 FE FF FF FF|01 00 00 00|18 AC FC EA|E7 53 03 15 ■   0   ↑мЮr
00873030 01 00 00 00|48 28 46 00|68 4E 46 00|00 00 00 00 0   H(F hNF
00873040 00 00 00 00|00 00 00 00|00 00 00 00|00 00 00 00
00873050 00 00 00 00|00 00 00 00|00 00 00 00|00 00 00 00
00873060 00 00 00 00|00 00 00 00|00 00 00 00|00 00 00 00
00873070 00 00 00 00|00 00 00 00|00 00 00 00|00 00 00 00
00873080 00 00 00 00|00 00 00 00|00 00 00 00|00 00 00 00
00873090 00 00 00 00|00 00 00 00|00 00 00 00|00 00 00 00
```

```
0030F8D8  00871036  6▶3   RETURN from glob
0030F8DC ┏0000007B  {
0030F8E0 ┃000001C8  ╚0
0030F8E4 ┃00873384  Д33
0030F8E8 ┃00873388  И33
0030F8EC ┗008711C1  ┴◀3   RETURN from glob
0030F8F0  00000001  0
0030F8F4  00464E68  hNF   ASCII "pNF"
0030F8F8  00462848  H(F
0030F8FC  EACC5534  4U╟ъ
0030F900  00000000
```

図 1.26: OllyDbg: f1() が開始

2つの値がスタック456（0x1C8）と 123（0x7B）に表示され、2つのグローバル変数のアドレスも表示されます。

f1() の終わりまでトレースしましょう。左下のウィンドウで、計算結果がグローバル変数にどのように表示されるかを確認します。



```
00871000  $  8B4C24 08    MOV ECX,DWORD PTR SS:[ARG.2]
00871004  .  8B4424 04    MOV EAX,DWORD PTR SS:[ARG.1]
00871008  .  8D1408       LEA EDX,[ECX+EAX]
0087100B  .  0FAFC1       IMUL EAX,ECX
0087100E  .  8B4C24 10    MOV ECX,DWORD PTR SS:[ARG.4]
00871012  .  56           PUSH ESI
00871013  .  8B7424 10    MOV ESI,DWORD PTR SS:[ARG.3]
00871017  .  8916         MOV DWORD PTR DS:[ESI],EDX
00871019  .  8901         MOV DWORD PTR DS:[ECX],EAX
0087101B  .  5E           POP ESI
0087101C  .  C3           RETN
0087101D     CC           INT3
0087101E     CC           INT3
0087101F     CC           INT3
00871020  $  68 88338700  PUSH OFFSET 00873388
00871025  .  68 84338700  PUSH OFFSET 00873384
```

```
Top of stack [0030F8D4]=1
ESI=global.00873384
```

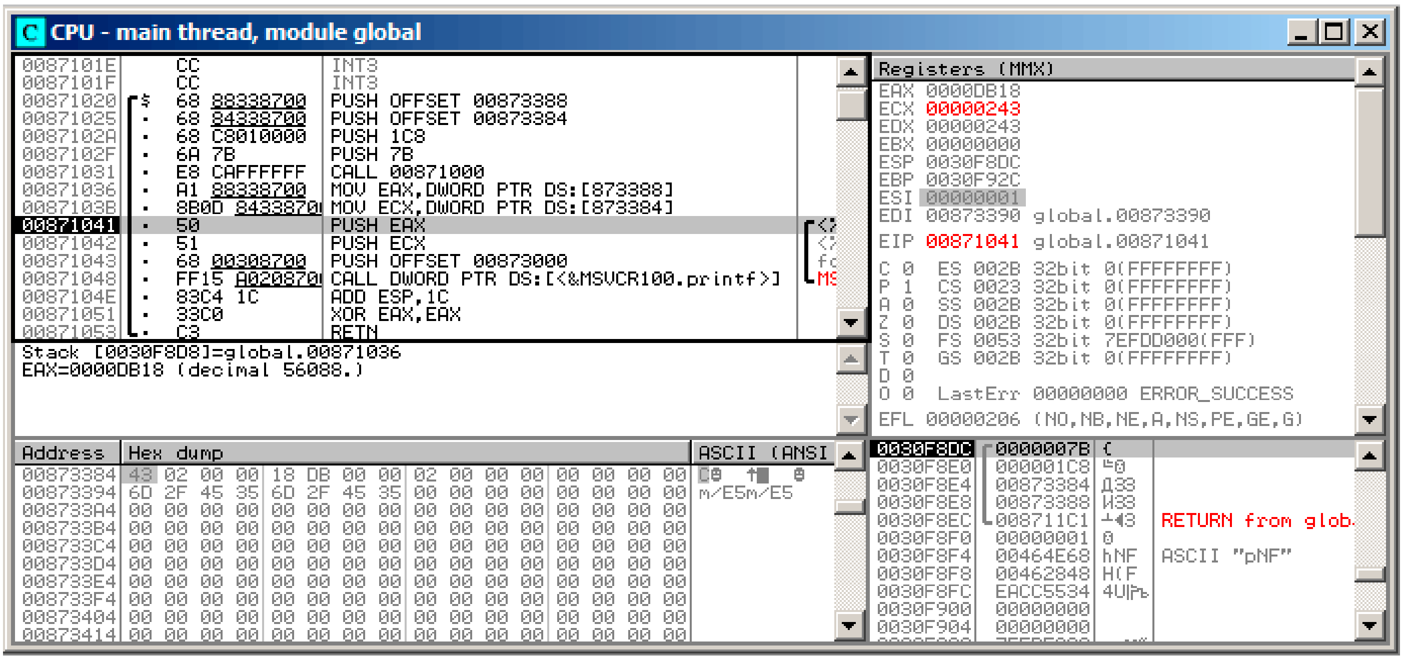
```
Registers (MMX)
EAX 0000DB18
ECX 00873388 global.00873388
EDX 00000243
EBX 00000000
ESP 0030F8D4
EBP 0030F92C
ESI 00873384 global.00873384
EDI 00873390 global.00873390

EIP 0087101B global.0087101B

C 0  ES 002B 32bit 0(FFFFFFFF)
P 1  CS 0023 32bit 0(FFFFFFFF)
A 0  SS 002B 32bit 0(FFFFFFFF)
Z 0  DS 002B 32bit 0(FFFFFFFF)
S 0  FS 0053 32bit 7EFDD000(FFF)
T 0  GS 002B 32bit 0(FFFFFFFF)
D 0
O 0  LastErr 00000000 ERROR_SUCCESS

EFL 00000206 (NO,NB,NE,A,NS,PE,GE,G)
```

```
Address  Hex dump                                         ASCII (ANSI
00873384 43 02 00 00 18 DB 00 00 02 00 00 00 00 00 00 00  C☻  ↑█  ☻
00873394 6D 2F 45 35 6D 2F 45 35 00 00 00 00 00 00 00 00  m/E5m/E5
008733A4 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
008733B4 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
008733C4 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
008733D4 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
008733E4 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
008733F4 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
00873404 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
00873414 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
```

```
0030F8D4  00000001  ☺
0030F8D8  00871036  6▶3   RETURN from glob
0030F8DC  0000007B  {
0030F8E0  000001C8  ╚☺
0030F8E4  00873384  Д33
0030F8E8  00873388  И33
0030F8EC  008711C1  ┴◀3   RETURN from glob
0030F8F0  00000001  ☺
0030F8F4  00464E68  hNF   ASCII "pNF"
0030F8F8  00462848  H(F
0030F8FC  EACC5534  4U╠ъ
```

図 1.27: OllyDbg: f1() 実行完了

グローバル変数の値は、（スタックを介して）printf() に渡す準備が整ったレジスタにロードされます。



```
0087101E     CC          INT3
0087101F     CC          INT3
00871020  ┌$ 68 88338700 PUSH OFFSET 00873388
00871025  │. 68 84338700 PUSH OFFSET 00873384
0087102A  │. 68 C8010000 PUSH 1C8
0087102F  │. 6A 7B       PUSH 7B
00871031  │. E8 CAFFFFFF CALL 00871000
00871036  │. A1 88338700 MOV EAX,DWORD PTR DS:[873388]
0087103B  │. 8B0D 8433870 MOV ECX,DWORD PTR DS:[873384]
00871041  │. 50          PUSH EAX
00871042  │. 51          PUSH ECX
00871043  │. 68 00308700 PUSH OFFSET 00873000
00871048  │. FF15 A020870 CALL DWORD PTR DS:[<&MSVCR100.printf>]
0087104E  │. 83C4 1C     ADD ESP,1C
00871051  │. 33C0        XOR EAX,EAX
00871053  └. C3          RETN
Stack [0030F8D8]=global.00871036
EAX=0000DB18 (decimal 56088.)
```

```
Registers (MMX)
EAX 0000DB18
ECX 00000243
EDX 00000243
EBX 00000000
ESP 0030F8DC
EBP 0030F92C
ESI 00000001
EDI 00873390 global.00873390

EIP 00871041 global.00871041

C 0  ES 002B 32bit 0(FFFFFFFF)
P 1  CS 0023 32bit 0(FFFFFFFF)
A 0  SS 002B 32bit 0(FFFFFFFF)
Z 0  DS 002B 32bit 0(FFFFFFFF)
S 0  FS 0053 32bit 7EFDD000(FFF)
T 0  GS 002B 32bit 0(FFFFFFFF)
D 0
O 0  LastErr 00000000 ERROR_SUCCESS

EFL 00000206 (NO,NB,NE,A,NS,PE,GE,G)
```

```
Address  Hex dump                                         ASCII (ANSI
00873384 43 02 00 00 18 DB 00 00 02 00 00 00 00 00 00 00
00873394 6D 2F 45 35 6D 2F 45 35 00 00 00 00 00 00 00 00 m/E5m/E5
008733A4 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
008733B4 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
008733C4 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
008733D4 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
008733E4 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
008733F4 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
00873404 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
00873414 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
```

```
0030F8DC ┌0000007B {
0030F8E0 │000001C8
0030F8E4 │00873384
0030F8E8 │00873388
0030F8EC └008711C1        RETURN from glob
0030F8F0  00000001
0030F8F4  00464E68 hNF    ASCII "pNF"
0030F8F8  00462848 H(F
0030F8FC  EACC5534
0030F900  00000000
0030F904  00000000
```

図 1.28: OllyDbg: グローバル変数の値は printf() に渡されます

ローカル変数の例

私たちの例を少し修正しましょう。

Listing 1.101: now the sum and product variables are local

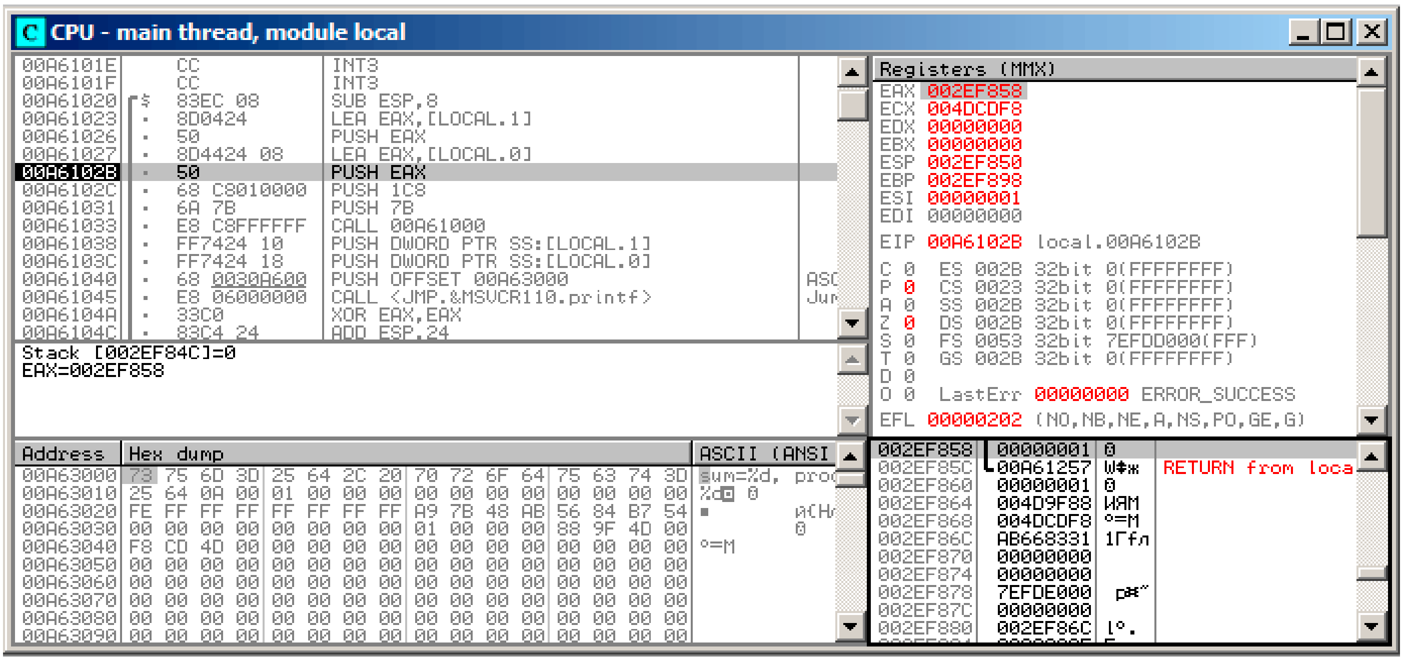
```
void main()
{
        int sum, product; // 変数は関数ローカルにあります

        f1(123, 456, &sum, &product);
        printf ("sum=%d, product=%d\n", sum, product);
};
```

f1() コードは変更されません。main() のコードだけが行います：

Listing 1.102: 最適化 MSVC 2010 (/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
```

OllyDbg をもう一度見てみましょう。スタック内のローカル変数のアドレスは 0x2EF854 and 0x2EF858 です。これらがスタックにどのようにプッシュされるのかを確認します。



```
00A6101E      CC            INT3
00A6101F      CC            INT3
00A61020  ┌$  83EC 08       SUB ESP,8
00A61023  │.  8D0424        LEA EAX,[LOCAL.1]
00A61026  │.  50            PUSH EAX
00A61027  │.  8D4424 08     LEA EAX,[LOCAL.0]
00A6102B  │.  50            PUSH EAX
00A6102C  │.  68 C8010000   PUSH 1C8
00A61031  │.  6A 7B         PUSH 7B
00A61033  │.  E8 C8FFFFFF   CALL 00A61000
00A61038  │.  FF7424 10     PUSH DWORD PTR SS:[LOCAL.1]
00A6103C  │.  FF7424 18     PUSH DWORD PTR SS:[LOCAL.0]
00A61040  │.  68 0030A600   PUSH OFFSET 00A63000                  ASC
00A61045  │.  E8 06000000   CALL <JMP.&MSVCR110.printf>           Jur
00A6104A  │.  33C0          XOR EAX,EAX
00A6104C  │.  83C4 24       ADD ESP,24
```

```
Stack [002EF84C]=0
EAX=002EF858
```

