Inżynieria wsteczna dla początkujących

(Rozumienie języka maszynowego)

Dlaczego aż dwa tytuły? Przeczytaj tutaj: **??**.

Dennis Yurichev
<dennis@yurichev.com>

Wersja tekstu (3 września 2020).

Aktualna wersja (a także wersja rosyjska) tego tekstu znajduje się na beginners.re.

Potrzebujemy tłumaczy!

Jeśli chcesz pomóc z tłumaczeniem tej książki na języki inne niż angielski i rosyjski, prześlij mailem fragment przetłumaczonego tekstu (obojętnie jakiej długości) a ja go dodam do kodu źródłowego.

Nie pytaj czy powinieneś tłumaczyć. Po prostu zrób coś. Przestałem odpowiadać na wiadomości “co powinienem zrobić”.

Czytać tutaj.

Statystyka tłumaczeń na inne języki znajduje się tutaj: https://beginners.re/.

Prędkość nie gra roli, jako że jest to projekt open-source. Twoje imię pojawi się obok imion innych uczestników projektu. Koreański, chiński i perski są zarezerwowane przez wydawców. Wersją angielską i rosyjską zajmuję się sam, ale mój angielski nadal jest daleki od ideału, więc będę wdzięczny za korekty. Nawet mój rosyjski nie jest idealny, więc będę wdzięczny również za korekty tekstu w języku rosyjskim!

Śmiało piszcie do mnie: <dennis@yurichev.com> lub <dennis.yurichev@gmail.com>.

Skrócony spis treści

Spis treści

Przedmowa

Skąd dwa tytuły?

W latach 2014-2018 książka nosiła tytuł ”Inżynieria wsteczna dla początkujących” ale zawsze podejrzewałem, że zawęża to grono czytelników.

Ludzie zajmujący się bezpieczeństwem informacji (infosec) wiedzą o inżynierii wstecznej, jednak rzadko kiedy słyszałem, by używali słowa asembler.

Podobnie, termin “inżynieria wsteczna” jest nieco tajemniczy dla ogólnego grona programistów, jednak wiedzą oni o istnieniu asemblera.

W lipcu 2018 roku, w ramach eksperymentu, zmieniłem tytuł na “Język maszynowy dla początkujących” i umieściłem link na portalu Hacker News[1], i książka została ogólnie dobrze przyjęta.

Niech więc tak będzie, książka ma teraz dwa tytuły.

Zmieniłem jednak drugi tytuł na ”Rozumienie kodu maszynowego”, ponieważ ktoś już napisał książkę o tytule ”Język maszynowy dla początkujących”. Ludzie twierdzą, że ”dla początkujących” brzmi odrobinę ironicznie jak na ~1000 stronicową książkę.

Dwie książki różnią się jedynie tytułem, nazwą pliku (UAL-XX.pdf versus RE4B-XX.pdf), URLem i kilkoma pierwszymi stronami.

O inżynierii wstecznej

Termin „inżynieria wsteczna” ma kilka popularnych definicji:

1) inżynieria wsteczna oprogramowania; analiza skompilowanych programów;

2) skanowanie modelu w 3D, żeby następnie go skopiować;

3) odzyskiwanie struktury **DBMS!**[2].

Nasza książka będzie powiązana z tą pierwszą definicją.

Pożądana wiedza

Podstawowa znajomość C **PL!**[3] . Polecane materiały: **??**.

Ćwiczenia i zadania

...wszystkie są na osobnej stronie: http://challenges.re.

Pochwały dla książki

https://beginners.re/#praise.

Uczelnie

Ta książka jest polecana przynajmniej na poniższych uczelniach: https://beginners.re/#uni.

Podziękowania

Za cierpliwe odpowiadanie na wszystkie moje pytania: SkullC0DEr.

Za wskazanie błędów i nieścisłości: Alexander Lysenko, Alexander „Solar Designer” Peslyak, Federico Ramondino, Mark Wilson, Razikhova Meiramgul Kayratovna, Anatoly Prokofiev, Kostya Begunets, Valentin “netch” Nechayev, Aleksandr Plakhov, Artem Metla, Alexander Yastrebov, Vlad Golovkin[4], Evgeny Proshin, Alexander Myasnikov, Alexey Tretiakov, Zhu Ruijin, Changmin Heo, Vitor Vidal, Stijn Crevits, Jean-Gregoire

[1] https://news.ycombinator.com/item?id=17549050
[2] **DBMS!**
[3] **PL!**
[4] goto-vlad@github

Foulon[5], Ben L., Etienne Khan, Norbert Szetei[6], Marc Remy, Michael Hansen, Derk Barten, The Renaissance[7], Hugo Chan, Emil Mursalimov, Tanner Hoke, Tan90909090@GitHub, Ole Petter Orhagen, Sourav Punoriyar, Vitor Oliveira, Alexis Ehret, Maxim Shlochiski, Greg Paton, Pierrick Lebourgeois.

Za inną pomoc: Andrew Zubinski, Arnaud Patard (rtp on #debian-arm IRC), noshadow on #gcc IRC, Aliaksandr Autayeu, Mohsen Mostafa Jokar, Peter Sovietov, Misha “tiphareth” Verbitsky.

Za przetłumaczenie tej książki na język chiński uproszczony: Antiy Labs (antiy.cn), Archer.

Za tłumaczenie na język koreański: Byungho Min.

Za tłumaczenie na język holenderski: Cedric Sambre (AKA Midas).

Za tłumaczenie na język hiszpański: Diego Boy, Luis Alberto Espinosa Calvo, Fernando Guida, Diogo Mussi, Patricio Galdames.

Za tłumaczenie na język portugalski: Thales Stevan de A. Gois, Diogo Mussi, Luiz Filipe, Primo David Santini.

Za tłumaczenie na język włoski: Federico Ramondino[8], Paolo Stivanin[9], twyK, Fabrizio Bertone, Matteo Sticco, Marco Negro[10], bluepulsar.

Za tłumaczenie na język francuski: Florent Besnard[11], Marc Remy[12], Baudouin Landais, Téo Dacquet[13], BlueSkeye@GitHub[14].

Za tłumaczenie na język niemiecki: Dennis Siekmeier[15], Julius Angres[16], Dirk Loser[17], Clemens Tamme, Philipp Schweinzer.

Za tłumaczenie na język polski: Kateryna Rozanova, Aleksander Mistewicz, Wiktoria Lewicka, Marcin Sokołowski.

Za tłumaczenie na język japoński: shmz@github[18],4ryuJP@github[19].

Za korektę: Alexander „Lstar” Chernenkiy, Vladimir Botov, Andrei Brazhuk, Mark “Logxen” Cooper, Yuan Jochen Kang, Mal Malakov, Lewis Porter, Jarle Thorsen, Hong Xie.

Vasil Kolev[20] wprowadził wiele poprawek i wskazał sporo błędów.

Dziękuję również wszystkim użytkownikom z github.com za ich komentarze i poprawki.

Użyłem wielu pakietów LaTeX. Chciałbym podziękować również ich autorom.

Darczyńcy

Tym wszystkim, którzy mnie wspierali w czasie pisania tej książki:

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

[5] https://github.com/pixjuan
[6] https://github.com/73696e65
[7] https://github.com/TheRenaissance
[8] https://github.com/pinkrab
[9] https://github.com/paolostivanin
[10] https://github.com/Internaut401
[11] https://github.com/besnardf
[12] https://github.com/mremy
[13] https://github.com/T30rix
[14] https://github.com/BlueSkeye
[15] https://github.com/DSiekmeier
[16] https://github.com/JAngres
[17] https://github.com/PolymathMonkey
[18] https://github.com/shmz
[19] https://github.com/4ryuJP
[20] https://vasil.ludost.net/

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), Nikolay Gavrilov ($300), Ernesto Bonev Reynoso ($30).

bardzo dziękuję.

mini-FAQ

Q: Czy ta książka jest prostsza niż inne?

A: Nie, poziom trudności jest mniej więcej taki sam jak innych książek na ten temat.

Q: Obawiam się zacząć czytać tę książkę, ma ponad 1000 stron. "... dla początkujących" w nazwie brzmi nieco ironicznie.

A: Wszelkiego rodzaju kody źródłowe stanowią większość tej książki. Ta książka naprawdę jest dla początkujących, wiele w niej (jeszcze) brakuje.

Q: Co trzeba wiedzieć zanim się przystąpi do czytania książki?

A: Umiejętności C/C++ są pożądane, ale nie są niezbędne.

Q: Czy powinienem uczyć się jednocześnie x86/x64/ARM i MIPS? Czy to nie za dużo?

A: Myślę, że na początek wystarczy czytać tylko o x86/x64, części o ARM i MIPS można pominąć.

Q: Czy można zakupić książki w wersji papierowej w języku rosyjskim lub angielskim?

A: Niestety nie, żaden wydawca jeszcze się nie zainteresował wydaniem rosyjskiej lub angielskiej wersji. Natomiast można ją wydrukować i zbindować w każdym ksero. https://yurichev.com/news/20200222_printed_RE4B/.

Q: Czy istnieje wersja epub/mobi?

A: Nie. W wielu miejscach książka korzysta z hacków specyficznych dla TeXa/LaTeXa, dlatego przerobienie jej na HTML (epub/mobi to jest HTML) nie jest łatwe.

Q: Po co uczyć się asemblera w dzisiejszych czasach?

A: Jeśli się nie jest developerem **OS!**[21], to prawdopodobnie nie trzeba pisać nic w asemblerze: współczesne kompilatory optymalizują kod lepiej niż człowiek [22].

Do tego, współczesne **CPU!**[23] są bardzo skomplikowanymi urządzeniami i znajomość asemblera nie pomoże poznać ich mechanizmów wewnętrznych.

Jednak zostają dwa obszary, w których dobra znajomość asemblera może być pomocna: 1) badanie malware (złośliwego oprogramowania) w celu jego analizy ; 2) lepsze zrozumienie skompilowanego kodu w trakcie debuggowania.

Wobec tego ta książka jest napisana dla tych ludzi, którzy raczej chcą rozumieć assembler, a nie w nim pisać. Stąd jest w niej bardzo dużo przykładów - wyjść kompilatora.

Q: Kliknąłem w odnośnik wewnątrz pliku PDF, jak teraz wrócić?

A: W Adobe Acrobat Reader trzeba wcisnąć Alt+LeftArrow. W Evince wcisnąć "<".

Q: Czy mogę wydrukować tę książkę? Korzystać z niej do nauczania?

A: Oczywiście, właśnie dlatego ta książka ma licencję Creative Commons (CC BY-SA 4.0).

Q: Dlaczego ta książka jest darmowa? Wykonałeś świetną robotę. To podejrzane, podobnie jak z innymi rzeczami za darmo.

A: Moim zdaniem, autorzy literatury technicznej robią to dla autoreklamy. Taka robota nie przynosi za dużo pieniędzy.

Q: Jak znaleźć pracę w zawodzie reverse engineer-a?

A: Na reddit (RE[24]), od czasu od czasu pojawiają się wątki poszukiwania pracowników. Można spróbować tam poszukać.

Q: Wersje kompilatorów z tej książki są już przestarzałe...

[21] **OS!**
[22] Bardzo ciekawy artykuł na ten temat: [Agner Fog, *The microarchitecture of Intel, AMD and VIA CPUs*, (2016)]
[23] **CPU!**
[24] reddit.com/r/ReverseEngineering/

A: Nie ma potrzeby by dokładnie wykonywać wszystkie kroki opisane w książce. Użyj kompilatorów, które masz już zainstalowane w swoim **OS!**. Poza tym, istnieje również: Compiler Explorer.

Q: Mam pytanie...

A: Napisz do mnie maila (<dennis@yurichev.com> lub <dennis.yurichev@gmail.com>).

O tłumaczeniu na język koreański

W styczniu 2015, wydawnictwo Acorn (www.acornpub.co.kr) z Korei Południowej wykonało ciężką pracę, żeby przetłumaczyć i wydać moją książkę (stanem na sierpień 2014) w języku koreańskim. Jest ona teraz dostępna na ich stronie.

Tłumaczył Byungho Min (twitter/tais9). Okładka namalował mój dobry przyjaciel, artysta, Andy Nechaevsky facebook/andydinka.

Acorn również ma prawa autorskie do tłumaczenia koreańskiego. Jakbyście chcieli mieć *prawdziwą* książkę w języku koreańskim i chcielibyście wesprzeć moją pracę, możecie ją kupić.

O tłumaczeniu na język perski (farsi)

W roku 2016 książkę przetłumaczył Mohsen Mostafa Jokar (znany w irańskiej społeczności z tłumaczenia instrukcji do Radare[25]) Książka jest dostępna na stronie wydawnictwa [26] (Pendare Pars).

Pierwsze 40 stron: https://beginners.re/farsi.pdf.

Pozycja książki w Narodowej Bibliotece Iranu: http://opac.nlai.ir/opac-prod/bibliographic/4473995.

O tłumaczeniu na język chiński

W kwietniu 2017, wydawnictwo PTPress skończyło tłumaczenie mojej książki na język chiński. Mają również prawo autorskie do tłumaczenia chińskiego.

Chińskie tłumaczenie można zamówić tutaj: http://www.epubit.com.cn/book/details/4174. Recenzje i historię tłumaczenia można znaleźć tutaj: http://www.cptoday.cn/news/detail/3155.

Głównym tłumaczem był Archer, u którego mam teraz dług wdzięczności. Był bardzo dociekliwy i znalazł w książce sporo bugów i błędów, co jest szczególnie ważne w literaturze, której dotyczy ta książka.

Będę polecał go również innym autorom!

Chłopaki z Antiy Labs również pomogli z tłumaczeniem. Tutaj słowo wstępne napisane przez nich.

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

Rozdział 1

Przykłady kodu

1.1 Metoda

Kiedy autor tej książki uczył się C, a później C++, pisał niewielkie kawałki kodu, kompilował i patrzył jak wyglądają w asemblerze. Tak było o wiele łatwiej zrozumieć co się dzieje w programie.[1]. Robił to wystarczająco dużo razy, żeby związek między kodem w C/C++ a tym co generuje kompilator wbił się w jego podświadomość bardzo głęboko. Po tym, patrząc na kod w asemblerze, od razu ogólnikowo rozumiał to co było napisane w C. Możliwe, że ta metoda pomoże komuś jeszcze.

Czasami wykorzystam stare kompilatory, żeby otrzymać bardzo krótki lub prosty kawałek kodu.

À propos, jest świetna strona, gdzie możesz zrobić to samo, używając różnych kompilatorów - bez konieczności instalowania ich u siebie: http://godbolt.org/.