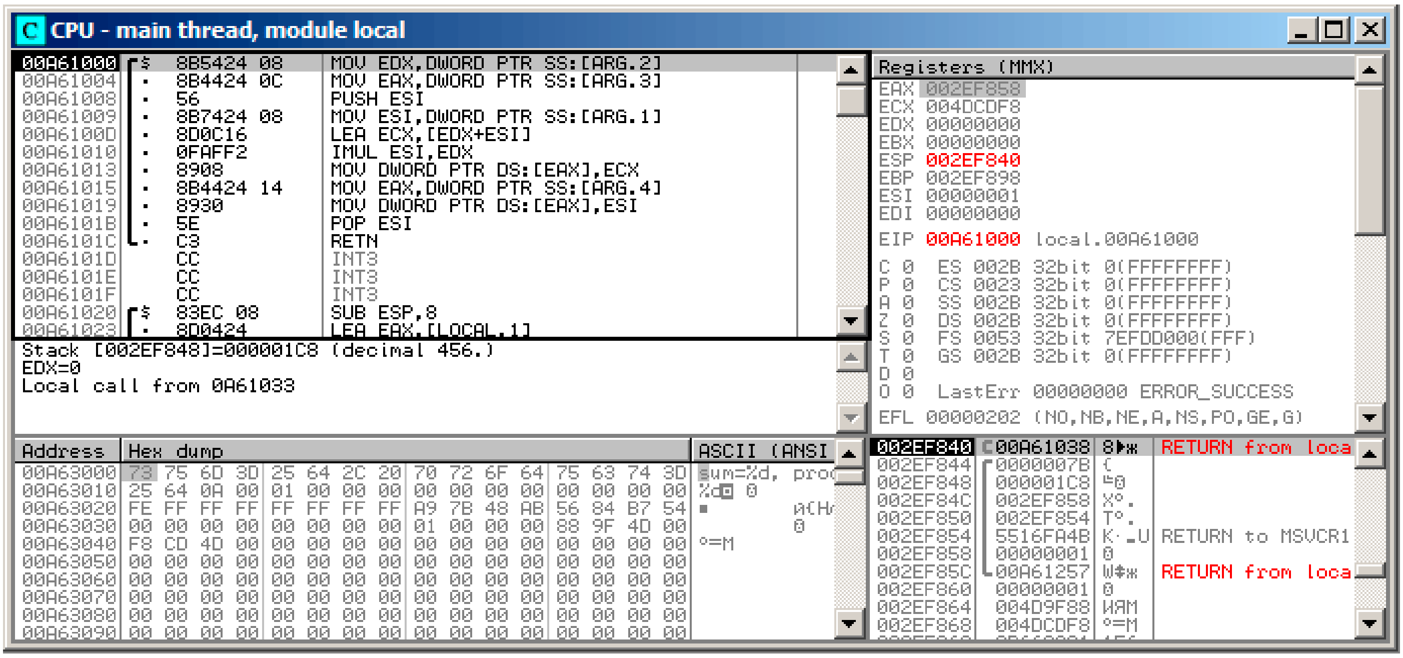
```
Registers (MMX)
EAX 002EF858
ECX 004DCDF8
EDX 00000000
EBX 00000000
ESP 002EF850
EBP 002EF898
ESI 00000001
EDI 00000000

EIP 00A6102B local.00A6102B

C 0  ES 002B 32bit 0(FFFFFFFF)
P 0  CS 0023 32bit 0(FFFFFFFF)
A 0  SS 002B 32bit 0(FFFFFFFF)
Z 0  DS 002B 32bit 0(FFFFFFFF)
S 0  FS 0053 32bit 7EFDD000(FFF)
T 0  GS 002B 32bit 0(FFFFFFFF)
D 0
O 0  LastErr 00000000 ERROR_SUCCESS

EFL 00000202 (NO,NB,NE,A,NS,PO,GE,G)
```

```
Address  Hex dump                                         ASCII (ANSI
00A63000 73 75 6D 3D 25 64 2C 20 70 72 6F 64 75 63 74 3D  sum=%d, proc
00A63010 25 64 0A 00 01 00 00 00 00 00 00 00 00 00 00 00  %d◘ 0
00A63020 FE FF FF FF FF FF FF FF A9 7B 48 AB 56 84 B7 54  ■          ø{H
00A63030 00 00 00 00 00 00 00 00 01 00 00 00 88 9F 4D 00             0
00A63040 F8 CD 4D 00 00 00 00 00 00 00 00 00 00 00 00 00  °=M
00A63050 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
00A63060 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
00A63070 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
00A63080 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
00A63090 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
```

```
002EF858 ┌00000001 0
002EF85C └00A61257 W‡ж    RETURN from loca
002EF860  00000001 0
002EF864  004D9F88 ИЯM
002EF868  004DCDF8 °=M
002EF86C  AB668331 1Γfл
002EF870  00000000
002EF874  00000000
002EF878  7EFDE000  pæ˝
002EF87C  00000000
002EF880  002EF86C l°.
```

図 1.29: OllyDbg: ローカル変数のアドレスがスタックにプッシュされる

f1() が開始します。今のところ 0x2EF854 and 0x2EF858 のスタックにランダムなゴミだけがあります。



```
00A61000 ┌$  8B5424 08    MOV EDX,DWORD PTR SS:[ARG.2]
00A61004 │.  8B4424 0C    MOV EAX,DWORD PTR SS:[ARG.3]
00A61008 │.  56           PUSH ESI
00A61009 │.  8B7424 08    MOV ESI,DWORD PTR SS:[ARG.1]
00A6100D │.  8D0C16       LEA ECX,[EDX+ESI]
00A61010 │.  0FAFF2       IMUL ESI,EDX
00A61013 │.  8908         MOV DWORD PTR DS:[EAX],ECX
00A61015 │.  8B4424 14    MOV EAX,DWORD PTR SS:[ARG.4]
00A61019 │.  8930         MOV DWORD PTR DS:[EAX],ESI
00A6101B │.  5E           POP ESI
00A6101C └.  C3           RETN
00A6101D     CC           INT3
00A6101E     CC           INT3
00A6101F     CC           INT3
00A61020 ┌$  83EC 08      SUB ESP,8
00A61023 │.  8D0424       LEA EAX,[LOCAL.1]
Stack [002EF848]=000001C8 (decimal 456.)
EDX=0
Local call from 0A61033
```

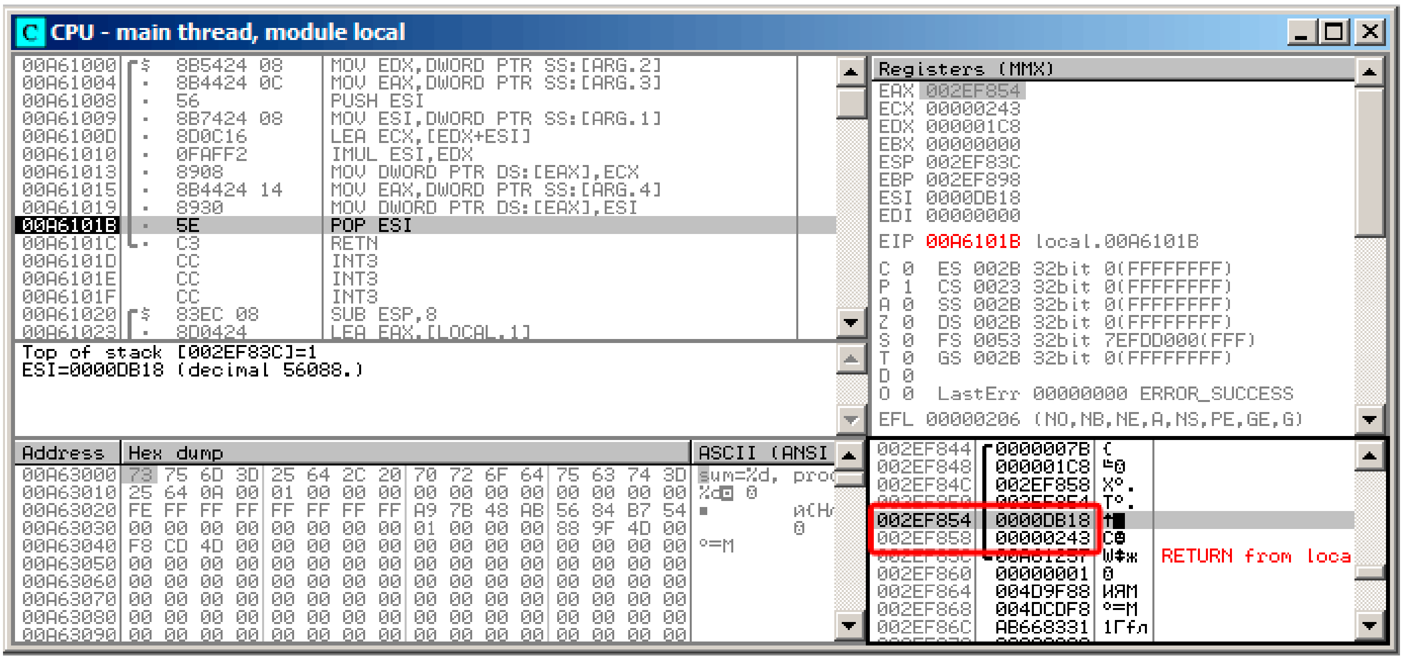
```
Registers (MMX)
EAX 002EF858
ECX 004DCDF8
EDX 00000000
EBX 00000000
ESP 002EF840
EBP 002EF898
ESI 00000001
EDI 00000000

EIP 00A61000 local.00A61000

C 0  ES 002B 32bit 0(FFFFFFFF)
P 0  CS 0023 32bit 0(FFFFFFFF)
A 0  SS 002B 32bit 0(FFFFFFFF)
Z 0  DS 002B 32bit 0(FFFFFFFF)
S 0  FS 0053 32bit 7EFDD000(FFF)
T 0  GS 002B 32bit 0(FFFFFFFF)
D 0
O 0  LastErr 00000000 ERROR_SUCCESS

EFL 00000202 (NO,NB,NE,A,NS,PO,GE,G)
```

```
Address  Hex dump                                         ASCII (ANSI
00A63000 73 75 6D 3D 25 64 2C 20 70 72 6F 64 75 63 74 3D sum=%d, proc
00A63010 25 64 0A 00 01 00 00 00 00 00 00 00 00 00 00 00 %d◘ 0
00A63020 FE FF FF FF FF FF FF FF A9 7B 48 AB 56 84 B7 54 ■       a{H
00A63030 00 00 00 00 00 00 00 00 01 00 00 00 88 9F 4D 00         0
00A63040 F8 CD 4D 00 00 00 00 00 00 00 00 00 00 00 00 00 °=M
00A63050 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
00A63060 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
00A63070 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
00A63080 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
00A63090 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
```

```
002EF840  00A61038 8▶ж  RETURN from loca
002EF844 ┌0000007B {
002EF848 │000001C8 ╚0
002EF84C │002EF858 X°.
002EF850 │002EF854 T°.
002EF854 │5516FA4B K·▄U RETURN to MSVCR1
002EF858 │00000001 0
002EF85C └00A61257 W‡ж  RETURN from loca
002EF860  00000001 0
002EF864  004D9F88 ИЯM
002EF868  004DCDF8 °=M
```

図 1.30: OllyDbg: f1() 開始

f1() が完了。



```
00A61000 ┌$  8B5424 08    MOV EDX,DWORD PTR SS:[ARG.2]
00A61004 │.  8B4424 0C    MOV EAX,DWORD PTR SS:[ARG.3]
00A61008 │.  56           PUSH ESI
00A61009 │.  8B7424 08    MOV ESI,DWORD PTR SS:[ARG.1]
00A6100D │.  8D0C16       LEA ECX,[EDX+ESI]
00A61010 │.  0FAFF2       IMUL ESI,EDX
00A61013 │.  8908         MOV DWORD PTR DS:[EAX],ECX
00A61015 │.  8B4424 14    MOV EAX,DWORD PTR SS:[ARG.4]
00A61019 │.  8930         MOV DWORD PTR DS:[EAX],ESI
00A6101B │.  5E           POP ESI
00A6101C └.  C3           RETN
00A6101D     CC           INT3
00A6101E     CC           INT3
00A6101F     CC           INT3
00A61020 ┌$  83EC 08      SUB ESP,8
00A61023 │.  8D0424       LEA EAX,[LOCAL.1]
```

```
Top of stack [002EF83C]=1
ESI=0000DB18 (decimal 56088.)
```

```
Registers (MMX)
EAX 002EF854
ECX 00000243
EDX 000001C8
EBX 00000000
ESP 002EF83C
EBP 002EF898
ESI 0000DB18
EDI 00000000

EIP 00A6101B local.00A6101B

C 0  ES 002B 32bit 0(FFFFFFFF)
P 1  CS 0023 32bit 0(FFFFFFFF)
A 0  SS 002B 32bit 0(FFFFFFFF)
Z 0  DS 002B 32bit 0(FFFFFFFF)
S 0  FS 0053 32bit 7EFDD000(FFF)
T 0  GS 002B 32bit 0(FFFFFFFF)
D 0
O 0  LastErr 00000000 ERROR_SUCCESS

EFL 00000206 (NO,NB,NE,A,NS,PE,GE,G)
```

```
Address  Hex dump                                         ASCII (ANSI
00A63000 73 75 6D 3D 25 64 2C 20 70 72 6F 64 75 63 74 3D sum=%d, proc
00A63010 25 64 0A 00 01 00 00 00 00 00 00 00 00 00 00 00 %d◘ ☺
00A63020 FE FF FF FF FF FF FF FF A9 7B 48 AB 56 84 B7 54 ■        ⌐{H
00A63030 00 00 00 00 00 00 00 00 01 00 00 00 88 9F 4D 00         ☺
00A63040 F8 CD 4D 00 00 00 00 00 00 00 00 00 00 00 00 00 °=M
00A63050 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
00A63060 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
00A63070 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
00A63080 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
00A63090 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
```

```
002EF844 ┌0000007B {
002EF848 │000001C8 ╚☺
002EF84C │002EF858 X°.
         │         T°.
002EF854 │0000DB18 ↑█
002EF858 │00000243 C☻
         │         W‡ж  RETURN from loca
002EF860  00000001 ☺
002EF864  004D9F88 ИЯM
002EF868  004DCDF8 °=M
002EF86C  AB668331 1Γfл
```

図 1.31: OllyDbg: f1() 実行完了

ここで、アドレス 0x2EF854 and 0x2EF858 に 0xDB18 と 0x243 が見つかります。これらの値は f1() の結果です。

結論

f1() は、メモリ内の任意の場所にポインタを返すことができます。

これは本質的にポインタの有用性です。

ところで、C++ リファレンスはまったく同じように動作します。それらの詳細については、（ **??** on page ??）を参照してください。

第1.13節GOTO演算子

GOTO演算子は、一般的にアンチパターンとみなされます。[Edgar Dijkstra, *Go To Statement Considered Harmful* (1968)[87]] を参照してください。それにもかかわらず、それは合理的に使用することができます [Donald E. Knuth, *Structured Programming with go to Statements* (1974)[88]] [89] を参照してください。

ここには非常に単純な例があります。

```
#include <stdio.h>

int main()
{
        printf ("begin\n");
        goto exit;
        printf ("skip me!\n");
exit:
        printf ("end\n");
};
```

MSVC 2012ではは次のようになります。

---

[87] http://yurichev.com/mirrors/Dijkstra68.pdf

[88] http://yurichev.com/mirrors/KnuthStructuredProgrammingGoTo.pdf

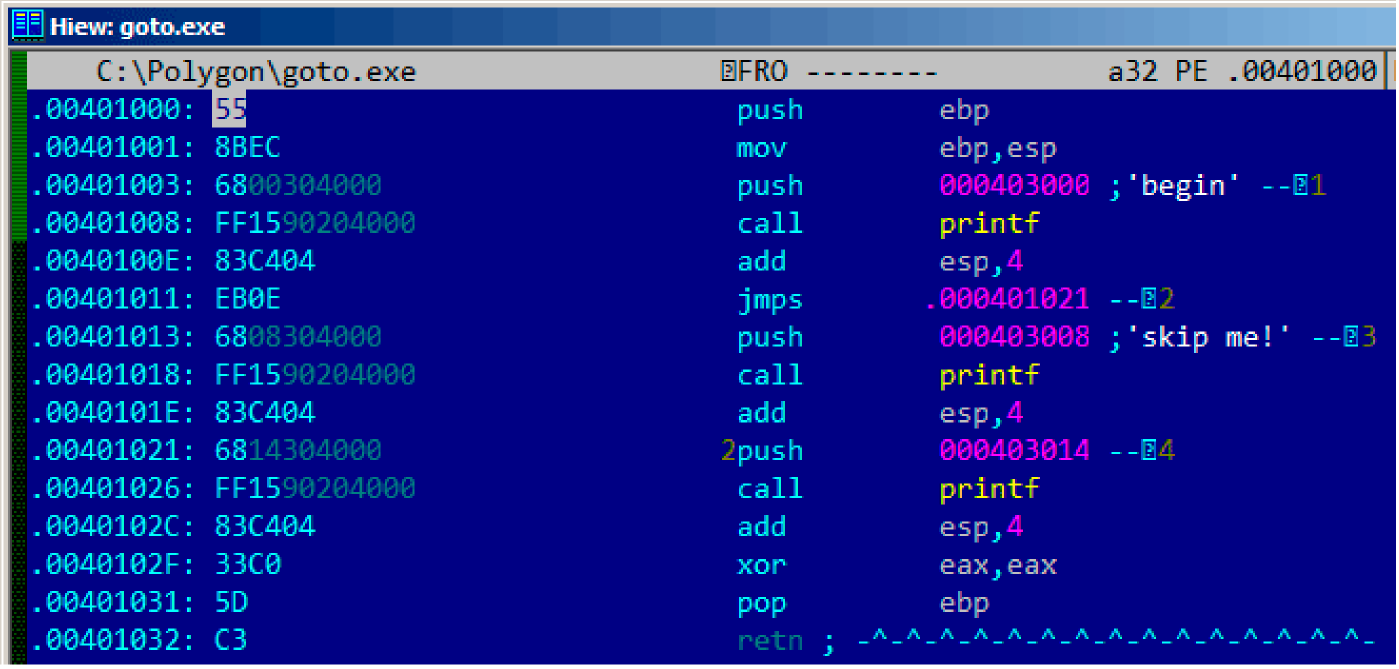
[89] [Dennis Yurichev, *C/C++ programming language notes*] にもいくつか例があります