Ćwiczenia

Kiedy autor tej książki uczył się asemblera, często kompilował krótkie funkcje w C i przepisywał je stopniowo na assemblera, starając się uzyskać jak najkrótszy kodu. Prawdopodobnie nie warto tego robić w praktycznych zastosowaniach, ponieważ trudno jest konkurować ze współczesnymi kompilatorami pod kątem wydajności. Jest to jednak bardzo dobry sposób na zrozumienie asemblera. Możesz wziąć dowolny fragment kodu w asemblerze z tej książki i postarać się uczynić go krótszym. Ale nie zapomnij przetestować swojego rezultatu.

Poziomy optymalizacji i debuggowanie

Kod źródłowy można kompilować różnymi kompilatorami z różnym poziomami optymalizacji. W typowym kompilatorze jest tych poziomów około trzech, gdzie poziom zerowy oznacza wyłączoną optymalizację. Optymalizować można rozmiar programu lub jego szybkość. Kompilator, który nie dokonuje optymalizacji, działa szybciej i generuje bardziej przejrzysty kod (choć i większy objętościowo). Kompilator, który dokonuje optymalizacji, działa wolniej i stara się wygenerować jak najszybszy kod (co nie zawsze znaczy, że kod będzie krótszy). Obok poziomów i kierunków optymalizacji kompilator może załączać do pliku wynikowego dodatkowe informacje dla debuggera, tworząc w ten sposób kod, który jest prostszy w debuggowaniu. Bardzo ważną cechą kodu debuggowanego jest to, że może on zawierać powiązanie między każdą linią kodu źródłowego a adresem w kodzie maszynowym. Kompilatory, dokonując optymalizacji, zwykle generują kod, gdzie całe linie kodu źródłowego mogą zostać pominięte nie będą nawet widoczne w kodzie maszynowym. Praktykujący reverse engineer z reguły ma styczność z obiema wersjami, jako że niektórzy developerzy włączają optymalizację, a niektórzy - nie.

Dlatego będziemy pracować z przykładami kodu w obu wariantach.

1.2 Niektóre podstawowe pojęcia

1.2.1 Krótkie wprowadzenie do CPU

CPU! (procesor) jest urządzeniem, które wykonuje bezpośrednio kod maszynowy programu.

[1] Szczerze mówiąc, dalej tak robi, kiedy nie rozumie jak jakiś kod działa. Ostatni przykład, z 2019 roku: `p += p+(i&1)+2;` z "SAT0W", SAT-solvera autorstwa D. Knutha.

Terminologia:

Instrukcja : prymitywny rozkaz **CPU!**. Najprostsze przykłady: przenoszenie (kopiowanie) danych między rejestrami, korzystanie z pamięci (zapis/odczyt), proste operacje arytmetyczne.

Z reguły, każdy **CPU!** ma swój zestaw instrukcji (**ISA!**[2]).

Kod maszynowy : kod wykonywany bezpośrednio przez **CPU!**. Każda instrukcja kodu maszynowego zwykle jest kodowana za pomocą kilku bajtów.

Język asemblera : kod maszynowy plus niektóre rozszerzenia (np. makra), stworzone po to, żeby ułatwić pracę programiście.

Rejestr CPU : Każdy **CPU!** ma swój zestaw rejestrów ogólnego przeznaczenia (**GPR!**[3]). ≈ 8 w x86, ≈ 16 w x86-64 i ≈ 16 w ARM. Najłatwiej traktować rejestr jako zmienną tymczasową bez określonego typu. Wyobraź sobie, że pisząc w języku wyższego poziomu masz dostępne tylko 8 zmiennych o szerokości 32 (lub 64) bitów. Te 8 zmiennych to właśnie rejestry. Wbrew pozorom można z nimi naprawdę wiele zrobić!

Dlaczego występują języki niższego i wyższego poziomu? Odpowiedź jest prosta: ludzie i procesory różnią się między sobą - dla człowieka jest o wiele łatwiej pisać w wysokopoziomowym języku programowania typu C/C++, Java czy Python, a dla procesora łatwiej jest pracować na niższym poziomie abstrakcji. Zapewne możnaby zbudować procesor, który wykonywałby kod wysokiego poziomu, ale jego budowa byłaby dużo bardziej skomplikowana niż budowa procesorów jakie obecnie znamy. I odwrotnie, pisanie w języku asemblera jest dla ludzi bardzo niewygodne, z uwagi na jego niski poziom i trudność kodowania bez popełniania całej masy drobnych błędów. Program, który potrafi konwertować język wysokiego poziomu na kod asemblera, nazywamy *kompilatorem*.

Kilka słów o różnicy między ISA!

x86 od zawsze zawierał instrukcje o różnej długości, więc kiedy nadeszła era 64-bitowej architektury, rozszerzenia x64 nie wpłynęły znacząco na **ISA!**.

ARM to procesor **RISC!**[4] zaprojektowany tak, żeby zawierał wszystkie instrukcje tej samej długości, co miało sporo zalet w przeszłosci. Na samym początku wszystkie instrukcje ARM były kodowane na czterech bajtach (obecnie „tryb ARM")[5].

Później się okazało, że nie jest to zbyt ekonomiczne, bo najczęściej używane przez procesor instrukcje[6] mogą być zakodowane z wykorzystaniem mniejszej ilości informacji. Więc dodano inną **ISA!** o nazwie "Thumb", gdzie każda instrukcja jest kodowana za pomocą tylko 2 bajtów. Jednak nie *wszystkie* instrukcje ARM mogą być zakodowane na 2 bajtach, więc zestaw instrukcji Thumb jest ograniczony. Kod skompilowany dla trybu ARM i Thumb może współdziałać w jednym programie.

Później twórcy ARM stwierdzili, że Thumb można rozszerzyć: tak pojawił się Thumb-2 (w ARMv7). Thumb-2 to wciąż dwubajtowe instrukcje, ale niektóre nowe instrukcje mają długość 4 bajtów. Szeroko rozpowszechnioną i błędną opinią jest to, że Thumb-2 to mieszanina ARM i Thumb. Tryb Thumb-2 został rozszerzony w celu wsparcia dla wszystkich możliwości procesora, by mógł konkurować z trybem ARM—co zostało w pełni osiągnięte, gdyż większość aplikacji na urządzenia iPod/iPhone/iPad są skompilowane dla zestawu instrukcji Thumb-2, (trzeba przyznać, że w dużej mierze jest to zasługa Xcode, który robi to domyślnie).

Później pojawił się 64-bitowy ARM. Jest to **ISA!** znowu z 4-bajtowymi instrukcjami, bez dodatkowego trybu Thumb. Jednak nowe 64-bitowe wymagania wpłynęły na **ISA!** tak, że obecnie mamy 3 zestawy instrukcji ARM: tryb ARM, tryb Thumb/Thumb-2 i ARM64. Te zestawy instrukcji częściowo się pokrywają, ale można powiedzieć, że są to różne zestawy a nie wariacje tego samego **ISA!**. W tej książce postaramy się zaprezentować fragmenty kodu we wszystkich trzech trybach **ISA!**. Istnieje jeszcze wiele innych **RISC! ISA!** z instrukcjami 32-bitowej długości — np. MIPS, PowerPC i Alpha AXP.