Listing 1.103: 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
```

goto 文は単に JMP 命令に置き換えられています。これは同じ効果があります。別の場所への無条件ジャンプです。2番目の `printf()` は、人間の介入、デバッガの使用、またはコードのパッチ適用によってのみ実行できます。

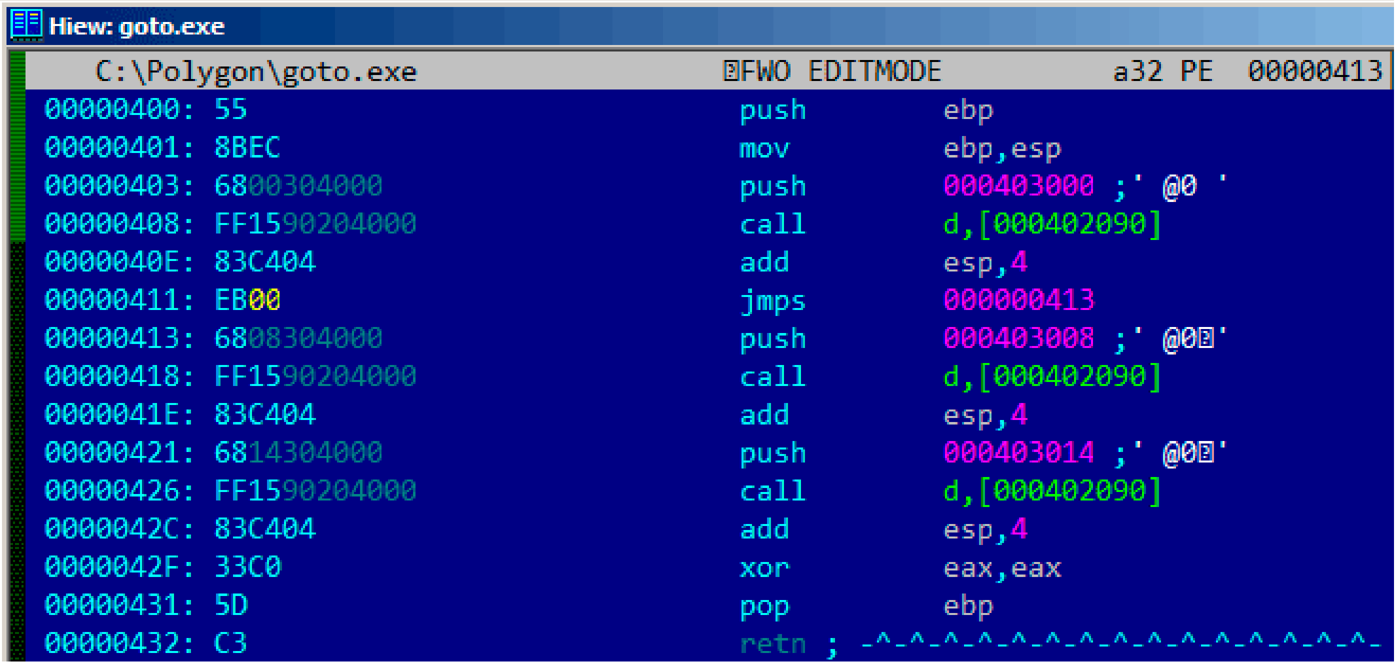
これは簡単なパッチの練習としても役立ちます。Hiewで結果の実行ファイルを開きましょう：



```
   C:\Polygon\goto.exe                     □FRO --------          a32 PE .00401000
.00401000: 55                               push        ebp
.00401001: 8BEC                             mov         ebp,esp
.00401003: 6800304000                       push        000403000 ;'begin' --□1
.00401008: FF1590204000                     call        printf
.0040100E: 83C404                           add         esp,4
.00401011: EB0E                             jmps       .000401021 --□2
.00401013: 6808304000                       push        000403008 ;'skip me!' --□3
.00401018: FF1590204000                     call        printf
.0040101E: 83C404                           add         esp,4
.00401021: 6814304000                      2push        000403014 --□4
.00401026: FF1590204000                     call        printf
.0040102C: 83C404                           add         esp,4
.0040102F: 33C0                             xor         eax,eax
.00401031: 5D                               pop         ebp
.00401032: C3                               retn ; -^-^-^-^-^-^-^-^-^-^-^-^-^-^-^-^-
```

図 1.32: Hiew

カーソルを JMP（0x410）に設定し、F3（編集）を押し、0を2回押すと、オペコードが EB 00 になります。



```
Hiew: goto.exe
   C:\Polygon\goto.exe                   □FWO EDITMODE          a32 PE  00000413
00000400: 55                              push        ebp
00000401: 8BEC                            mov         ebp,esp
00000403: 6800304000                      push        000403000 ;' @0 '
00000408: FF1590204000                    call        d,[000402090]
0000040E: 83C404                          add         esp,4
00000411: EB00                            jmps        000000413
00000413: 6808304000                      push        000403008 ;' @0□'
00000418: FF1590204000                    call        d,[000402090]
0000041E: 83C404                          add         esp,4
00000421: 6814304000                      push        000403014 ;' @0□'
00000426: FF1590204000                    call        d,[000402090]
0000042C: 83C404                          add         esp,4
0000042F: 33C0                            xor         eax,eax
00000431: 5D                              pop         ebp
00000432: C3                              retn ; -^-^-^-^-^-^-^-^-^-^-^-^-^-^-^-
```

図 1.33: Hiew

JMP オペコードの第2バイトはジャンプの相対オフセットを示し、0は現在の命令の直後のポイントを示します。

したがって、JMP は2番目の printf() 呼び出しをスキップしません。

F9（保存）を押して終了します。実行ファイルを実行すると、次のように表示されます。

Listing 1.104: Patched executable output

```
C:\...>goto.exe

begin
skip me!
end
```

JMP 命令を2つの NOP 命令に置き換えることによっても同じ結果が得られます。

NOP のオペコードは 0x90 で、長さは1バイトなので、JMP の代わりに2バイトの命令（サイズは2バイト）が必要です。

第**1.13.1**節デッドコード

2番目の printf() 呼び出しは、コンパイラの用語で「デッドコード」とも呼ばれます。

つまり、コードは決して実行されません。したがって、この例を最適化してコンパイルすると、コンパイラは痕跡を残さずに、「デッドコード」を削除します。

Listing 1.105: 最適化 MSVC 2012

```
$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
```

しかし、コンパイラは 「skip me!」文字列を削除するのを忘れていました。

第**1.13.2**節練習問題

あなたの好きなコンパイラとデバッガを使って同じ結果を達成してみてください。

第**1.14**節条件付きジャンプ

第**1.14.1**節 シンプルな例

```
#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

以下は、`f_signed()` 関数がどうなっているかを示しています。

Listing 1.106: 非最適化 MSVC 2010

```
_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
```

最初の命令 `JLE` は、*Jump if Less or Equal* の場合はJumpを表します。言い換えれば、第2オペランドが第1オペランドより大きいか等しい場合、制御フローは命令で指定されたアドレスまたはラベルに移ります。第2オペランドが最初のオペランドより小さいためにこの条件がトリガされない場合、制御フローは変更されず、最初の `printf()` が実行されます。2番目のチェックは、`JNE` ：*Jump if Not Equal* です。オペランドが等しい場合、制御フローは変更されません。

3番目のチェックは、最初のオペランドが2番目のオペランドより大きい場合、または等しい場合は `JGE` ：*Jump if Greater or Equal* です。したがって、3つの条件ジャンプがすべてトリガされた場合、`printf()` の呼び出しはまったく実行されません。これは特別な介入なしには不可能です。`f_unsigned()` 関数を見てみましょう。`f_unsigned()` 関数は、次のように、`JLE` および `JGE` の代わりにJBEおよびJAE命令が使用される点を除いて、`f_signed()` と同じです。

Listing 1.107: 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
```

すでに説明したように、分岐命令は異なります。`JBE`—*Jump if Below or Equal* and `JAE`—*Jump if Above or Equal* これらの命令（`JA/JAE/JB/JBE` ）は、`JG/JGE/JL/JLE` とは、符号なしの数字で動作する点が異なります。

また、符号付き数値表現についてのセクションも参照してください（ **??** on page ??)。`JA/JB` の代わりに `JG/JL` が使用されている場合や、その逆の場合は、変数がそれぞれ符号付きか、または符号なしなのかがほぼはっきりします。ここには、もう何も新しくない、`main()` 関数もあります。

Listing 1.108: `main()`

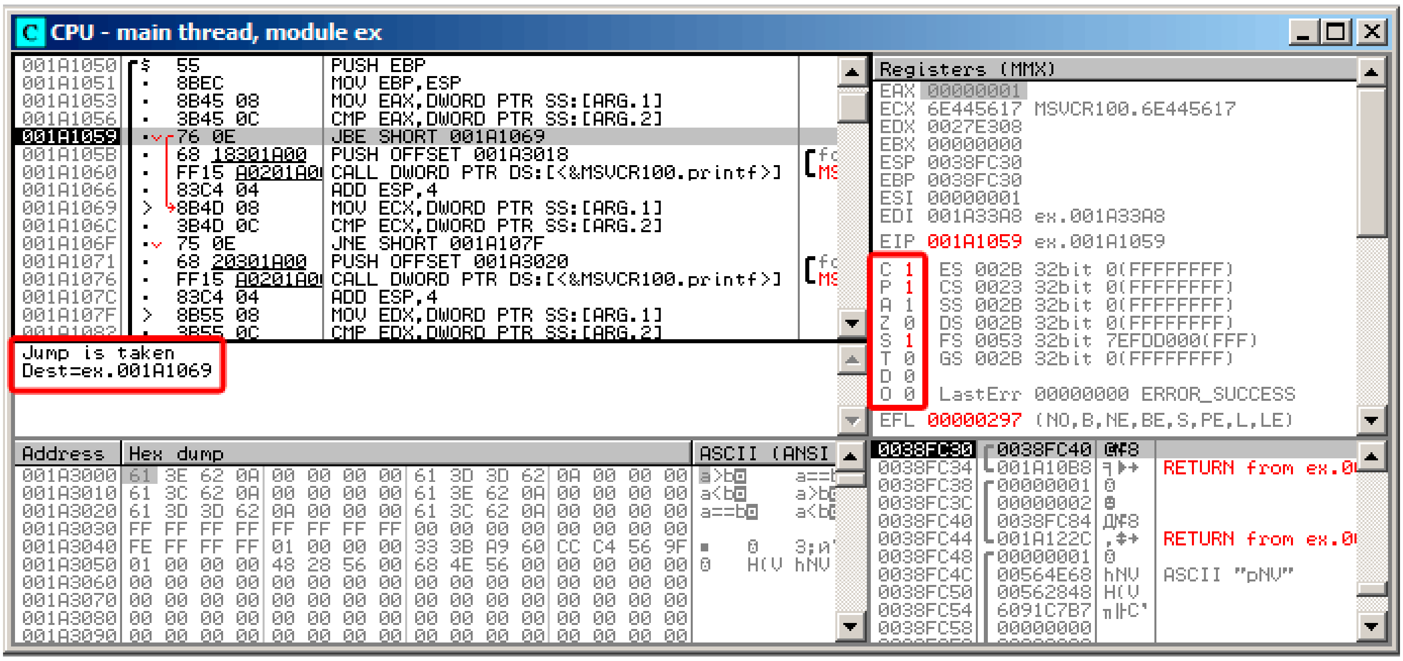
```
_main   PROC
        push    ebp
        mov     ebp, esp
        push    2
        push    1
        call    _f_signed
        add     esp, 8
        push    2
        push    1
        call    _f_unsigned
        add     esp, 8
        xor     eax, eax
        pop     ebp
        ret     0
_main   ENDP
```

x86 + MSVC + OllyDbg

OllyDbg でこの例を実行すると、フラグがどのように設定されているかを見ることができます。符号なしの数値で動作する `f_unsigned()` から始めましょう。

CMP はここで3回実行されますが、同じ引数についてはフラグは毎回同じです。

最初の比較の結果は、



```
001A1050  $   55            PUSH EBP
001A1051  .   8BEC          MOV EBP,ESP
001A1053  .   8B45 08       MOV EAX,DWORD PTR SS:[ARG.1]
001A1056  .   3B45 0C       CMP EAX,DWORD PTR SS:[ARG.2]
001A1059  .v  76 0E         JBE SHORT 001A1069
001A105B  .   68 18301A00   PUSH OFFSET 001A3018
001A1060  .   FF15 A0201A0  CALL DWORD PTR DS:[<&MSVCR100.printf>]
001A1066  .   83C4 04       ADD ESP,4
001A1069  >   8B4D 08       MOV ECX,DWORD PTR SS:[ARG.1]
001A106C  .   3B4D 0C       CMP ECX,DWORD PTR SS:[ARG.2]
001A106F  .v  75 0E         JNE SHORT 001A107F
001A1071  .   68 20301A00   PUSH OFFSET 001A3020
001A1076  .   FF15 A0201A0  CALL DWORD PTR DS:[<&MSVCR100.printf>]
001A107C  .   83C4 04       ADD ESP,4
001A107F  >   8B55 08       MOV EDX,DWORD PTR SS:[ARG.1]
```

```
Jump is taken
Dest=ex.001A1069
```

```
Registers (MMX)
EAX 00000001
ECX 6E445617 MSVCR100.6E445617
EDX 0027E308
EBX 00000000
ESP 0038FC30
EBP 0038FC30
ESI 00000001
EDI 001A33A8 ex.001A33A8

EIP 001A1059 ex.001A1059

C 1  ES 002B 32bit 0(FFFFFFFF)
P 1  CS 0023 32bit 0(FFFFFFFF)
A 1  SS 002B 32bit 0(FFFFFFFF)
Z 0  DS 002B 32bit 0(FFFFFFFF)
S 1  FS 0053 32bit 7EFDD000(FFF)
T 0  GS 002B 32bit 0(FFFFFFFF)
D 0
O 0  LastErr 00000000 ERROR_SUCCESS

EFL 00000297 (NO,B,NE,BE,S,PE,L,LE)
```

```
Address   Hex dump                                         ASCII (ANSI
001A3000  61 3E 62 0A 00 00 00 00 61 3D 3D 62 0A 00 00 00  a>b     a==b
001A3010  61 3C 62 0A 00 00 00 00 61 3E 62 0A 00 00 00 00  a<b     a>b
001A3020  61 3D 3D 62 0A 00 00 00 61 3C 62 0A 00 00 00 00  a==b    a<b
001A3030  FF FF FF FF FF FF FF FF 00 00 00 00 00 00 00 00
001A3040  FE FF FF FF 01 00 00 00 33 3B A9 60 CC C4 56 9F
001A3050  01 00 00 00 48 28 56 00 68 4E 56 00 00 00 00 00      H(V hNV
001A3060  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
001A3070  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
001A3080  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
001A3090  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
```

```
0038FC30  0038FC40
0038FC34  001A10B8        RETURN from ex.0
0038FC38  00000001
0038FC3C  00000002
0038FC40  0038FC84
0038FC44  001A122C        RETURN from ex.0
0038FC48  00000001
0038FC4C  00564E68  hNV   ASCII "pNV"
0038FC50  00562848  H(V
0038FC54  6091C7B7
0038FC58  00000000
```

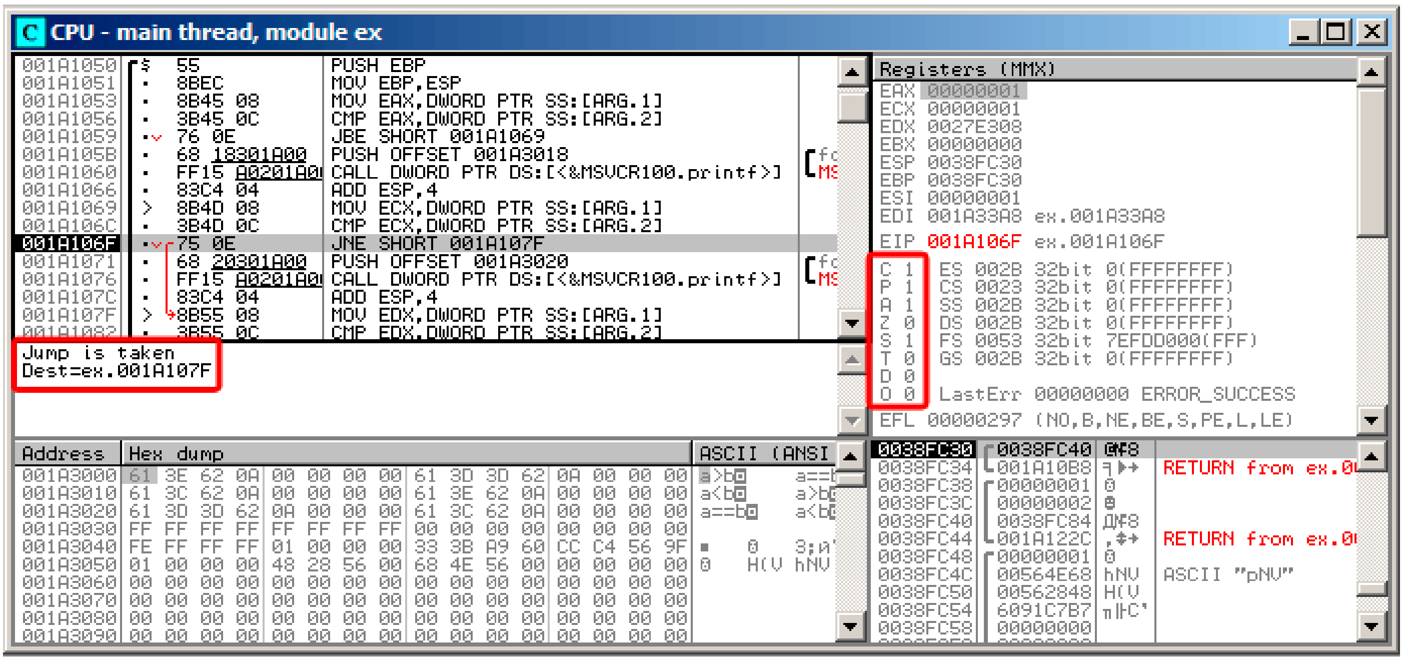
図 1.34: OllyDbg: `f_unsigned()`: 最初の条件付きジャンプ

従って、フラグは、C=1、P=1、A=1、Z=0、S=1、T=0、D=0、O=0です。

これらは OllyDbg では1文字の略号で命名されています。

OllyDbg は、（JBE ）ジャンプがトリガーされることを示唆しています。実際に、インテルのマニュアル（ 7.1.4 on page 164）を調べると、CF=1またはZF=1の場合、JBEが起動することがわかります。条件はここに当てはまるので、ジャンプが開始されます。

次の条件付きジャンプは、