[2] **ISA!**
[3] **GPR!**
[4] **RISC!**
[5] Instrukcje o stałym rozmiarze są wygodne, bo dzięki temu można łatwo znaleźć adres następnej (lub poprzedniej) instrukcji. Dokładniejsze wyjaśnienie znajduje się w sekcji o operatorze switch() (**??**).
[6] Są nimi MOV/PUSH/CALL/Jcc

1.2.2 Systemy liczbowe

> Nowadays octal numbers seem to be used for exactly one purpose—file permissions on POSIX systems—but hexadecimal numbers are widely used to emphasize the bit pattern of a number over its numeric value.
>
> Alan A. A. Donovan, Brian W. Kernighan — The Go Programming Language

Ludzie przyzwyczaili się do systemu dziesiętnego prawdopodobnie dlatego, że każdy ma 10 palców. Natomiast, liczba 10 nie odgrywa szczególnej roli w nauce i matematyce. Binarny (dwójkowy) system liczbowy jest naturalny dla techniki cyfrowej i elektroniki: 0 oznacza brak prądu, a 1 — jego obecność. 10 w systemie binarnym to 2 w dziesiętnym; 100 w binarnym to 4 w dziesiętnym, itd.

Jeżeli w systemie liczbowym jest 10 znaków, jego *podstawa* (ang. *radix* lub *base*) to 10. System dwójkowy ma *podstawę* 2.

Ważne rzeczy, które warto sobie przypomnieć:

1) *liczba* jest liczbą, natomiast *cyfra* to umowny znak pisarski służący do zapisywania liczb; 2) sama w sobie liczba się nie zmienia przy przeliczaniu z jednego systemu na inny: zmienia się tylko sposób jej zapisu (lub reprezentacja w pamięci).

Jak skonwertować liczbę z jednego systemu na drugi?

Z notacji pozycyjnej korzysta się prawie wszędzie, to znaczy że każda cyfra posiada swoją wagę w zależności od jej usytuowania wewnątrz liczby. Jeżeli 2 znajduje sie na ostatnim miejscu od prawej, jest to 2. Jeżeli jest ona usytuowana w miejscu przed ostatnim, jest to 20.

Co oznacza zapis 1234?

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

Tak samo to wygląda w przypadku liczb binarnych, tyle że podstawą jest 2, a nie 10. Co oznacza zapis 0b101011?

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

Notację pozycyjną można przeciwstawić notacji niepozycyjnej, np. rzymskiej. [7]. Prawdopodobnie ludzkość przeszła na notację pozycyjną, ponieważ w ten sposób łatwiej jest wykonywać proste operacje (dodawanie, mnożenie, itd.) na papierze, ręcznie.

Liczby binarne również można pisemnie dodawać, odejmować itd., dokładnie tak samo, jak uczy się tego w szkole, tylko z użyciem dwóch cyfr.

Liczby w zapisie binarnym są nieporęczne, kiedy stosuje się je w kodzie źródłowym i zrzutach pamięci, dlatego w tych miejscach używa się systemu szesnastkowego (heksadecymalnego), o podstawie 16. System szesnastkowy używa szesnastu cyfr - znaków 0-9 oraz A-F. Każda cyfra zajmuje 4 bity lub 4 cyfry w systemie binarnym, więc łatwo można konwertować z reprezentacji binarnej na szesnastkową i odwrotnie, nawet w pamięci.

szesnastkowy	binarny	dziesiętny
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

[7] O ewolucji systemów liczbowych przeczytasz w [Donald E. Knuth, *The Art of Computer Programming*, Volume 2, 3rd ed., (1997), 195–213.]

F	1111	15

Jak rozpoznać w jakim systemie zapisana jest konkretna liczba?

Liczby dziesiętne, z reguły, są zapisywane jak zwykle, czyli 1234. Ale niektóre asemblery pozwalają podkreślić to przez sufiks ”d”: 1234d.

Do liczb binarnych czasami dodaje się prefiks ”0b”: 0b100110111 (W **GCC!**[8] istnieje do tego niestandardowe rozszerzenie [9]). Jest jeszcze inny sposób: sufiks ”b”, np: 100110111b. W tej książce będę się trzymał konwencji prefiksowej ”0b” dla liczb binarnych.

Liczby szesnastkowe mają prefiks ”0x” w C/C++ i niektórych innych **PL!**: 0x1234ABCD. Lub mają sufiks ”h”: 1234ABCDh — zwykle są w ten sposób reprezentowane w asemblerach lub debuggerach. W tej konwencji, jeśli liczba zaczyna się od A..F, przed nimi dopisuje się 0: 0ABCDEFh. Za czasów 8-bitowych komputerów domowych, był również sposób zapisu liczb za pomocą prefiksu $, np., $ABCD. W tej książce będę się trzymał prefiksu ”0x” dla liczb szesnastkowych.

Czy trzeba umieć konwertować liczby w głowie? Tablicę liczb szesnastkowych składających się z jednej cyfry łatwo zapamiętać, ale raczej nie warto zapamiętywać większych liczb.

Prawdopodobnie najczęściej liczby szesnastkowe są spotykane w **URL!**[10]-ach. W ten sposób są kodowane litery spoza alfabetu łacińskiego. Np.: https://en.wiktionary.org/wiki/na%C3%AFvet%C3%A9 to **URL!** strony w Wiktionary o słowie „naïveté”.

System ósemkowy

Jeszcze jeden system, z którego często korzystało się w informatyce w przeszłości to system oktalny. System oktalny ma 8 cyfr (0-7), każda opisująca 3 bity, więc łatwo przeliczać liczbę na inne systemu, w obie strony. Ten system prawie wszędzie został zastąpiony przez szesnastkowy, ale, o dziwo, w systemach *NIX nadal jest program korzystający z systemu ósemkowego: `chmod`.

Jak wiedzą użytkownicy systemów *NIX, argumentem `chmod` jest liczba składająca się z 3 cyfr. Pierwsza cyfra określa uprawnienia właściciela pliku, druga - to uprawnienia grupy (do której plik należy), trzecia dla reszty użytkowników. Każda cyfra może być przedstawiona binarnie:

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

Więc każdy bit jest powiązany z flagą: read/write/execute (prawo do odczytu/zapisu/wykonania).

Właśnie dlatego wspomniałem o `chmod`, liczba, będąca argumentem, może być reprezentowana w systemie ósemkowym. Na przykład, weźmy 644. Kiedy uruchamiasz `chmod 644 file`, ustawiasz uprawnienia read/write (odczyt zapis) dla właściciela, uprawnienia read (zapis) dla grupy, i read dla wszystkich innych użytkowników. Jeśli skonwertujemy liczbę 644 z systemu ósemkowego na binarny, to otrzymamy `110100100`, lub (w grupach po 3 bity) `110 100 100`.

Teraz widzimy, że każda ’trójka’ opisuje uprawnienia dla właściciela/grupy/reszty: pierwsza `rw-`, druga to `r--` i trzecia to `r--`.

System ósemkowy był również popularny na starych komputerach, jak PDP-8, dlatego że słowo (podstawowa porcja informacji) mogło składać się 12, 24 lub 36 bitów, a wszystkie te liczby są podzielne przez 3, więc wybór systemu ósemkowego był całkiem logiczny. Obecnie, wszystkie popularne komputery mają

[8] **GCC!**
[9] https://gcc.gnu.org/onlinedocs/gcc/Binary-constants.html
[10] **URL!**

słowa/adresy 16-, 32- lub 64-bitowe, i wszystkie te liczby są podzielne przez 4, więc system szesnastkowy jest wygodniejszy.