```
001A1050  $   55            PUSH EBP
001A1051  .   8BEC          MOV EBP,ESP
001A1053  .   8B45 08       MOV EAX,DWORD PTR SS:[ARG.1]
001A1056  .   3B45 0C       CMP EAX,DWORD PTR SS:[ARG.2]
001A1059  .v  76 0E         JBE SHORT 001A1069
001A105B  .   68 18301A00   PUSH OFFSET 001A3018
001A1060  .   FF15 A0201A0  CALL DWORD PTR DS:[<&MSVCR100.printf>]
001A1066  .   83C4 04       ADD ESP,4
001A1069  >   8B4D 08       MOV ECX,DWORD PTR SS:[ARG.1]
001A106C  .   3B4D 0C       CMP ECX,DWORD PTR SS:[ARG.2]
001A106F  .v  75 0E         JNE SHORT 001A107F
001A1071  .   68 20301A00   PUSH OFFSET 001A3020
001A1076  .   FF15 A0201A0  CALL DWORD PTR DS:[<&MSVCR100.printf>]
001A107C  .   83C4 04       ADD ESP,4
001A107F  >   8B55 08       MOV EDX,DWORD PTR SS:[ARG.1]
001A1082  .   3B55 0C       CMP EDX,DWORD PTR SS:[ARG.2]
```

```
Jump is taken
Dest=ex.001A107F
```

```
Registers (MMX)
EAX 00000001
ECX 00000001
EDX 0027E308
EBX 00000000
ESP 0038FC30
EBP 0038FC30
ESI 00000001
EDI 001A33A8 ex.001A33A8

EIP 001A106F ex.001A106F

C 1  ES 002B 32bit 0(FFFFFFFF)
P 1  CS 0023 32bit 0(FFFFFFFF)
A 1  SS 002B 32bit 0(FFFFFFFF)
Z 0  DS 002B 32bit 0(FFFFFFFF)
S 1  FS 0053 32bit 7EFDD000(FFF)
T 0  GS 002B 32bit 0(FFFFFFFF)
D 0
O 0  LastErr 00000000 ERROR_SUCCESS

EFL 00000297 (NO,B,NE,BE,S,PE,L,LE)
```

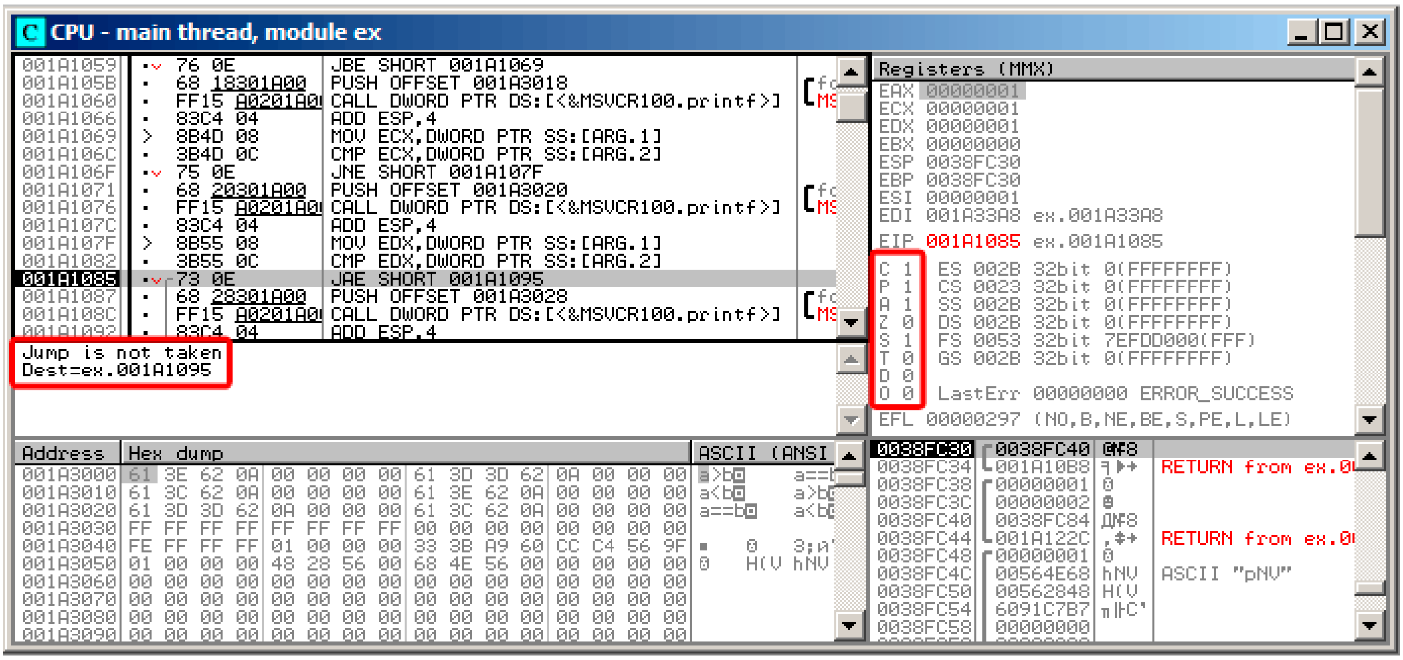
```
Address   Hex dump
001A3000  61 3E 62 0A 00 00 00 00 61 3D 3D 62 0A 00 00 00
001A3010  61 3C 62 0A 00 00 00 00 61 3E 62 0A 00 00 00 00
001A3020  61 3D 3D 62 0A 00 00 00 61 3C 62 0A 00 00 00 00
001A3030  FF FF FF FF FF FF FF FF 00 00 00 00 00 00 00 00
001A3040  FE FF FF FF 01 00 00 00 33 3B A9 60 CC C4 56 9F
001A3050  01 00 00 00 48 28 56 00 68 4E 56 00 00 00 00 00
001A3060  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
001A3070  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
001A3080  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
001A3090  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
```

```
0038FC30  0038FC40
0038FC34  001A10B8  RETURN from ex.0
0038FC38  00000001
0038FC3C  00000002
0038FC40  0038FC84
0038FC44  001A122C  RETURN from ex.0
0038FC48  00000001
0038FC4C  00564E68  ASCII "pNV"
0038FC50  00562848
0038FC54  6091C7B7
0038FC58  00000000
```

図 1.35: OllyDbg: f_unsigned(): 2番目の条件付きジャンプ

OllyDbg は、JNZ がトリガーされることを示唆しています。実際、ZF=0（ゼロフラグ）の場合、JNZが起動します。

3番目の条件付きジャンプは、JNB です。



```
001A1059  .v 76 0E          JBE SHORT 001A1069
001A105B  .  68 18301A00    PUSH OFFSET 001A3018
001A1060  .  FF15 A0201A0   CALL DWORD PTR DS:[<&MSVCR100.printf>]
001A1066  .  83C4 04        ADD ESP,4
001A1069  >  8B4D 08        MOV ECX,DWORD PTR SS:[ARG.1]
001A106C  .  3B4D 0C        CMP ECX,DWORD PTR SS:[ARG.2]
001A106F  .v 75 0E          JNE SHORT 001A107F
001A1071  .  68 20301A00    PUSH OFFSET 001A3020
001A1076  .  FF15 A0201A0   CALL DWORD PTR DS:[<&MSVCR100.printf>]
001A107C  .  83C4 04        ADD ESP,4
001A107F  >  8B55 08        MOV EDX,DWORD PTR SS:[ARG.1]
001A1082  .  3B55 0C        CMP EDX,DWORD PTR SS:[ARG.2]
001A1085  .v 73 0E          JAE SHORT 001A1095
001A1087  .  68 28301A00    PUSH OFFSET 001A3028
001A108C  .  FF15 A0201A0   CALL DWORD PTR DS:[<&MSVCR100.printf>]
001A1092  .  83C4 04        ADD ESP,4
```

```
Jump is not taken
Dest=ex.001A1095
```

```
Registers (MMX)
EAX 00000001
ECX 00000001
EDX 00000001
EBX 00000000
ESP 0038FC30
EBP 0038FC30
ESI 00000001
EDI 001A33A8 ex.001A33A8

EIP 001A1085 ex.001A1085

C 1  ES 002B 32bit 0(FFFFFFFF)
P 1  CS 0023 32bit 0(FFFFFFFF)
A 1  SS 002B 32bit 0(FFFFFFFF)
Z 0  DS 002B 32bit 0(FFFFFFFF)
S 1  FS 0053 32bit 7EFDD000(FFF)
T 0  GS 002B 32bit 0(FFFFFFFF)
D 0
O 0  LastErr 00000000 ERROR_SUCCESS
EFL 00000297 (NO,B,NE,BE,S,PE,L,LE)
```

```
Address  Hex dump                                         ASCII (ANSI
001A3000 61 3E 62 0A 00 00 00 00 61 3D 3D 62 0A 00 00 00 a>b     a==b
001A3010 61 3C 62 0A 00 00 00 00 61 3E 62 0A 00 00 00 00 a<b     a>b
001A3020 61 3D 3D 62 0A 00 00 00 61 3C 62 0A 00 00 00 00 a==b    a<b
001A3030 FF FF FF FF FF FF FF FF 00 00 00 00 00 00 00 00
001A3040 FE FF FF FF 01 00 00 00 33 3B A9 60 CC C4 56 9F
001A3050 01 00 00 00 48 28 56 00 68 4E 56 00 00 00 00 00      H(V hNV
001A3060 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
001A3070 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
001A3080 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
001A3090 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
```

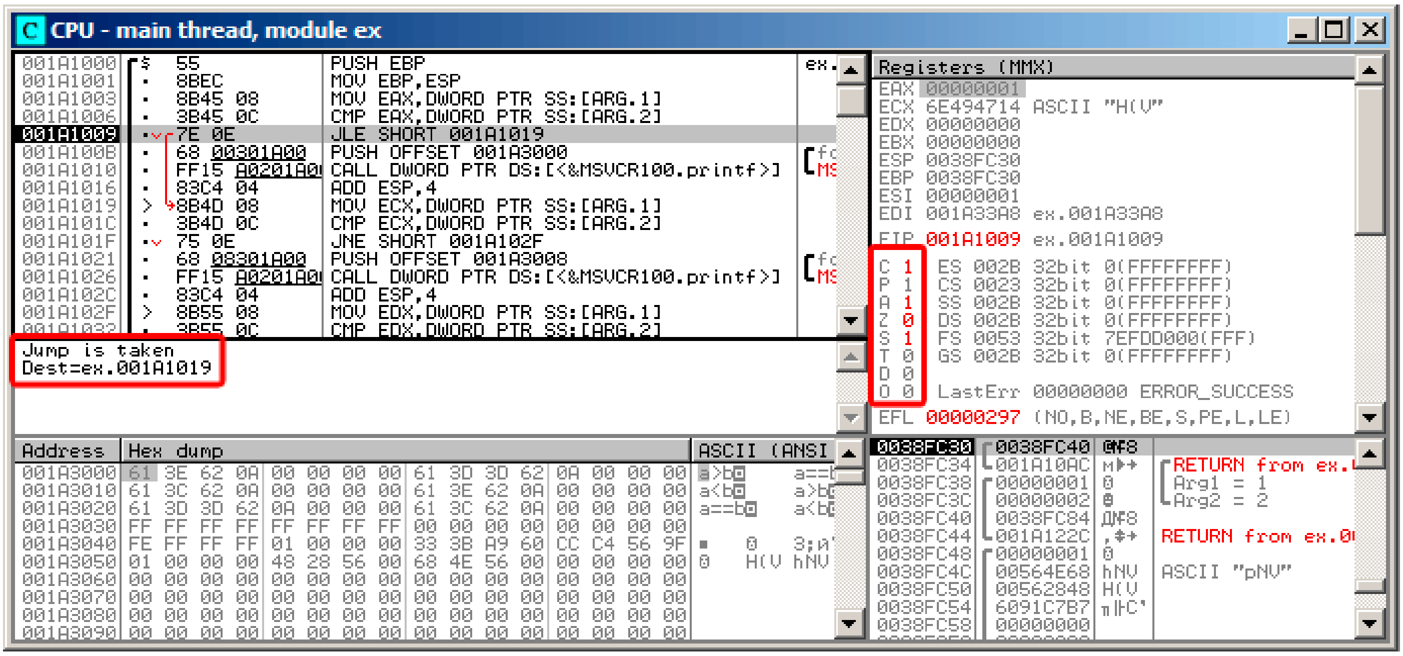
```
0038FC30  0038FC40
0038FC34  001A10B8        RETURN from ex.0
0038FC38  00000001
0038FC3C  00000002
0038FC40  0038FC84
0038FC44  001A122C        RETURN from ex.0
0038FC48  00000001
0038FC4C  00564E68  hNV   ASCII "pNV"
0038FC50  00562848  H(V
0038FC54  6091C7B7
0038FC58  00000000
```

図 1.36: OllyDbg: f_unsigned(): 3番目の条件付きジャンプ

インテルのマニュアル（7.1.4 on page 164）では、CF=0（キャリーフラグ）の場合に JNB が起動することがわかります。今回は当てはまらないので、3番目の printf() が実行されます。

次に、OllyDbg で、符号付きの値で動作する `f_signed()` 関数を見てみましょう。フラグは、C=1、P=1、A=1、Z=0、S=1、T=0、D=0、O=0と同様に設定されます。最初の条件付きジャンプ JLE が起動されます。

インテルマニュアル（166ページの7.1.4）では、ZF = 1またはSFxOFの場合にこの命令がトリガされることがわかりました。SFxOF私たちの場合は、ジャンプがトリガするように。



CPU - main thread, module ex

```
001A1000  $   55          PUSH EBP
001A1001  .   8BEC        MOV EBP,ESP
001A1003  .   8B45 08     MOV EAX,DWORD PTR SS:[ARG.1]
001A1006  .   3B45 0C     CMP EAX,DWORD PTR SS:[ARG.2]
001A1009  .v  7E 0E       JLE SHORT 001A1019
001A100B  .   68 00301A00 PUSH OFFSET 001A3000
001A1010  .   FF15 A0201A0 CALL DWORD PTR DS:[<&MSVCR100.printf>]
001A1016  .   83C4 04     ADD ESP,4
001A1019  >   8B4D 08     MOV ECX,DWORD PTR SS:[ARG.1]
001A101C  .   3B4D 0C     CMP ECX,DWORD PTR SS:[ARG.2]
001A101F  .v  75 0E       JNE SHORT 001A102F
001A1021  .   68 08301A00 PUSH OFFSET 001A3008
001A1026  .   FF15 A0201A0 CALL DWORD PTR DS:[<&MSVCR100.printf>]
001A102C  .   83C4 04     ADD ESP,4
001A102F  >   8B55 08     MOV EDX,DWORD PTR SS:[ARG.1]
001A1032  .   3B55 0C     CMP EDX,DWORD PTR SS:[ARG.2]
```

```
Jump is taken
Dest=ex.001A1019
```

```
Registers (MMX)
EAX 00000001
ECX 6E494714 ASCII "H(V"
EDX 00000000
EBX 00000000
ESP 0038FC30
EBP 0038FC30
ESI 00000001
EDI 001A33A8 ex.001A33A8

EIP 001A1009 ex.001A1009

C 1  ES 002B 32bit 0(FFFFFFFF)
P 1  CS 0023 32bit 0(FFFFFFFF)
A 1  SS 002B 32bit 0(FFFFFFFF)
Z 0  DS 002B 32bit 0(FFFFFFFF)
S 1  FS 0053 32bit 7EFDD000(FFF)
T 0  GS 002B 32bit 0(FFFFFFFF)
D 0
O 0  LastErr 00000000 ERROR_SUCCESS

EFL 00000297 (NO,B,NE,BE,S,PE,L,LE)
```

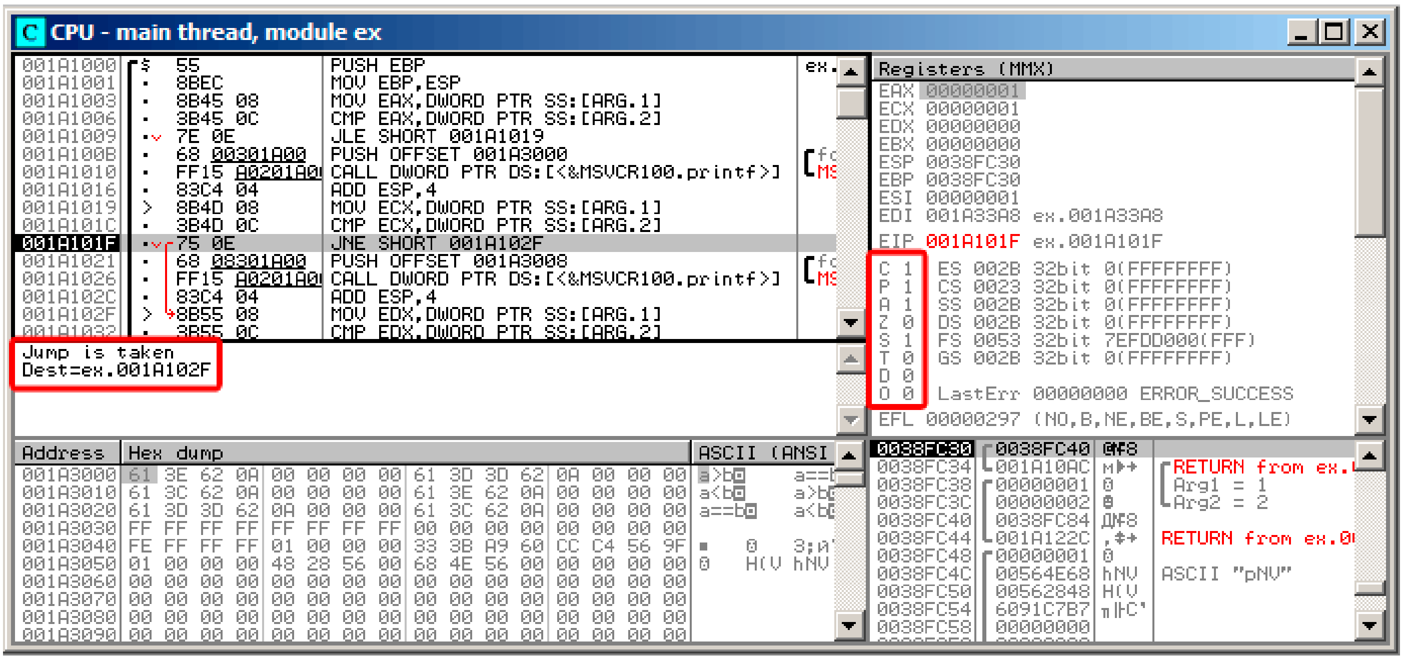
```
Address  Hex dump                                         ASCII (ANSI
001A3000 61 3E 62 0A 00 00 00 00 61 3D 3D 62 0A 00 00 00  a>b       a==b
001A3010 61 3C 62 0A 00 00 00 00 61 3E 62 0A 00 00 00 00  a<b       a>b
001A3020 61 3D 3D 62 0A 00 00 00 61 3C 62 0A 00 00 00 00  a==b      a<b
001A3030 FF FF FF FF FF FF FF FF 00 00 00 00 00 00 00 00
001A3040 FE FF FF FF 01 00 00 00 33 3B A9 60 CC C4 56 9F  ■   0   3;
001A3050 01 00 00 00 48 28 56 00 68 4E 56 00 00 00 00 00  0   H(V hNV
001A3060 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
001A3070 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
001A3080 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
001A3090 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
```

```
0038FC30  0038FC40  @Ї8
0038FC34  001A10AC         RETURN from ex.
0038FC38  00000001  0      Arg1 = 1
0038FC3C  00000002         Arg2 = 2
0038FC40  0038FC84  ДЇ8
0038FC44  001A122C  ,      RETURN from ex.0
0038FC48  00000001  0
0038FC4C  00564E68  hNV    ASCII "pNV"
0038FC50  00562848  H(V
0038FC54  6091C7B7
0038FC58  00000000
```

図 1.37: OllyDbg: f_signed(): 最初の条件付きジャンプ

インテルマニュアル（ 7.1.4 on page 164）では、ZF=1または SF≠OF の場合にこの命令が起動されることがわかりました。私たちの場合では SF≠OF が、ジャンプが起動されます。

2番目の JNZ 条件付きジャンプはZF=0の場合（ゼロ・フラグ）に起動します。



```
001A1000 $  55            PUSH EBP
001A1001 .  8BEC          MOV EBP,ESP
001A1003 .  8B45 08       MOV EAX,DWORD PTR SS:[ARG.1]
001A1006 .  3B45 0C       CMP EAX,DWORD PTR SS:[ARG.2]
001A1009 .v 7E 0E         JLE SHORT 001A1019
001A100B .  68 00301A00   PUSH OFFSET 001A3000
001A1010 .  FF15 A0201A0  CALL DWORD PTR DS:[<&MSVCR100.printf>]
001A1016 .  83C4 04       ADD ESP,4
001A1019 >  8B4D 08       MOV ECX,DWORD PTR SS:[ARG.1]
001A101C .  3B4D 0C       CMP ECX,DWORD PTR SS:[ARG.2]
001A101F .v 75 0E         JNE SHORT 001A102F
001A1021 .  68 08301A00   PUSH OFFSET 001A3008
001A1026 .  FF15 A0201A0  CALL DWORD PTR DS:[<&MSVCR100.printf>]
001A102C .  83C4 04       ADD ESP,4
001A102F >  8B55 08       MOV EDX,DWORD PTR SS:[ARG.1]
001A1032 .  3B55 0C       CMP EDX,DWORD PTR SS:[ARG.2]
```

```
Jump is taken
Dest=ex.001A102F
```

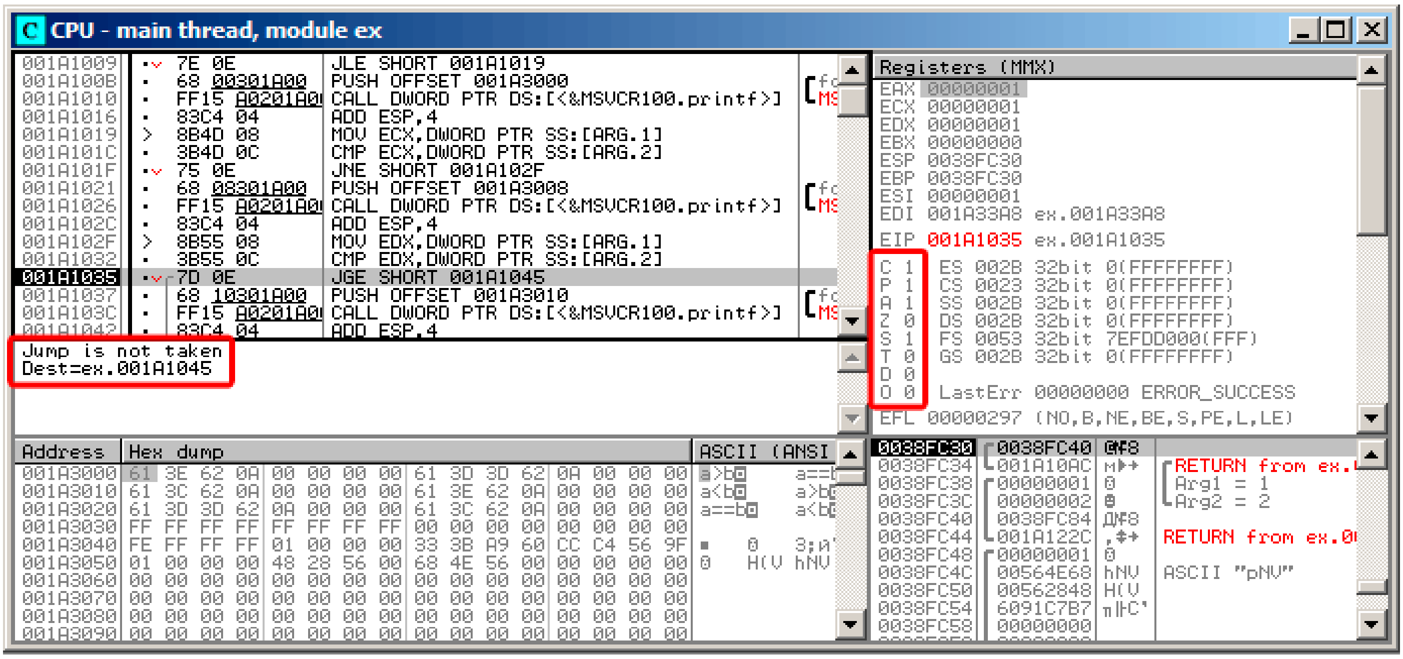
```
Registers (MMX)
EAX 00000001
ECX 00000001
EDX 00000000
EBX 00000000
ESP 0038FC30
EBP 0038FC30
ESI 00000001
EDI 001A33A8 ex.001A33A8

EIP 001A101F ex.001A101F

C 1  ES 002B 32bit 0(FFFFFFFF)
P 1  CS 0023 32bit 0(FFFFFFFF)
A 1  SS 002B 32bit 0(FFFFFFFF)
Z 0  DS 002B 32bit 0(FFFFFFFF)
S 1  FS 0053 32bit 7EFDD000(FFF)
T 0  GS 002B 32bit 0(FFFFFFFF)
D 0
O 0  LastErr 00000000 ERROR_SUCCESS
EFL 00000297 (NO,B,NE,BE,S,PE,L,LE)
```

```
Address  Hex dump                                         ASCII (ANSI
001A3000 61 3E 62 0A 00 00 00 00 61 3D 3D 62 0A 00 00 00 a>b      a==b
001A3010 61 3C 62 0A 00 00 00 00 61 3E 62 0A 00 00 00 00 a<b      a>b
001A3020 61 3D 3D 62 0A 00 00 00 61 3C 62 0A 00 00 00 00 a==b     a<b
001A3030 FF FF FF FF FF FF FF FF 00 00 00 00 00 00 00 00
001A3040 FE FF FF FF 01 00 00 00 33 3B A9 60 CC C4 56 9F ■   0    3;
001A3050 01 00 00 00 48 28 56 00 68 4E 56 00 00 00 00 00 0   H(V  hNV
001A3060 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
001A3070 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
001A3080 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
001A3090 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
```

```
0038FC30 0038FC40 @¥8
0038FC34 001A10AC м▶+  RETURN from ex.
0038FC38 00000001 0    Arg1 = 1
0038FC3C 00000002      Arg2 = 2
0038FC40 0038FC84 ДЊ8
0038FC44 001A122C ,‡+  RETURN from ex.0
0038FC48 00000001 0
0038FC4C 00564E68 hNV  ASCII "pNV"
0038FC50 00562848 H(V
0038FC54 6091C7B7 ╥╟C'
0038FC58 00000000
```

図 1.38: OllyDbg: f_signed(): 2番目の条件付きジャンプ

第3の条件付きジャンプ JGE は、SF=OFの場合にのみ実行されるため、起動しません。今回は、当てはまりません。



```
001A1009  .v  7E 0E          JLE SHORT 001A1019
001A100B  .   68 00301A00    PUSH OFFSET 001A3000
001A1010  .   FF15 A0201A0   CALL DWORD PTR DS:[<&MSVCR100.printf>]
001A1016  .   83C4 04        ADD ESP,4
001A1019  >   8B4D 08        MOV ECX,DWORD PTR SS:[ARG.1]
001A101C  .   3B4D 0C        CMP ECX,DWORD PTR SS:[ARG.2]
001A101F  .v  75 0E          JNE SHORT 001A102F
001A1021  .   68 08301A00    PUSH OFFSET 001A3008
001A1026  .   FF15 A0201A0   CALL DWORD PTR DS:[<&MSVCR100.printf>]
001A102C  .   83C4 04        ADD ESP,4
001A102F  >   8B55 08        MOV EDX,DWORD PTR SS:[ARG.1]
001A1032  .   3B55 0C        CMP EDX,DWORD PTR SS:[ARG.2]
001A1035  .v  7D 0E          JGE SHORT 001A1045
001A1037  .   68 10301A00    PUSH OFFSET 001A3010
001A103C  .   FF15 A0201A0   CALL DWORD PTR DS:[<&MSVCR100.printf>]
001A1042  .   83C4 04        ADD ESP,4
```

```
Jump is not taken
Dest=ex.001A1045
```

```
EAX 00000001
ECX 00000001
EDX 00000001
EBX 00000000
ESP 0038FC30
EBP 0038FC30
ESI 00000001
EDI 001A33A8 ex.001A33A8

EIP 001A1035 ex.001A1035

C 1  ES 002B 32bit 0(FFFFFFFF)
P 1  CS 0023 32bit 0(FFFFFFFF)
A 1  SS 002B 32bit 0(FFFFFFFF)
Z 0  DS 002B 32bit 0(FFFFFFFF)
S 1  FS 0053 32bit 7EFDD000(FFF)
T 0  GS 002B 32bit 0(FFFFFFFF)
D 0
O 0  LastErr 00000000 ERROR_SUCCESS

EFL 00000297 (NO,B,NE,BE,S,PE,L,LE)
```

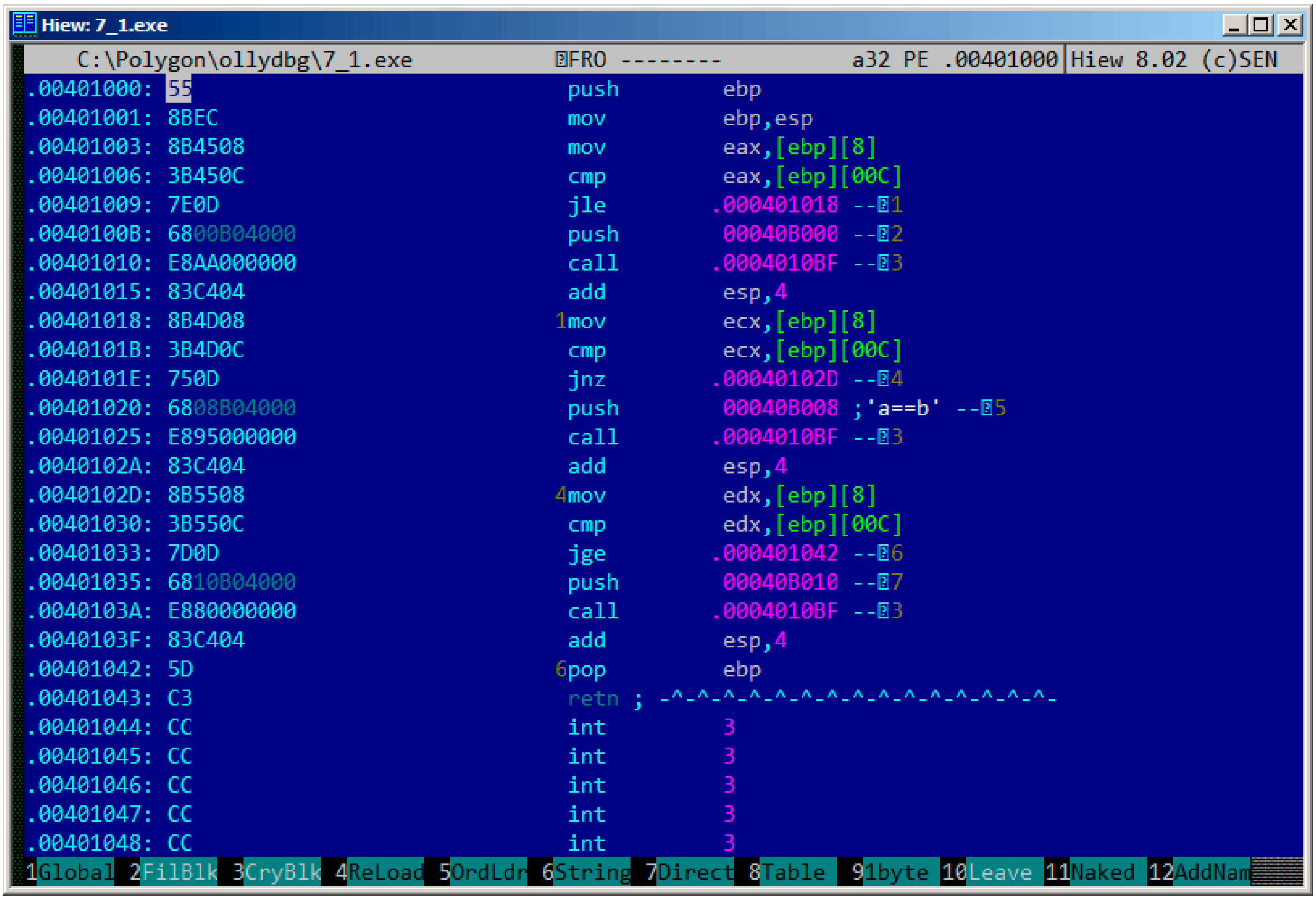
```
001A3000 61 3E 62 0A 00 00 00 00 61 3D 3D 62 0A 00 00 00 a>b     a==b
001A3010 61 3C 62 0A 00 00 00 00 61 3E 62 0A 00 00 00 00 a<b     a>b
001A3020 61 3D 3D 62 0A 00 00 00 61 3C 62 0A 00 00 00 00 a==b    a<b
001A3030 FF FF FF FF FF FF FF FF 00 00 00 00 00 00 00 00
001A3040 FE FF FF FF 01 00 00 00 33 3B A9 60 CC C4 56 9F ■   0   3;й`
001A3050 01 00 00 00 48 28 56 00 68 4E 56 00 00 00 00 00 0   H(V hNV
001A3060 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
001A3070 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
001A3080 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
001A3090 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
```

```
0038FC30  0038FC40  @ₘ8
0038FC34  001A10AC  м▶+  RETURN from ex.
0038FC38  00000001  0    Arg1 = 1
0038FC3C  00000002  ☻    Arg2 = 2
0038FC40  0038FC84  Дₘ8
0038FC44  001A122C  ,‡+  RETURN from ex.0
0038FC48  00000001  0
0038FC4C  00564E68  hNV  ASCII "pNV"
0038FC50  00562848  H(V
0038FC54  6091C7B7  π╟C'
0038FC58  00000000
```

図 1.39: OllyDbg: f_signed(): 3番目の条件付きジャンプ

x86 + MSVC + Hiew

入力値にかかわらず、`f_unsigned()` 関数が常に 「a==b」を出力するように、実行可能ファイルにパッチを当てることができます。ここで、Hiewでどのように見えるか見てみましょう。