System ósemkowy jest wspierany przez wszystkie standardowe kompilatory C/C++. Czasami jest to źródłem nieporozumień, dlatego że liczby ósemkowe są kodowane z zerem z przodu, na przykład: 0377 to 255. Gdy pomylisz się i napiszesz ”09” zamiast 9, to kompilator zgłosi błąd. GCC może zwrócić podobny komunikat : `error: invalid digit "9" in octal constant`.

System ósemkowy jest również popularny w Javie. Gdy **IDA!**[11] wyświetla string ze znakami niedrukowalnymi, są one zakodowane w systemie ósemkowym (a nie szesnastkowym) . Dekompilator JAD zachowuje się w taki sam sposób.

Podzielność

Kiedy widzisz liczbę 120, to można szybko się zorientować, że jest ona podzielna przez 10, dlatego że ostatnią cyfrą jest 0. Podobnie, 123400 jest podzielne przez 100, bo ostatnie dwie cyfry są zerami.

Analogicznie, liczba szesnastkowa 0x1230 jest podzielna przez 0x10 (16 w systemie dziesiętnym), 0x123000 jest podzielne przez 0x1000 (4096 w systemie dziesiętnym), itd.

Liczba binarna 0b1000101000 jest podzielna przez 0b1000 (8), itd.

Tę właściwość można wykorzystać, żeby szybko sprawdzić, czy jakiś adres lub rozmiar bloku jest wyrównany do pewnej granicy (czy rozmiar bloku jest całkowitą krotnością długości słowa, np. krotnością 32 bitów). Na przykład sekcje w plikach **PE!**[12] prawie zawsze zaczynają się od adresów kończących się trzema szesnastkowymi zerami: 0x41000, 0x10001000, itd., gdyż prawie wszystkie sekcje w plikach **PE!** są wyrównane do granicy 0x1000 (4096) bajtów.

Arytmetyka wielokrotnej precyzji a baza

Arytmetyka wielokrotnej precyzji (multi-precision arithmetic) może operować na dowolnie dużych liczbach, które mogą być przechowywane w kilku bajtach. Na przykład, klucze RSA, zarówno prywatne jak i publiczne, mogą zajmować 4096 bitów a nawet więcej.

W [Donald E. Knuth, *The Art of Computer Programming*, Volume 2, 3rd ed., (1997), 265] przedstawiono ideę: kiedy przechowuje się liczbę wielokrotnej precyzji w kilku bajtach, cała liczba może być reprezentowana jako zapisana w systemie liczbowym o podstawie $2^8 = 256$, i każda cyfra reprezentuje jeden bajt. Podobnie, gdybyś liczbę wielokrotnej precyzji przechowywał w kilku 32-bitowych całkowitoliczbowych wartościach, każda cyfra byłaby reprezentowana przez 32-bitowy slot (4 bajty), i można by uważać tę liczbę za zapisaną w systemie o podstawie 2^{32}.

Wymowa liczb w systemach niedziesiętnych

Liczby zapisane w systemie o podstawie innej niż 10, zwykle wymawia się cyfra po cyfrze: “jeden-zero-zero-jeden-jeden-...“. Nie używa się słów jak “dziesięć“ czy “tysiąc“ by przez pomyłkę nie potraktowano liczby jako zapisanej w systemie dziesiętnym.

Liczby zmiennoprzecinkowe

Żeby odróżniać liczby zmiennoprzecinkowe od całkowitoliczbowych, często na końcu dodaje się “.0”, np 0.0, 123.0, itd.

1.3 Pusta funkcja

Najprostszą istniejącą funkcją jest funkcja, która nic nie robi:

Listing 1.1: Kod w C/C++

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

Skompilujmy ją!

[11] **IDA!**
[12] **PE!**

1.3.1 x86

Dla x86 i MSVC, i GCC generują ten sam kod:

Listing 1.2: Optymalizujący GCC/MSVC (wyjście w asemblerze)

```
f:
        ret
```

Mamy tu tylko jedną instrukcję: `RET`, która jest instrukcją powrotu do funkcji wywołującej.

1.3.2 ARM

Listing 1.3: Optymalizujący Keil 6/2013 (tryb ARM) (wyjście w asemblerze)

```
f       PROC
        BX       lr
        ENDP
```

Adres powrotu (**RA!**[13]) w ARM zapisywany jest nie na stosie, a w rejestrze **LR!**[14]. Instrukcja `BX LR` wykonując skok pod ten adres, zwraca sterowanie do funkcji wywołującej.

1.3.3 MIPS

Są dwie konwencje nazewnictwa rejestrów w architekturze MIPS: używająca numeru rejestru (od $0 do $31) lub nazwy umownej ($V0, $A0, itd.).

Wyjście asemblera w GCC pokazuje numery rejestrów

Listing 1.4: Optymalizujący GCC 4.4.5 (wyjście w asemblerze)

```
        j       $31
        nop
```

...a **IDA!**— nazwy umowne:

Listing 1.5: Optymalizujący GCC 4.4.5 (IDA)

```
        j       $ra
        nop
```

Pierwsza instrukcja jest instrukcją skoku (J lub JR), która zwraca sterowanie do funkcji wywołującej, skacząc pod adres w rejestrze $31 ($RA).

Jest to rejestr odpowiadający **LR!** w ARM.

Druga instrukcja to **NOP!**[15], która nic nie robi. Na razie możemy ją zignorować.

Jeszcze małe co nieco o konwecji nazewnictwa w MIPS

Nazwy rejestrów i instrukcji w MIPS tradycyjnie są zapisywane małymi literami, lecz my będziemy je zapisywać dużymi literami, dlatego że nazwy instrukcji i rejestrów innych **ISA!** w tej książce są zapisywane dużymi.

1.3.4 Puste funkcje w praktyce

Mimo, że puste funkcje są bezużyteczne, są one dość często spotykane w niskopoziomowym kodzie.

Po pierwsze, często spotykamy funkcje zapisujące szczegółowe informacje do logów, na przykład:

Listing 1.6: Kod w C/C++

```
void dbg_print (const char *fmt, ...)
{
#ifdef _DEBUG
        // open log file
```

[13] **RA!**
[14] **LR!**
[15] **NOP!**

```
        // write to log file
        // close log file
#endif
};

void some_function()
{
        ...

        dbg_print ("we did something\n");

        ...
};
```

Przy kompilacji wersji programu przeznaczonej do wdrożenia, `_DEBUG` jest niezdefiniowane, więc funkcja `dbg_print()`, mimo, że jest wywoływana, będzie pusta.

Po drugie, popularnym sposobem na ochronę oprogramowania jest kompilacja kilku wersji: pierwsza dla legalnych konsumentów, druga - demonstracyjna. Demonstracyjna wersja może nie zawierać jakiejś ważnej funkcjonalności, na przykład:

Listing 1.7: Kod w C/C++

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

Funkcja `save_file()` może być wywołana, kiedy użytkownik klika w menu `File->Save`. Wersja demo może zawierać wyłączony przycisk menu, ale nawet jeśli cracker go włączy, to zostanie wywoływana jedynie pusta funkcja, w której nie ma użytecznego kodu.

IDA oznacza takie funkcje jako `nullsub_00`, `nullsub_01`, itd.

1.4 Zwracanie wartości

Inną prostą funkcją jest taka, która zwraca stałą wartość.

Listing 1.8: Kod w C/C++

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

Skompilujmy ją.