```
    C:\Polygon\ollydbg\7_1.exe               □FRO --------          a32 PE .00401000 Hiew 8.02 (c)SEN
.00401000: 55                                push        ebp
.00401001: 8BEC                              mov         ebp,esp
.00401003: 8B4508                            mov         eax,[ebp][8]
.00401006: 3B450C                            cmp         eax,[ebp][00C]
.00401009: 7E0D                              jle        .000401018 --□1
.0040100B: 6800B04000                        push        00040B000 --□2
.00401010: E8AA000000                        call       .0004010BF --□3
.00401015: 83C404                            add         esp,4
.00401018: 8B4D08                           1mov         ecx,[ebp][8]
.0040101B: 3B4D0C                            cmp         ecx,[ebp][00C]
.0040101E: 750D                              jnz        .00040102D --□4
.00401020: 6808B04000                        push        00040B008 ;'a==b' --□5
.00401025: E895000000                        call       .0004010BF --□3
.0040102A: 83C404                            add         esp,4
.0040102D: 8B5508                           4mov         edx,[ebp][8]
.00401030: 3B550C                            cmp         edx,[ebp][00C]
.00401033: 7D0D                              jge        .000401042 --□6
.00401035: 6810B04000                        push        00040B010 --□7
.0040103A: E880000000                        call       .0004010BF --□3
.0040103F: 83C404                            add         esp,4
.00401042: 5D                               6pop         ebp
.00401043: C3                                retn ; -^-^-^-^-^-^-^-^-^-^-^-^-^-^-^-
.00401044: CC                                int         3
.00401045: CC                                int         3
.00401046: CC                                int         3
.00401047: CC                                int         3
.00401048: CC                                int         3
1Global 2FilBlk 3CryBlk 4ReLoad 5OrdLdr 6String 7Direct 8Table 91byte 10Leave 11Naked 12AddNam
```

図 1.40: Hiew: `f_unsigned()` 関数

本質的には、次の3つのタスクを実行する必要があります。

- 最初のジャンプが常に起動しなければならない
- 2番目のジャンプが決して起動してはならない
- 3番目のジャンプが常にト起動しなければならない

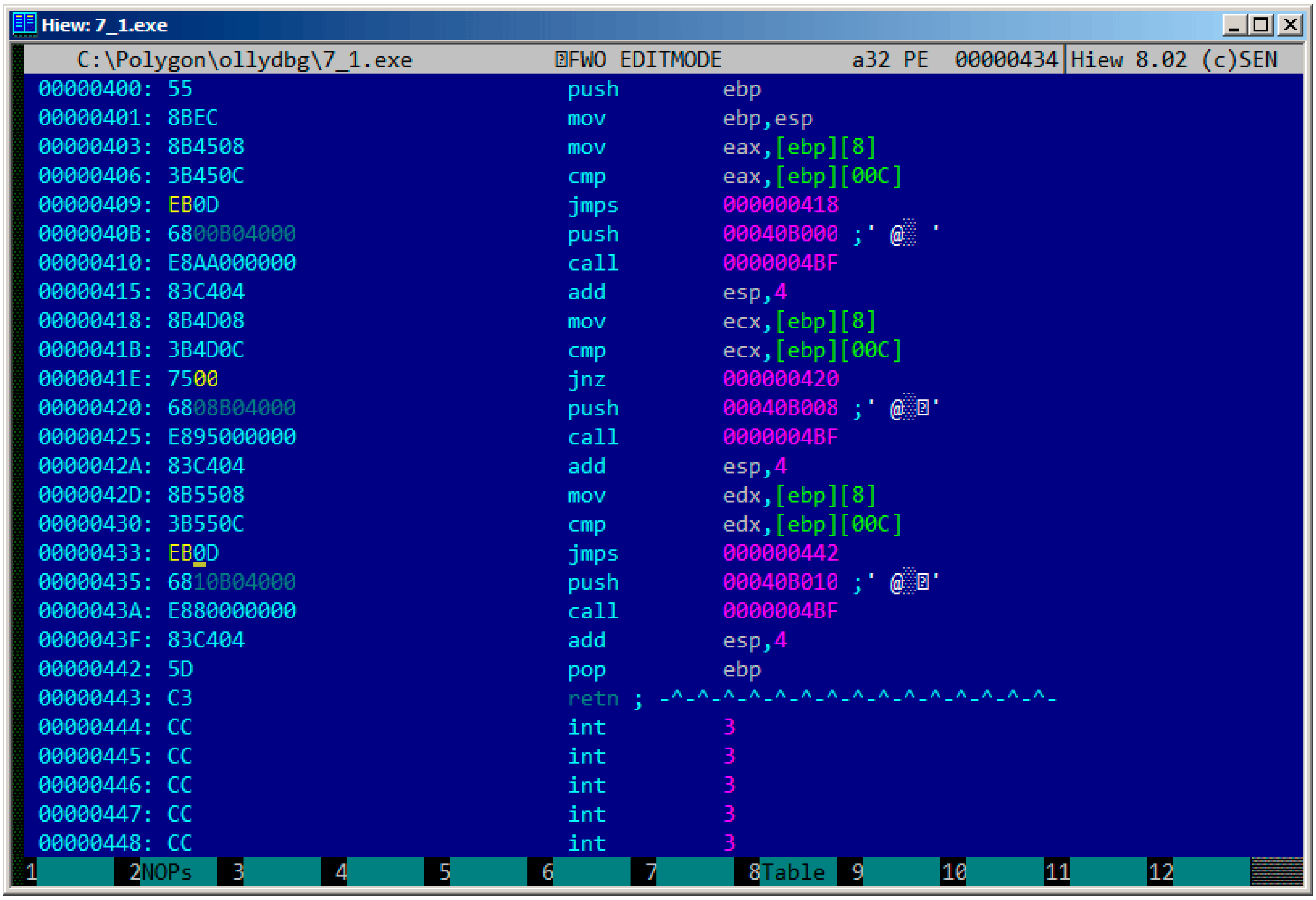
したがって、コードフローは常に2番目の `printf()` を通過し、「a==b」を出力するように指示できます。

3つの命令（またはバイト）をパッチする必要があります。

- 最初のジャンプはJMPになりますが、jump offsetは同じままです。
- 2回目のジャンプがトリガされることもありますが、いずれにしても次の命令にジャンプします。

  なぜなら、jump offsetを0に設定しているからです。これらの命令では、ジャンプオフセットが次の命令のアドレスに追加されます。オフセットが0の場合、ジャンプは制御を次の命令に移します。
- 私たちが最初のものと同様に JMP を置き換える3番目のジャンプは、常に起動します。

変更されたコードは次のとおりです。



```
Hiew: 7_1.exe
   C:\Polygon\ollydbg\7_1.exe             ⊡FWO EDITMODE          a32 PE   00000434 Hiew 8.02 (c)SEN
 00000400: 55                           push        ebp
 00000401: 8BEC                         mov         ebp,esp
 00000403: 8B4508                       mov         eax,[ebp][8]
 00000406: 3B450C                       cmp         eax,[ebp][00C]
 00000409: EB0D                         jmps        000000418
 0000040B: 6800B04000                   push        00040B000 ;' @░ '
 00000410: E8AA000000                   call        0000004BF
 00000415: 83C404                       add         esp,4
 00000418: 8B4D08                       mov         ecx,[ebp][8]
 0000041B: 3B4D0C                       cmp         ecx,[ebp][00C]
 0000041E: 7500                         jnz         000000420
 00000420: 6808B04000                   push        00040B008 ;' @░⊡'
 00000425: E895000000                   call        0000004BF
 0000042A: 83C404                       add         esp,4
 0000042D: 8B5508                       mov         edx,[ebp][8]
 00000430: 3B550C                       cmp         edx,[ebp][00C]
 00000433: EB0D                         jmps        000000442
 00000435: 6810B04000                   push        00040B010 ;' @░⊡'
 0000043A: E880000000                   call        0000004BF
 0000043F: 83C404                       add         esp,4
 00000442: 5D                           pop         ebp
 00000443: C3                           retn ; -^-^-^-^-^-^-^-^-^-^-^-^-^-^-^-^-
 00000444: CC                           int         3
 00000445: CC                           int         3
 00000446: CC                           int         3
 00000447: CC                           int         3
 00000448: CC                           int         3
1      2NOPs  3      4      5      6      7      8Table 9      10     11     12
```

図 1.41: Hiew: let's modify the f_unsigned() function

これらのジャンプのいずれかを変更することができなければ、printf() 呼び出しを1回だけ実行したいのですが、何回か実行することになるでしょう。

非最適化 **GCC**

非最適化 GCC 4.4.1 はほとんど同じコードを生成しますが、printf() ではなく puts() (1.5.4 on page 20) が生成されます。

最適化 **GCC**

実行される度にフラグが同じ値を持つ場合、鋭い読者はなぜ CMP が何度も実行されるのかと尋ねるかもしれません。

おそらく、最適化されたMSVCではこうはできませんが、GCC 4.8.1の最適化はより深刻です。

Listing 1.109: 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
```

また、`CALL puts / RETN` の代わりにここに `JMP`を入れています。

この種のトリックは後で説明します : **??** on page ??

この種のx86コードは、まれです。MSVC 2012のように、そのようなコードを生成することはできません。一方、アセンブリ言語プログラマは、`Jcc` 命令を積み重ねることができるという事実を十分に認識しています。

だから、どこかでそのような積み重ねを見ると、コードは手書きの可能性が高いです。

`f_unsigned()` 関数は巧妙に短いものではありません :

Listing 1.110: 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  ; この命令は削除することができます
        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
```

それにもかかわらず、3つではなく2つの `CMP` 命令があります。

したがって、GCC 4.8.1の最適化アルゴリズムはまだ完璧ではないでしょう。

ARM

32-bit ARM

最適化 Keil 6/2013 (ARMモード)

Listing 1.111: 最適化 Keil 6/2013 (ARMモード)

```
.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
```

ARMモードの多くの命令は、特定のフラグがセットされている場合にのみ実行できます。例えば、これは数字を比較するときによく使用されます。

例えば、ADD 命令は実際には内部で ADDAL と名付けられ、ALは常に、すなわち常に実行する。述語は、32ビットARM命令の4つの上位ビット（条件フィールド）でエンコードされます。無条件ジャンプの B 命令は、実際には条件付きで他の条件ジャンプと同様にエンコードされますが、条件フィールドには AL があり、フラグを無視して常に実行することを意味します。

ADRGT 命令は ADR と同じように動作しますが、前の CMP 命令が2つ（大きい方）を比較しながら、他の命令より大きな数値の1つを検出した場合にのみ実行されます。

次の BLGT 命令は BL と同じように動作し、比較の結果が（より大きい）場合にのみ実行されます。ADRGT は文字列 `a>b\n` へのポインタを R0 に書き込み、BLGT は `printf()` を呼び出します。したがって、-GT の後に続く命令は、R0 （a）の値が R4 （b）の値より大きい場合にのみ実行されます。

ADREQ 命令と BLEQ 命令が順方向に進みます。それらは ADR と BL のように動作しますが、最後の比較時にオペランドが等しい場合にのみ実行されます。`printf()` の実行によってフラグが改ざんされた可能性があるため、別の CMP がその前に配置されます。

次に、LDMGEFD を参照してください。この命令は LDMFD[90]のように機能しますが、一方の値が他方の値より大きいか等しい場合にのみ実行されます。`LDMGEFD SP!, {R4-R6,PC}` 命令は関数エピローグのように動作しますが、$a>=b$ の場合にのみトリガされ、その後に関数の実行が終了します。

しかし、その条件が満たされない場合、すなわち $a<b$ の場合、制御フローは次の 「`LDMFD SP!, {R4-R6,LR}` 命令に続き、これはもう1つの関数エピローグです。この命令は、R4-R6 だけでなくPCの代わりにLRも登録されているため、関数からは戻りません。最後の2つの命令は、文字列«a<b\n»を唯一の引数として `printf()` を呼び出します。`printf()` セクション（ 1.8.2 on page 52）の関数の戻り値ではなく、`printf()` 関数への無条件ジャンプを調べました。

f_unsigned は類似しており、ADRHI、BLHI、および LDMCSFD 命令のみが使用されています。これらの述部（*HI = Unsigned higher, CS = Carry Set (greater than or equal)*）は、前に説明したものと類似しています。

`main()` 関数にはそんなに新しい点はありません。

Listing 1.112: `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
.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
```

これは、ARMモードでの条件付きジャンプを取り除く方法です。

なぜこれがよいのでしょう？以下を読んでください：**??** on page ??

---

[90] LDMFD

x86では、CMOVcc 命令以外は MOV と同じですが、通常は CMP によって設定された特定のフラグが設定されている場合にのみ実行されます。

最適化 Keil 6/2013 (Thumbモード)

Listing 1.113: 最適化 Keil 6/2013 (Thumbモード)

```
.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
```

Thumbモードの B 命令だけが条件コードで補完されるため、Thumbコードはより一般的に見えます。

BLE は通常の条件ジャンプであり、*Less than or Equal* の意味です。BNE は *Not Equal* の意味です。BGE は *Greater than or Equal* の意味です。

f_unsigned は似ていますが、符号なしの値を扱う際には、BLS（*Unsigned lower or same*）および BCS（*Carry Set (Greater than or equal)*）命令しか使用されません。

ARM64: 最適化 GCC (Linaro) 4.9

Listing 1.114: f_signed()

```
f_signed:
; W0=a, W1=b
        cmp     w0, w1
        bgt     .L19    ; 大きければ (a>b)分岐
        beq     .L20    ; 等しければ (a==b)分岐
        bge     .L15    ; 大きい、または等しければ分岐(a>=b) (不可能)
        ; 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:   ; ここに来るのは不可能
        ret
.L20:
        adrp    x0, .LC10       ; "a==b"
        add     x0, x0, :lo12:.LC10
        b       puts
```

Listing 1.115: 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    ; 大きければ (a>b)分岐
        cmp     w19, w1
        beq     .L26    ; 等しければ (a==b)分岐
.L23:
        bcc     .L27    ; キャリーフラグがクリアされてたら分岐 (小さければ) (a<b)
; 関数エピローグ、ここに来るのは不可能
        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    ; 等しくなければ分岐
.L26:
        ldr     x19, [sp,16]
        adrp    x0, .LC10       ; "a==b"
        ldp     x29, x30, [sp], 48
        add     x0, x0, :lo12:.LC10
        b       puts
```

コメントはこの本の著者によって追加されました。目立ったことは、コンパイラはいくつかの条件がまったく不可能であることを認識していないため、決して実行できない場所ではデッドコードがあることです。

練習問題

これらの機能をサイズが少なくなるように手動で最適化し、新しい命令を追加せずに冗長な命令を削除してください。

MIPS

1つの特徴的なMIPS機能は、フラグが存在しないことです。明らかに、データ依存性の分析を簡素化するために行われました。

x86には SETcc に似た命令があります。SLT （「Set on Less Than」: 符号付きバージョン）と SLTU （符号なしバージョン）です。これらの命令は、条件が真であれば宛先レジスタの値を1に設定し、そうでない場合は0に設定します。

宛先レジスタは、BEQ (「Branch on Equal」) または BNE (「Branch on Not Equal」) を使用してチェックされ、ジャンプが発生することがあります。したがって、この命令ペアは比較および分岐のためにMIPSで使用されなければなりません。最初に関数の符号付きバージョンから始めましょう。

Listing 1.116: 非最適化 GCC 4.4.5 (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)
; 入力値をローカルスタックに格納する
.text:0000001C                 sw      $a0, 0x20+arg_0($fp)
.text:00000020                 sw      $a1, 0x20+arg_4($fp)
; リロードする
.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
; これは疑似命令です。実際は、"slt $v0,$v0,$v1"です。
; $v0<$v1 (b<a)なら$v0に1が設定され、そうでなければ0が設定されます
.text:00000030                 slt     $v0, $v1
; 条件が真でない場合、loc_5cにジャンプします。
; これは疑似命令です。 実際は、"beq $v0,$zero,loc_5c"です。
.text:00000034                 beqz    $v0, loc_5C
; "a>b"を表示して終了します
.text:00000038                 or      $at, $zero ; 分岐遅延スロット、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 ; 分岐遅延スロット、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
; a==bであるかどうかを調べ、真でなければloc_90にジャンプします。
.text:00000068                 bne     $v1, $v0, loc_90
.text:0000006C                 or      $at, $zero ; 分岐遅延スロット、NOP
; 条件が真なので、 "a==b"をプリントして終了する
.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 ; 分岐遅延スロット、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
; $v1<$v0（a <b）かどうかをチェックし、条件が真であれば$v0を1に設定する
.text:0000009C                 slt     $v0, $v1, $v0
; 条件が真でない場合（すなわち、$v0==0）、loc_c8にジャンプします
.text:000000A0                 beqz    $v0, loc_C8
.text:000000A4                 or      $at, $zero ; 分岐遅延スロット、NOP
; 条件が真であれば、 "a<b"をプリントして終了します
.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 ; 分岐遅延スロット、NOP
.text:000000C4                 lw      $gp, 0x20+var_10($fp)
```

```
.text:000000C8
; all 3 conditions were false, so just finish:
3つの条件はすべて偽でした。
.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 ; 分岐遅延スロット、NOP
.text:000000DC  # End of function f_signed
```