1.4.1 x86

Poniżej efekt kompilacji z optymalizacją kompilatorami GCC i MSVC na x86:

Listing 1.9: Optymalizujący GCC/MSVC (wyjście w asemblerze)

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

Są tu tylko dwie instrukcje: pierwsza zapisuje wartość 123 do rejestru EAX, który umownie jest używany do przechowywania wartości zwracanej, a drugą jest `RET`, która zwraca sterowanie do funkcji wywołującej.

Funkcja wywołująca pobierze wynik z rejestru `EAX`.

1.4.2 ARM

W kodzie maszynowym na ARM widać kilka różnic:

Listing 1.10: Optymalizujący Keil 6/2013 (tryb ARM) (wyjście w asemblerze)

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

ARM używa rejestru `R0` do zwracania wartości z funkcji, więc 123 kopiowane jest do `R0`.

Warto zaznaczyć, że nazwa instrukcji `MOV` jest myląca, zarówno na x86 jak i na ARM.

Dane nie są *przenoszone*, tylko *kopiowane*.

1.4.3 MIPS

Wyjście asemblera GCC oznacza rejestry numerami:

Listing 1.11: Optymalizujący GCC 4.4.5 (wyjście w asemblerze)

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

...a **IDA!** używa nazw umownych:

Listing 1.12: Optymalizujący GCC 4.4.5 (IDA)

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

Rejestr $2 (nazwa umowna - $V0) używany jest do przechowywanie wartości zwracanej z funkcji. `LI` to skrót od “Load Immediate” i w architekturze MIPS jest odpowiednikiem instrukcji `MOV` z x86.

Drugą instrukcją jest instrukcja skoku (J lub JR), która zwraca sterowanie do funkcji wywołującej.

Możesz się zastanawiać dlaczego instrukcje LI i J/JR są w odwrotnej kolejności. Jest to efekt optymalizacji przetwarzania potokowego w architekturze **RISC!**, zwanej „branch delay slot”.

Przyczyną wprowadzanie takiego rozwiązania jest dziwactwo w niektórych architekturach typu RISC i nie jest to dla nas istotne - po prostu przyjmijmy, że w asemblerze MIPS instrukcja następująca po instrukcji skoku jump/branch jest wykonywana przed samym skokiem.

W rezultacie, instrukcje typu branch są zawsze zamienione z instrukcją, która jest wykonywana przed nimi.

W praktyce często występują funkcje, które jedynie zwracają 1 (*true* - *prawdę*) lub 0 (*false* - *fałsz*).

Najmniejsze UNIXowe narzędzia, */bin/true* i */bin/false* zwracają odpowiednio 0 i 1, jako kod wyjścia. (Zero, jako kod wyjścia, oznacza zwykle sukces, natomiast kod niezerowy oznacza błąd.)

1.5 Hello, world!

Przejdźmy do słynnego przykładu z książki [Brian W. Kernighan, Dennis M. Ritchie, *The C Programming Language*, 2ed, (1988)]:

Listing 1.13: Kod w C/C++

```
#include <stdio.h>

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

1.5.1 x86

MSVC

Skompilujmy kod w MSVC 2010:

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

(Opcja `/Fa` oznacza wygenerowanie listingu w asemblerze)

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 generuje listingi w składni Intela. Różnica między składnią Intela a AT&T będzie omówiona w in **??**.

Kompilator wygenerował plik `1.obj`, który następnie będzie połączony konsolidatorem (ang. linker) w `1.exe`. W naszym przypadku, ten plik składa się z dwóch segmentów: `CONST` (dane-stałe) i `_TEXT` (kod).

Łańcuch znaków `hello, world` w C/C++ ma typ `const char[]` [Bjarne Stroustrup, *The C++ Programming Language, 4th Edition*, (2013)p176, 7.3.2], jednak nie posiada nazwy. Kompilator potrzebuje nazwy, żeby z tym łańcuchem znaków pracować, dlatego nadaje mu własną nazwę - `$SG3830`.

Dlatego ten przykład można by zapisać w ten sposób:

```
#include <stdio.h>

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

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

Wróćmy do kodu w asemblerze. Jak widać, łańcuch znaków kończy się bajtem zerowym — jest to element standardu C/C++ o łańcuchach znaków. Więcej o łańcuchach znaków w C/C++: **??**.

W segmencie kodu `_TEXT` znajduje się na razie tylko jedna funkcja: `main()` i, jak prawie każda funkcja, zaczyna się od prologu a kończy się epilogiem [16].

Po prologu następuje wywołanie funkcji `printf()`:
`CALL _printf`. Przed tym wywołaniem adres łańcucha znaków (lub wskaźnik na niego) z naszym pozdrowieniem (“Hello, world!”) odkładany jest na stos, za pomocą instrukcji `PUSH`.

Po tym jak funkcja `printf()` zwraca sterowanie do funkcji `main()`, adres łańcucha znaków (lub wskaźnik na niego) wciąż jest na stosie. Nie jest on dalej potrzebny, więc wskaźnik wierzchołka stosu (rejestr `ESP`) musi zostać poprawiony.

`ADD ESP, 4` oznacza: dodaj wartość 4 do rejestru `ESP`.

[16] Więcej o prologu i epilogu przeczytasz w sekcji (**??**).

Dlaczego 4? Z racji tego, że jest to kod 32-bitowy, do przekazania adresu za pomocą stosu potrzebowaliśmy 4 bajtów. W x64 potrzebowalibyśmy 8 bajtów.
`ADD ESP, 4` jest równoważne instrukcji `POP register`, ale nie wykorzystuje dodatkowego rejestru[17]. W pierwszym przypadku jedynie bezpośrednio manipulujemy na rejestrze ESP (wskaźniku wierzchołka stosu), a w drugim przypadku zdejmujemy ze stosu jeden element i odkładamy go do rejestru *register* a rejestr ESP zmienia się automatycznie.

Niektóre kompilatory, jak Intel C++ Compiler, w tej samej sytuacji mogą wygenerować `POP ECX` zamiast `ADD` (można to zaobserwować w kodzie Oracle RDBMS, gdyż jest on kompilowany właśnie tym kompilatorem). `POP ECX` oznacza zdjęcie elementu ze stosu i umieszczenie go w rejestrze `ECX`. Osiągnęliśmy taki sam efekt jak w przypadku instrukcji `ADD` (zmiana wskaźnika stosu), ale skutkiem ubocznym jest nadpisanie `ECX`.

Możliwe, że kompilator stosuje tu `POP ECX` dlatego, że ta instrukcja jest krótsza (1 bajt w przypadku `POP` kontra 3 bajty w przypadku `ADD`).

Poniżej przykład wykorzystania `POP` zamiast `ADD` z Oracle RDBMS:

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

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

MSVC czasami zachowuje się tak samo.

Listing 1.16: Saper na systemie Windows 7 32-bit

```
.text:0102106F                push    0
.text:01021071                call    ds:time
.text:01021077                pop     ecx
```

Po wywołaniu `printf()` kod w C/C++ zawiera instrukcję `return 0` — zwróć 0 jako wynik funcji `main()`.

W kodzie wygenerowanym robi to instrukcja `XOR EAX, EAX`.

`XOR`, jak można się domyśleć, to — „alternatywa wykluczająca”[18], ale kompilatory często korzystają z niej zamiast z `MOV EAX, 0` — znów dlatego, że kod operacji (opcode) jest krótszy (2 bajty w `XOR` kontra 5 w `MOV`).