`SLT REG0, REG0, REG1` は、IDAによって短縮形式 `SLT REG0, REG1` に縮小されます。

実際には `BEQ REG, $ZERO, LABEL` の BEQZ 擬似命令もあります（「Branch if Equal to Zero」）。

符号なしバージョンはまったく同じですが、SLT の代わりに `SLTU`（符号なしバージョン、したがって「U」という名前）が使用されます。

Listing 1.117: 非最適化 GCC 4.4.5 (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
```

第**1.14.2**節絶対値の計算

簡単な関数の例。

```
int my_abs (int i)
{
        if (i<0)
                return -i;
        else
                return i;
};
```

最適化 **MSVC**

これは普通、どのようにコードが生成されるのかを示したものです。

Listing 1.118: 最適化 MSVC 2012 x64

```
i$ = 8
my_abs  PROC
; ECX = input
        test    ecx, ecx
; 入力値の符号をチェックする
; 符号が正の場合はNEG命令をスキップする
        jns     SHORT $LN2@my_abs
; 値を反転する
        neg     ecx
$LN2@my_abs:
; EAXに結果を準備
        mov     eax, ecx
        ret     0
my_abs  ENDP
```

GCC 4.9はほとんど同じです。

最適化 **Keil 6/2013: Thumb**モード

Listing 1.119: 最適化 Keil 6/2013: Thumbモード

```
my_abs PROC
        CMP      r0,#0
; 入力値はゼロに等しいかゼロより大きいか
; RSBS命令をスキップする
```

```
        BGE      |L0.6|
; 入力値を0から減算する
        RSBS     r0,r0,#0
|L0.6|
        BX       lr
        ENDP
```

ARMにはネゲート命令がないため、Keilコンパイラは「逆引き命令」を使用します。これは逆のオペランドで減算するだけです。

最適化 **Keil 6/2013: ARM**モード

ARMモードでは、いくつかの命令に条件コードを追加することができます。そのため、Keilコンパイラは次のように処理します。

Listing 1.120: 最適化 Keil 6/2013: ARMモード

```
my_abs PROC
        CMP      r0,#0
; 入力値が0より小さい場合にのみ "逆引"命令を実行する
        RSBLT    r0,r0,#0
        BX       lr
        ENDP
```

今度は条件付きジャンプはありません。これは良いですね。: **??** on page ??

非最適化 **GCC 4.9 (ARM64)**

ARM64には、否定するための命令 `NEG` があります。

Listing 1.121: 最適化 GCC 4.9 (ARM64)

```
my_abs:
        sub     sp, sp, #16
        str     w0, [sp,12]
        ldr     w0, [sp,12]
; 入力値をWZRレジスタの内容と比較する
; （常にゼロを保持する）
        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 1.122: 最適化 GCC 4.4.5 (IDA)

```
my_abs:
; $a0<0ならジャンプ:
                bltz    $a0, locret_10
; 入力値 ($a0)を$v0に設定してリターン
                move    $v0, $a0
                jr      $ra
                or      $at, $zero ; branch delay slot, NOP
locret_10:
; 入力値を反転し、$v0に保存する:
                jr      $ra
; これは疑似命令です。実際には、"subu $v0,$zero,$a0" ($v0=0-$a0)です。
                negu    $v0, $a0
```

ここでは BLTZ（「Branch if Less Than Zero」）という新しい命令があります。

NEGU 擬似命令もあります。これはゼロからの減算だけです。SUBU と NEGU の両方の「U」接尾辞は、整数オーバーフローの場合に発生する例外がないことを意味します。

Branchless version?

このコードを分岐がないバージョンにすることもできます。これについては、後述の **??** on page ??を参照してください。

第1.14.3節三項条件演算子

C/C++ の三項条件演算子の構文は次のとおりです。

```
expression ? expression : expression
```

次に例を示します。

```
const char* f (int a)
{
        return a==10 ? "it is ten" : "it is not ten";
};
```

x86

古いコンパイラと最適化していないコンパイラは、if/else 文が使用されたかのようにアセンブリコードを生成します。

Listing 1.123: 非最適化 MSVC 2008

```
$SG746  DB      'it is ten', 00H
$SG747  DB      'it is not ten', 00H

tv65 = -4 ; this will be used as a temporary variable
_a$ = 8
_f      PROC
        push    ebp
        mov     ebp, esp
        push    ecx
; 入力値と10を比較
        cmp     DWORD PTR _a$[ebp], 10
; 同じでなければ、$LN3@fにジャンプ
        jne     SHORT $LN3@f
; 文字列へのポインタを一時変数に保存
        mov     DWORD PTR tv65[ebp], OFFSET $SG746 ; 'it is ten'
; exitにジャンプ
        jmp     SHORT $LN4@f
$LN3@f:
; 文字列へのポインタを一時変数に保存
        mov     DWORD PTR tv65[ebp], OFFSET $SG747 ; 'it is not ten'
$LN4@f:
; exitです。文字列へのポインタを一時変数からEAXにコピー
        mov     eax, DWORD PTR tv65[ebp]
        mov     esp, ebp
        pop     ebp
        ret     0
_f      ENDP
```

Listing 1.124: 最適化 MSVC 2008

```
$SG792  DB      'it is ten', 00H
$SG793  DB      'it is not ten', 00H

_a$ = 8 ; size = 4
_f      PROC
```

```
; 入力値と10を比較
        cmp     DWORD PTR _a$[esp-4], 10
        mov     eax, OFFSET $SG792 ; 'it is ten'
; 同じなら$LN4@fにジャンプ
        je      SHORT $LN4@f
        mov     eax, OFFSET $SG793 ; 'it is not ten'
$LN4@f:
        ret     0
_f      ENDP
```

新しいコンパイラはより簡潔です。

Listing 1.125: 最適化 MSVC 2012 x64

```
$SG1355 DB      'it is ten', 00H
$SG1356 DB      'it is not ten', 00H

a$ = 8
f       PROC
; 両方の文字列のポインタをロードする
        lea     rdx, OFFSET FLAT:$SG1355 ; 'it is ten'
        lea     rax, OFFSET FLAT:$SG1356 ; 'it is not ten'
; 入力値と10を比較
        cmp     ecx, 10
; 同じなら、値をRDXからコピー("it is ten")
; 異なるなら、何もしない。文字列へのポインタ"it is not ten"はまだRAXにある。
        cmove   rax, rdx
        ret     0
f       ENDP
```

x86用の最適化 GCC 4.8も `CMOVcc` 命令を使用し、非最適化GCC 4.8は条件付きジャンプを使用します。

ARM

ARMモード用の 最適化 Keilでは、条件付き命令 `ADRcc` を使います。

Listing 1.126: 最適化 Keil 6/2013 (ARMモード)

```
f PROC
; 入力値と10を比較
        CMP      r0,#0xa
; 結果が同じか比較し同じなら、"it is ten"文字列へのポインタをR0にコピー
        ADREQ    r0,|L0.16| ; "it is ten"
; 結果が同じか比較し異なるなら、"it is not ten"文字列へのポインタをR0にコピー
        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
```

手動で介入しなければ、2つの命令 `ADREQ` と `ADRNE` を同じときにで実行することはできません。

Thumbモードでは、最適化 Keilは、条件付きフラグをサポートするロード命令がないため、条件付きジャンプ命令を使用する必要があります。

Listing 1.127: 最適化 Keil 6/2013 (Thumbモード)

```
f PROC
; 入力値と10を比較
        CMP      r0,#0xa
; 同じなら、|L0.8|にジャンプ
        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

ARM64の 最適化 GCC（Linaro）4.9でも、条件付きジャンプが使用されます。

Listing 1.128: 最適化 GCC (Linaro) 4.9

```
f:
        cmp     x0, 10
        beq     .L3             ; 等しければ分岐
        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"
```

これは、ARM64には32ビットARMモードの `ADRcc` やx86の `CMOVcc` などの条件フラグを伴った単純なロード命令がないためです。

しかし、「Conditional SELect」命令（CSEL）[*ARM Architecture Reference Manual, ARMv8, for ARMv8-A architecture profile*, (2013)p390, C5.5] を使用していますが、GCC 4.9ではこのようなコードの中で使用するには十分スマートではないようです。

MIPS

残念ながら、MIPS用のGCC 4.4.5はそれほどスマートではありません。

Listing 1.129: 最適化 GCC 4.4.5 (アセンブリ出力)

```
$LC0:
        .ascii  "it is not ten\000"
$LC1:
        .ascii  "it is ten\000"
f:
        li      $2,10                   # 0xa
; $a0と10を比較し、等しければ分岐
        beq     $4,$2,$L2
        nop ; branch delay slot

; "it is not ten"文字列へのアドレスを$v0に残しつつリターン
        lui     $2,%hi($LC0)
        j       $31
        addiu   $2,$2,%lo($LC0)

$L2:
; "it is ten"文字列へのアドレスを$v0に残しつつリターン
        lui     $2,%hi($LC1)
        j       $31
        addiu   $2,$2,%lo($LC1)
```

if/else の方法で書き直しましょう

```
const char* f (int a)
{
        if (a==10)
                return "it is ten";
        else
                return "it is not ten";
};
```

興味深いことに、x86用のGCC 4.8の最適化は、この場合に `CMOVcc` を使用することもできました。

Listing 1.130: 最適化 GCC 4.8

```
.LC0:
        .string "it is ten"
.LC1:
        .string "it is not ten"
f:
.LFB0:
; 入力値と10を比較
        cmp     DWORD PTR [esp+4], 10
        mov     edx, OFFSET FLAT:.LC1 ; "it is not ten"
        mov     eax, OFFSET FLAT:.LC0 ; "it is ten"
; 比較結果が同じでなければ、EDXの値をEAXにコピー
; そうでなければ、何もしない
        cmovne  eax, edx
        ret
```

ARMモードの 最適化 Keilでは、リスト1.126 と同じコードが生成されます。

しかし、MSVC 2012の最適化は（まだ）あまり良くありません。

結論

コンパイラを最適化するとどうして条件付きジャンプを取り除こうとするのでしょうか？以下を読んでください：**??** on page ??

第**1.14.4**節最小値と最大値の取得

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 1.131: 非最適化 MSVC 2013

```
_a$ = 8
_b$ = 12
_my_min PROC
        push    ebp
        mov     ebp, esp
        mov     eax, DWORD PTR _a$[ebp]
; AとBを比較
        cmp     eax, DWORD PTR _b$[ebp]
; AがB以上の場合にジャンプする
        jge     SHORT $LN2@my_min
; それ以外ではAをEAXにリロードして終了する
        mov     eax, DWORD PTR _a$[ebp]
        jmp     SHORT $LN3@my_min
        jmp     SHORT $LN3@my_min ; これは冗長なJMP命令
$LN2@my_min:
; Bをリターン
        mov     eax, DWORD PTR _b$[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$[ebp]
; AとBを比較
        cmp     eax, DWORD PTR _b$[ebp]
; AがB以下の場合ジャンプする
        jle     SHORT $LN2@my_max
; それ以外ではAをEAXにリロードして終了する
        mov     eax, DWORD PTR _a$[ebp]
        jmp     SHORT $LN3@my_max
        jmp     SHORT $LN3@my_max ; これは冗長なJMP命令
$LN2@my_max:
; Bをリターン
        mov     eax, DWORD PTR _b$[ebp]
$LN3@my_max:
        pop     ebp
        ret     0
_my_max ENDP
```

これらの2つの機能は条件ジャンプ命令でのみ異なります。最初の命令では JGE（「Jump if Greater or Equal」）が使用され、2番目の場合は JLE（「Jump if Less or Equal」）が使用されます。

各関数には不必要な JMP 命令が1つありますが、おそらく誤って残っています。

分岐

ThumbモードのARMは、x86コードを思い起こします。

Listing 1.132: 最適化 Keil 6/2013 (Thumbモード)

```
my_max PROC
; R0=A
; R1=B
; AとBを比較
        CMP      r0,r1
; AがBより大きければ分岐
        BGT      |L0.6|
; それ以外(A<=B)の場合は、R1(B)をリターン
        MOVS     r0,r1
|L0.6|
; リターン
        BX       lr
        ENDP

my_min PROC
; R0=A
; R1=B
; AとBを比較
```

```
        CMP      r0,r1
; AがBより小さければ分岐
        BLT      |L0.14|
; それ以外(A>=B)の場合は、R1(B)をリターン
        MOVS     r0,r1
|L0.14|
; リターン
        BX       lr
        ENDP
```

関数は分岐命令が異なります。BGT と BLT です。ARMモードでは条件付きの接尾辞を使用することができるため、コードは短くなります。

`MOVcc` は、条件が満たされた場合にのみ実行されます。

Listing 1.133: 最適化 Keil 6/2013 (ARMモード)

```
my_max PROC
; R0=A
; R1=B
; AとBを比較
        CMP      r0,r1
; BをR0に入れて、AではなくBをリターン
; A<=Bのときにのみ、この命令は実行されます (つまり、 LE - Less or Equal)
; 命令が実行されない場合(A>Bのとき)、AはR0レジスタにあります。
        MOVLE    r0,r1
        BX       lr
        ENDP

my_min PROC
; R0=A
; R1=B
; AとBを比較
        CMP      r0,r1
; BをR0に入れて、AではなくBをリターン
; A>=Bのときにのみ、この命令は実行されます (つまり、 GE - Greater or Equal)
; 命令が実行されない場合(A<Bのとき)、AはR0レジスタにあります。
        MOVGE    r0,r1
        BX       lr
        ENDP
```

最適化 GCC 4.8.1とMSVC 2013の最適化では、ARMの `CMOVcc` に似た `MOVcc` 命令を使用できます。

Listing 1.134: 最適化 MSVC 2013

```
my_max:
        mov     edx, DWORD PTR [esp+4]
        mov     eax, DWORD PTR [esp+8]
; EDX=A
; EAX=B
; AとBを比較
        cmp     edx, eax
; A>=Bなら、Aの値をEAXにロード
; それ以外(A<B)の場合は、アイドル命令
; the instruction idle if otherwise (if A<B)
        cmovge  eax, edx
        ret

my_min:
        mov     edx, DWORD PTR [esp+4]
        mov     eax, DWORD PTR [esp+8]
; EDX=A
; EAX=B
; AとBを比較
        cmp     edx, eax
; A<=Bなら、Aの値をEAXにロード
; それ以外(A>B)の場合は、アイドル命令
        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;
};
```

いくつかの不要な値のシャッフルがありますが、コードは理解できます。

Listing 1.135: 非最適化 GCC 4.9.1 ARM64

```
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
```

分岐なし

スタックから関数の引数をロードする必要はありません。レジスタにすでに入っています。

Listing 1.136: 最適化 GCC 4.9.1 x64

```
my_max:
; RDI=A
; RSI=B
; AとBを比較
        cmp     rdi, rsi
; Bを戻り値としてRAXにコピー
```

```
        mov     rax, rsi
; A>=Bの場合、A(RDI)を戻り値としてRAXにコピー
; それ以外(A<B)では、アイドル命令
        cmovge  rax, rdi
        ret

my_min:
; RDI=A
; RSI=B
; AとBを比較
        cmp     rdi, rsi
; Bを戻り値としてRAXにコピー
        mov     rax, rsi
; A<=Bの場合、A(RDI)を戻り値としてRAXにコピー
; それ以外(A>B)では、アイドル命令
        cmovle  rax, rdi
        ret
```

MSVC 2013はほぼ同じです。

ARM64にはARMの `MOVcc` またはx86の `CMOVcc` と同じように機能する `CSEL` 命令がありますが、その名前は「Conditional SELect」とは異なります。

Listing 1.137: 最適化 GCC 4.9.1 ARM64

```
my_max:
; X0=A
; X1=B
; AとBを比較
        cmp     x0, x1
; X0>=X1 または A>=B (Greater or Equal)の場合、X0(A)を選択する
; A<Bの場合、X1 (B)を選択する
        csel    x0, x0, x1, ge
        ret

my_min:
; X0=A
; X1=B
; AとBを比較
        cmp     x0, x1
; X0<=X1 または A<=B (Less or Equal)の場合、X0(A)を選択する
; A>Bの場合、X1 (B)を選択する
        csel    x0, x0, x1, le
        ret
```

MIPS

残念ながら、MIPS用のGCC 4.4.5はあまり良くありません。

Listing 1.138: 最適化 GCC 4.4.5 (IDA)

```
my_max:
; $a1<$a0なら、$v1に1を設定し、それ以外($a1>$a0)ではクリアする
                slt     $v1, $a1, $a0
; $v1が0(または $a1>$a0)ならジャンプ
                beqz    $v1, locret_10
; これは分岐遅延スロットです
; 分岐が実行された場合に、$a1を$v0にコピー
                move    $v0, $a1
; 分岐は実行されず、$a0を$v0にコピー
                move    $v0, $a0

locret_10:
                jr      $ra
                or      $at, $zero ; 分岐遅延スロット、NOP

; min()関数は同じですが、SLT命令の入力オペランドはスワップされます
my_min:
                slt     $v1, $a0, $a1
                beqz    $v1, locret_28
                move    $v0, $a1
                move    $v0, $a0

locret_28:
                jr      $ra
                or      $at, $zero ; branch delay slot, NOP
```

分岐遅延スロットを忘れないでください。最初の `MOVE` は `BEQZ` の前に実行され、2番目の `MOVE` は分岐が実行されなかった場合にのみ実行されます。

第1.14.5節結論

x86

条件付きジャンプの基本骨格は次のとおりです。

Listing 1.139: x86

```
CMP register, register/value
Jcc true ; cc=condition code
false:
... 比較結果が偽の場合に実行されるコード ...
JMP exit
true:
... 比較結果が真の場合に実行されるコード ...
exit:
```

ARM

Listing 1.140: ARM

```
CMP register, register/value
Bcc true ; cc=condition code
false:
... 比較結果が偽の場合に実行されるコード ...
JMP exit
true:
... 比較結果が真の場合に実行されるコード ...
exit:
```

MIPS

Listing 1.141: Check for zero

```
BEQZ REG, label
...
```

Listing 1.142: Check for less than zero using pseudoinstruction

```
BLTZ REG, label
...
```

Listing 1.143: Check for equal values

```
BEQ REG1, REG2, label
...
```

Listing 1.144: Check for non-equal values

```
BNE REG1, REG2, label
...
```

Listing 1.145: Check for less than (signed)

```
SLT REG1, REG2, REG3
BEQ REG1, label
...
```

Listing 1.146: Check for less than (unsigned)

```
SLTU REG1, REG2, REG3
BEQ REG1, label
...
```

Branchless

条件文の本体が非常に短い場合は、ARMの `MOVcc` （ARMモードの場合)、ARM64の場合は `CSEL` 、x86の場合は `CMOVcc` の条件付き移動命令を使用できます。

ARM

命令によっては、ARMモードで条件付き接尾辞を使用することもできます。

Listing 1.147: ARM (ARMモード)

```
CMP register, register/value
instr1_cc ; 条件コードが真の場合、何らかの命令が実行されます
instr2_cc ; 他の条件コードが真の場合、他の命令が実行されます
... etc...
```

もちろん、CPUフラグがいずれかで変更されない限り、条件付きコードの接尾辞付き命令の数に制限はありません。

Thumbモードには `IT` 命令があり、次の4つの命令に条件付きサフィックスを追加できます。詳しくは、**??** on page ??を参照してください。

Listing 1.148: ARM (Thumbモード)

```
CMP register, register/value
ITEEE EQ ; 接尾辞を設定します: if-then-else-else-else
instr1   ; 条件が真であれば命令が実行されます
instr2   ; 条件が偽であれば命令が実行されます
instr3   ; 条件が偽であれば命令が実行されます
instr4   ; 条件が偽であれば命令が実行されます
```

第1.14.6節練習問題

（ARM64）すべての条件付きジャンプ命令を削除し、`CSEL` 命令を使用して、リスト1.128 のコードを書き直してみてください。

第1.15節switch()/case/default

第1.15.1節

```
#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
};
```

結論

リスト**??**.

第**1.15.2**節練習問題

練習問題 **#1**

第**1.16**節ループ

第**1.16.1**節練習問題

- http://challenges.re/54
- http://challenges.re/55
- http://challenges.re/56
- http://challenges.re/57

第**1.17**節文字列に関する加筆

第**1.17.1**節**strlen()**

```
int my_strlen (const char * str)
{
        const char *eos = str;

        while( *eos++ ) ;

        return( eos - str - 1 );
}

int main()
{
        // test
        return my_strlen("hello!");
};
```

ARM

第**1.18**節算術命令を他の命令に置換する

第**1.18.1**節練習問題

- http://challenges.re/59

第1.19節配列

第1.19.1節

```
#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;
};
```

第1.19.2節

第1.19.3節練習問題

- http://challenges.re/62
- http://challenges.re/63
- http://challenges.re/64
- http://challenges.re/65
- http://challenges.re/66

第1.20節構造体

第1.20.1節UNIX: struct tm

第1.20.2節

第1.20.3節練習問題

- http://challenges.re/71
- http://challenges.re/72

第1.21節

第1.21.1節

Listing 1.149: : 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 */
}
```

第**1.22**節

第**1.22.1**節

Dominic Sweetman, *See MIPS Run, Second Edition*, (2010).

第 2 章

Japanese text placeholder

第 **3** 章

第 4 章

第4.1節Linux

第4.2節Windows NT

第4.2.1節Windows SEH

SEH

[Matt Pietrek, *A Crash Course on the Depths of Win32™ Structured Exception Handling*, (1997)][1], [Igor Skochinsky, *Compiler Internals: Exceptions and RTTI*, (2012)] [2].

---

[1] 以下で利用可能 `http://go.yurichev.com/17293`

[2] 以下で利用可能 `http://go.yurichev.com/17294`

第 5 章

ツール

> Now that Dennis Yurichev has made this book free (libre), it is a contribution to the world of free knowledge and free education. However, for our freedom's sake, we need free (libre) reverse engineering tools to replace the proprietary tools described in this book.
>
> Richard M. Stallman

第5.1節バイナリ解析

Tools you use when you don't run any process.

- (Free, open-source) *ent*[1]: entropy analyzing tool. Read more about entropy: **??** on page ??.
- *Hiew*[2]: for small modifications of code in binary files. Has assembler/disassembler.
- (Free, open-source) *GHex*[3]: simple hexadecimal editor for Linux.
- (Free, open-source) *xxd* and *od*: standard UNIX utilities for dumping.
- (Free, open-source) *strings*: *NIX tool for searching for ASCII strings in binary files, including executable ones. Sysinternals has alternative[4] supporting wide char strings (UTF-16, widely used in Windows).
- (Free, open-source) *Binwalk*[5]: analyzing firmware images.
- (Free, open-source) *binary grep*: a small utility for searching any byte sequence in a big pile of files, including non-executable ones: GitHub. There is also rafind2 in rada.re for the same purpose.

第5.1.1節ディスアセンブラ

- *IDA*. An older freeware version is available for download [6]. ホットキーチートシート: **??** on page ??
- *Binary Ninja*[7]
- (Free, open-source) *zynamics BinNavi*[8]
- (Free, open-source) *objdump*: simple command-line utility for dumping and disassembling.
- (Free, open-source) *readelf*[9]: dump information about ELF file.

---

[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`
[9] `https://sourceware.org/binutils/docs/binutils/readelf.html`

第5.1.2節デコンパイラ

There is only one known, publicly available, high-quality decompiler to C code: *Hex-Rays*: hex-rays.com/products/decompiler/

Read more about it: **??** on page ??.

第5.1.3節Patch comparison/diffing

You may want to use it when you compare original version of some executable and patched one, in order to find what has been patched and why.

- (Free) *zynamics BinDiff*[10]
- (Free, open-source) *Diaphora*[11]

第5.2節ライブ解析

Tools you use on a live system or during running of a process.

第5.2.1節デバッガ

- (Free) *OllyDbg*. Very popular user-mode win32 debugger[12]. ホットキーチートシート: **??** on page ??
- (Free, open-source) *GDB*. Not quite popular debugger among reverse engineers, because it's intended mostly for programmers. Some commands: **??** on page ??. There is a visual interface for GDB, "GDB dashboard"[13].
- (Free, open-source) *LLDB*[14].
- *WinDbg*[15]: kernel debugger for Windows.
- *IDA* has internal debugger.
- (Free, open-source) *Radare* AKA rada.re AKA r2[16]. A GUI also exists: *ragui*[17].
- (Free, open-source) *tracer*. The author often uses *tracer* [18] instead of a debugger.

  The author of these lines stopped using a debugger eventually, since all he needs from it is to spot function arguments while executing, or registers state at some point. Loading a debugger each time is too much, so a small utility called *tracer* was born. It works from command line, allows intercepting function execution, setting breakpoints at arbitrary places, reading and changing registers state, etc.

  N.B.: the *tracer* isn't evolving, because it was developed as a demonstration tool for this book, not as everyday tool.

第5.2.2節ライブラリコールトレース

ltrace[19].

---

[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
[19] `http://www.ltrace.org/`

第**5.2.3**節システムコールトレース

strace / dtruss

It shows which system calls (syscalls(**??** on page ??)) are called by a process right now.

For example:

```
# 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 has dtruss for doing the same.

Cygwin also has strace, but as far as it's known, it works only for .exe-files compiled for the cygwin environment itself.

第**5.2.4**節ネットワーク傍受

Sniffing is intercepting some information you may be interested in.

(Free, open-source) *Wireshark*[20] for network sniffing. It has also capability for USB sniffing[21].

Wireshark has a younger (or older) brother *tcpdump*[22], simpler command-line tool.

第**5.2.5**節**Sysinternals**

(Free) Sysinternals (developed by Mark Russinovich) [23]. At least these tools are important and worth studying: Process Explorer, Handle, VMMap, TCPView, Process Monitor.

第**5.2.6**節**Valgrind**

(Free, open-source) a powerful tool for detecting memory leaks: `http://valgrind.org/`. Due to its powerful JIT[24] mechanism, Valgrind is used as a framework for other tools.

第**5.2.7**節エミュレータ

- (Free, open-source) *QEMU*[25]: emulator for various CPUs and architectures.
- (Free, open-source) *DosBox*[26]: MS-DOS emulator, mostly used for retrogaming.
- (Free, open-source) *SimH*[27]: emulator of ancient computers, mainframes, etc.

---

[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/`

第5.3節他のツール

Microsoft Visual Studio Express [28]: Stripped-down free version of Visual Studio, convenient for simple experiments.

Some useful options: **??** on page ??.

There is a website named “Compiler Explorer”, allowing to compile small code snippets and see output in various GCC versions and architectures (at least x86, ARM, MIPS): `http://godbolt.org/`—I would have used it myself for the book if I would know about it!

第5.3.1節電卓

Good calculator for reverse engineer’s needs should support at least decimal, hexadecimal and binary bases, as well as many important operations like XOR and shifts.

- IDA has built-in calculator (“?”).
- rada.re has *rax2*.
- `https://github.com/DennisYurichev/progcalc`
- As a last resort, standard calculator in Windows has programmer’s mode.

第5.4節何か足りないものは？

If you know a great tool not listed here, please drop a note:
`dennis@yurichev.com`.

第5.5節

第5.6節

Pierre Capillon – Black-box cryptanalysis of home-made encryption algorithms: a practical case study.

---

[28] visualstudio.com/en-US/products/visual-studio-express-vs

第 **6** 章

その他

第 7 章

読むべき本/ブログ

第7.1節本と他の資料

第7.1.1節リバースエンジニアリング

- 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)

Also, Kris Kaspersky's books.

第7.1.2節Windows

- Mark Russinovich, *Microsoft Windows Internals*

:

- Microsoft: Raymond Chen
- nynaeve.net

第7.1.3節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] 以下で利用可能 http://go.yurichev.com/17274

[2] 以下で利用可能 http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2013/n3690.pdf.

[3] 以下で利用可能 http://agner.org/optimize/optimizing_cpp.pdf.

[4] 以下で利用可能 http://go.yurichev.com/17291

[5] 以下で利用可能 http://yurichev.com/C-book.html

[6] 以下で利用可能 https://yurichev.com/mirrors/C/JPL_Coding_Standard_C.pdf

第7.1.4節x86 / x86-64

- Intel manuals[7]
- AMD manuals[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)]

Somewhat outdated, but still interesting to read:

Michael Abrash, *Graphics Programming Black Book*, 1997[11] (he is known for his work on low-level optimization for such projects as Windows NT 3.1 and id Quake).

第7.1.5節ARM

- ARM manuals[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]

第7.1.6節アセンブリ言語

Richard Blum — Professional Assembly Language.

第7.1.7節Java

[Tim Lindholm, Frank Yellin, Gilad Bracha, Alex Buckley, *The Java(R) Virtual Machine Specification / Java SE 7 Edition*] [15].

第7.1.8節UNIX

Eric S. Raymond, *The Art of UNIX Programming*, (2003)

第7.1.9節プログラミング一般

- Brian W. Kernighan, Rob Pike, *Practice of Programming*, (1999)
- Henry S. Warren, *Hacker's Delight*, (2002).
- (For hard-core geeks with computer science and mathematical background) Donald E. Knuth, *The Art of Computer Programming*.

---

[7] 以下で利用可能 http://www.intel.com/content/www/us/en/processors/architectures-software-developer-manuals.html

[8] 以下で利用可能 http://developer.amd.com/resources/developer-guides-manuals/

[9] 以下で利用可能 http://agner.org/optimize/microarchitecture.pdf

[10] 以下で利用可能 http://www.agner.org/optimize/calling_conventions.pdf

[11] 以下で利用可能 https://github.com/jagregory/abrash-black-book

[12] 以下で利用可能 http://infocenter.arm.com/help/index.jsp?topic=/com.arm.doc.subset.architecture.reference/index.html

[13] 以下で利用可能 http://yurichev.com/mirrors/ARMv8-A_Architecture_Reference_Manual_(Issue_A.a).pdf

[14] 以下で利用可能 http://go.yurichev.com/17273

[15] 以下で利用可能 https://docs.oracle.com/javase/specs/jvms/se7/jvms7.pdf; http://docs.oracle.com/javase/specs/jvms/se7/html/

第**7.1.10**節暗号学

- 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] 以下で利用可能 `https://www.crypto101.io/`

[17] 以下で利用可能 `https://crypto.stanford.edu/~dabo/cryptobook/`

Afterword

第7.2節Questions?

恥ずかしがらずに著者へメールしてみよう `<dennis@yurichev.com>`. この本への新たなコンテンツについての提案がありますか？怖がらずにどんな訂正でも送ってください (文法ミスも含め (私の日本語がとってもひどいのを見てるでしょ？))

The author is working on the book a lot, so the page and listing numbers, etc., are changing very rapidly. Please do not refer to page and listing numbers in your emails to me. There is a much simpler method: make a screenshot of the page, in a graphics editor underline the place where you see the error, and send it to the author. He'll fix it much faster. And if you familiar with git and LaTeX you can fix the error right in the source code:

GitHub.

Do not worry to bother me while writing me about any petty mistakes you found, even if you are not very confident. I'm writing for beginners, after all, so beginners' opinions and comments are crucial for my job.

Japanese text placeholder

OS	Japanese text placeholder	viii
PL	Japanese text placeholder	vi
ROM	Japanese text placeholder	80
ALU	Japanese text placeholder	25
RA	リターンアドレス	21
PE	Portable Executable	5
SP	stack pointer. SP/ESP/RSP in x86/x64. SP in ARM.	18
PC	Program Counter. IP/EIP/RIP in x86/64. PC in ARM.	19
LR	Link Register	6
IDA	Japanese text placeholder	6
MSVC	Microsoft Visual C++	
AKA	Japanese text placeholder	29
CRT	C Runtime library	10
CPU	Central Processing Unit	2
CISC	Complex Instruction Set Computing	19
RISC	Reduced Instruction Set Computing	2
BSS	Block Started by Symbol	24
DBMS	Database Management Systems	vi
ISA	Instruction Set Architecture	2
SEH	Structured Exception Handling	36
ELF	Linuxを含め*NIXシステムで広く使用される実行ファイルフォーマット	78
NOP	No Operation	6
BEQ	(PowerPC, ARM) Branch if Equal	94
RAM	Random-Access Memory	3
GCC	GNU Compiler Collection	3
ASCIIZ	ASCII Zero ( )	91

VM Virtual Memory

GDB GNU Debugger 46

FP Frame Pointer 23

STMFD Store Multiple Full Descending ()

LDMFD Load Multiple Full Descending ()

STMED Store Multiple Empty Descending () 29

LDMED Load Multiple Empty Descending () 29

STMFA Store Multiple Full Ascending () 29

LDMFA Load Multiple Full Ascending () 29

STMEA Store Multiple Empty Ascending () 29

LDMEA Load Multiple Empty Ascending () 29

JIT Just-In-Time compilation 160

EOF End of File 84

URL Uniform Resource Locator 4

用語集

OSが提供する大きなメモリの塊のことで､アプリケーションが好きなように分割することができる｡malloc()/free()を呼び出して使用する. 29

デクリメント 1の減算 . 18

インクリメント 1の加算 . 15, 19

product 乗算結果. 96

スタックポインタ スタックの場所を示すレジスタ. 10, 19, 29, 34, 41, 53, 54, 72, 98, 169

anti-pattern 一般に、よくないと考えられるやり方 . 31, 75

callee 呼び出された関数. 31, 45, 65, 85, 96, 98, 100

caller 呼び出し元の関数. 5–8, 10, 29, 45, 85, 96, 97, 99

endianness : **??** on page ??バイトオーダー. 77

GiB ギガバイト : 2^{30} または1024メガバイトまたは1073741824バイト . 15

jump offset JMP命令またはJcc命令のオペコードの一部を次の命令のアドレスに追加する必要があります。これが新しいPCの計算方法です。負となる場合もあります . 92, 132

leaf function 他の関数から呼び出されない関数. 27, 31

link register (RISC) リターンアドレスが保存されるレジスタ。これはleaf functionをスタックを使わずに呼び出すのを可能にする. 31

reverse engineering 時にはクローンするため、どうやって動いているのかを理解しようとする行為 . iv

stack frame 現在の関数に固有の情報（ローカル変数、関数の引数、RAなど）を含むスタックの一部 . 66, 96, 97

stdout standard output. 21, 34

thunk function 単一の役割だけ持つ小さな関数 : 他の関数を呼び出す等. 22

索引