Niektóre kompilatory generują `SUB EAX, EAX`, co oznacza *odejmij wartość w* `EAX` *od wartości w* `EAX`, co w każdym przypadku da 0.

Ostatnia instrukcja `RET` zwraca sterowanie do funkcji wywołującej. Zwykle jest to kod C/C++ **CRT!**[19], który z kolei zwróci sterowanie do systemu operacyjnego.

GCC

Skompilujmy teraz ten sam kod za pomocą kompilatora GCC 4.4.1 na systemie Linux: `gcc 1.c -o 1`. Następnie za pomocą deasemblera **IDA!** podejrzymy wynik kompilacji funkcji `main()`. **IDA!**, jak i MSVC, pokazują kod w składni Intela[20].

Listing 1.17: Kod w programie **IDA!**

```
main            proc near

var_10          = dword ptr -10h

                push    ebp
                mov     ebp, esp
                and     esp, 0FFFFFFF0h
                sub     esp, 10h
```

[17] Ale za to modyfikowany jest rejestr stanu
[18] Wikipedia
[19] **CRT!**
[20] Można zmusić również GCC do generowania listingów w tym formacie, za pomocą opcji `-S -masm=intel`.

```
                mov     eax, offset aHelloWorld ; "hello, world\n"
                mov     [esp+10h+var_10], eax
                call    _printf
                mov     eax, 0
                leave
                retn
main            endp
```

Wynik jest prawie taki sam. Adres łańcucha znaków `hello, world`, leżącego w segmencie danych, najpierw zapisywany jest do `EAX`, a później odkładany na stos. Dodatkowo, w prologu funkcji widzimy `AND ESP, 0FF` ta instrukcja wyrównuje `ESP` do granicy 16 bajtów. Dzięki temu wszystkie wartości na stosie będą również wyrównane w taki sam sposób (procesor pracuje efektywniej z adresami wyrównanymi do granicy 4 lub 16 bajtów.).

`SUB ESP, 10h` alokuje na stosie 16 bajtów. Choć tutaj wystarczyłyby 4 bajty, co będzie widoczne dalej.

Dzieje się tak dlatego, że ilość przydzielanego miejsca na stosie jest również wyrównywana do granicy 16 bajtów.

Adres łańcucha znaków (lub wskaźnik na niego) jest zatem odkładany prosto na stos, bez wykorzystywania instrukcji `PUSH`. *var_10* jest jednocześnie zmienną lokalną i argumentem dla funkcji `printf()`. Więcej na ten temat dowiesz się później.

Następnie jest wywoływana funkcja `printf()`.

W odróżnieniu od MSVC, GCC przy kompilacji z wyłączoną optymalizacją generuje `MOV EAX, 0` zamiast krótszego kod operacji (opcode).

Ostatnia instrukcja `LEAVE` — jest analogiczna do pary instrukcji `MOV ESP, EBP` i `POP EBP` — jest to powrót wskaźnika stosu i rejestru `EBP` do stanu początkowego. Jest to niezbędne, ponieważ na początku funkcji modyfikowaliśmy rejestry `ESP` i `EBP`
(za pomocą `MOV EBP, ESP` / `AND ESP, …`).

GCC: składnia AT&T

Sprawdźmy jak będzie wyglądał listing w składni AT&T, która jest bardziej popularna świecie UNIXa.

Listing 1.18: Kompilujemy za pomocą GCC 4.7.3

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

Otrzymamy taki plik:

Listing 1.19: 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
```

Listing zawiera wiele makr (rozpoczynających się od kropki). Na razie nie są one dla nas istotne.

Zignorujmy je wszystkie, (za wyjątkiem *.string*, które koduje sekwencję znaków zakończonych znakiem null — tak jak łańcuchy znaków w C++). Otrzymamy wtedy: [21]:

Listing 1.20: 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
```

Główne różnice między składnią Intela a AT&T :

- Operandy są zapisywane w odwotnej kolejności

  W składni Intela:
  <instrukcja> <operand docelowy> <operand źródłowy>.

  W składni AT&T:
  <instrukcja> <operand źródłowy> <operand docelowy>.

  Istnieje łatwy sposób na zapamiętanie tej różnicy: kiedy pracujecie ze składnią Intela — możecie w głowie postawić znak równości (=) między operandami, a z AT&T — strzałkę w prawo (→) [22].

- AT&T: Przed nazwami rejestrów stawia się symbol (%), a przed liczbami ($). Zamiast nawiasów kwadratowych używa się okrągłych.

- AT&T: Do każdej instrukcji dodaje się przyrostek określający typ danych:
  - q — quad (64 bity)
  - l — long (32 bity)
  - w — word (16 bitów)
  - b — byte (8 bitów)

Wracając do wyniku kompilacji: jest on niemal identyczny do tego, który prezentuje **IDA!**. Jedna drobnostka: `0FFFFFFF0h` jest zapisywane jako `$-16`. Oba zapisy oznaczją dokładnie to samo: `16` w sytemie dziesiętnym to `0x10` w szesnastkowym. `-0x10` to `0xFFFFFFF0` (dla liczb całkowitych 32-bitowych) [23].

Zwracana wartość jest ustawiany na 0 za pomocą zwykłej instrukcji `MOV`, a nie `XOR`. `MOV` zapisuje wartość do rejestru. Nazwa tej instrukcji nie jest do końca poprawna (wartości nie są przemieszczane, tylko kopiowane). W innych architekturach instrukcja ta nosi nazwę „LOAD”, „STORE” lub podobną.

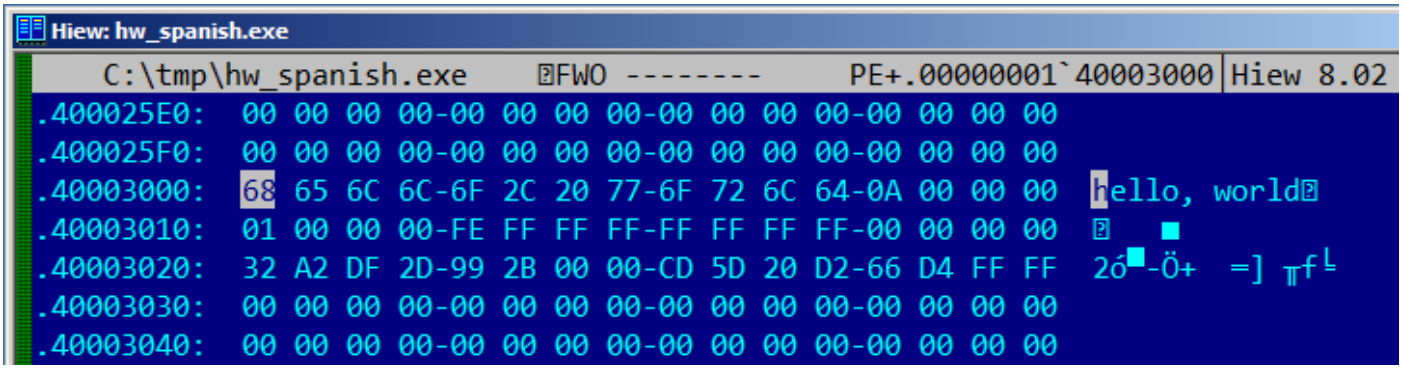
Korekcja (patching) łańcuchów znaków (Win32)

Możemy w łatwy sposób znaleźć linię “hello, world” w pliku wykonywalnym za pomocą Hiew:

[21] Eliminację zbędnych makr można uzyskać za pomocą opcji GCC: *-fno-asynchronous-unwind-tables*

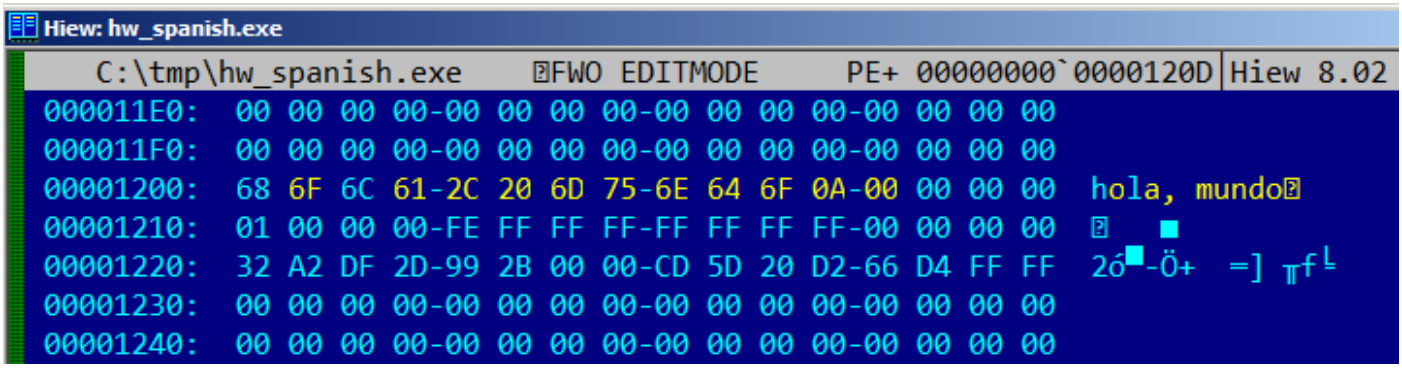
[22] W niektórych standardowych funkcjach biblioteki C (memcpy(), strcpy(), ...) również korzysta się z kolejności argumentów jak w składni Intela: najpierw wskaźnik na miejsce docelowe w pamięci, następnie wskaźnik na miejsce źródłowe.

[23] W kodowaniu U2 - https://pl.wikipedia.org/wiki/Kod_uzupe%C5%82nie%C5%84_do_dw%C3%B3ch i kolejności bajtów big endian - https://pl.wikipedia.org/wiki/Kolejno%C5%9B%C4%87_bajt%C3%B3w



Rysunek 1.1: Hiew

Możemy przetłumaczyć naszą wiadomość na język hiszpański:



Rysunek 1.2: Hiew

Tekst w języku hiszpańskim jest o 1 bajt krótszy od tekstu w języku angielskim, dlatego dodajemy na koniec bajt 0x0A (`\n`) i bajt zerowy.

Działa.

A co jeśli chcielibyśmy wstawić dłuższy tekst? Po oryginalnym tekście w języku angielskim widzimy kilka bajtów zerowych. Trudno powiedzieć czy można je nadpisać: mogą (ale nie muszą!) one być wykorzystywane gdzieś w kodzie **CRT!**. Tak czy inaczej, możemy je nadpisywać tylko jeśli naprawdę wiemy co robimy.

Korekcja łańcuchów znaków (Linux x64)

Spróbujmy edytować plik wykonywalny systemu Linux x64, korzystając z rada.re:

Listing 1.21: Sesja w rada.re

```
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
```

Co tu się dzieje: szukam łańcucha znaków „hello”, korzystając z komendy `/`, następnie ustawiam *kursor* (*seek* w terminologii rada.re) pod ten adres. Następnie chcę się upewnić, że jest to rzeczywiście poszukiwane miejsce: `px` wyświetla bajty pod tym adresem. `oo+` przełącza rada.re w tryb *odczytu/zapis*. `w` zapisuje łańuch znaków ASCII w miejscu kursora (*seek*). Warto zauważyć `\00` na końcu, jest to bajt zerowy. `q` wyłącza rada.re.

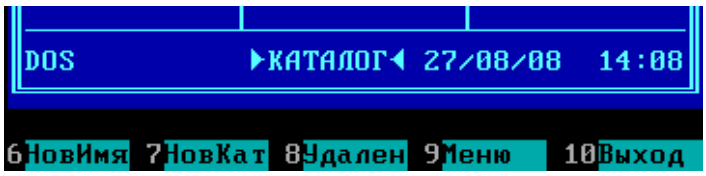
Prawdziwa historia crackowania oprogramowania

Program do przetwarzania obrazów, niezarejestrowany, dodawał do pliku znak wodny, na przykład napis „This image was processed by evaluation version of [nazwa oprogramowania]” Spróbowaliśmy najprostszego rozwiązania: znaleźliśmy ten tekst w pliku wykonywalnym i zastąpiliśmy go spacjami Znak wodny zniknął. Ogólnie rzecz biorąc, wciąż był nakładany przez program. Za pomocą funkcji Qt, znak wodny wciąż był dodawany do obrazu. Ale dodawanie spacji nie zmieniało go w żaden sposób...

Tłumaczenie oprogramowania za czasów MS-DOS

Sposób przedstawiony wyżej był powszechnie wykorzystywany w latach 80. i 90. przy tłumaczeniu oprogramowania pod MS-DOS na język rosyjski. Ta technika może być wykorzystywana przy braku wiedzy na temat kodu maszynowego i formatów plików wykonywalnych. Nowy łańcuch znaków nie powinien być dłuższy niż stary, ponieważ istnieje ryzyko nadpisania innej wartości albo fragmentu kodu wykonywalnego

Rosyjskie słowa i zdania zwykle są trochę dłuższe od angielskich odpowiedników, dlatego *przetłumaczone* oprogramowanie zawierało sporo dziwnych akronimów (skrótowców) i trudnych do zrozumienia skrótów.



Rysunek 1.3: Norton Commander 5.51 przetłumaczony na język rosyjski

Prawdopodobnie sytuacja wyglądała podobnie z tłumaczeniem na inne języki.

W przypadku łańcuch znaków w Delphi, rozmiar również musi być poprawiony, jeśli zachodzi taka potrzeba.

1.5.2 x86-64

MSVC: x86-64

Przyjrzyjmy się wynikom kompilacji 64-bitowego MSVC:

Listing 1.22: 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
```

W x86-64 wszystkie rejestry zostały rozszerzone do 64 bitów i ich nazwy zyskały prefiks `R-`. Żeby jak najrzadziej korzystać ze stosu (inaczej mówiąc, jak najmniej korzystać z pamięci cache i pamięci zewnętrznej), istnieje popularna metoda przekazywania argumentów funkcji przez rejestry (*fastcall*) **??**. Tzn. część argumentów funkcji jest przekazywana przez rejestry a część — przez stos. W Win64 pierwsze 4 argumenty funkcji są przekazywane przez rejestry `RCX`, `RDX`, `R8` i `R9`. Widać to w powyższym przykładzie: wskaźnik na argument funkcji `printf()` (łańcuch znaków) teraz jest przekazywany nie przez stos, a przez rejestr `RCX`. Wskaźniki są teraz 64-bitowe, więc są przekazywane przez przez 64-bitowe rejestry (mające prefiks `R-`). Ale dla wstecznej kompatybilności można adresować również młodsze 32 bity rejestrów poprzez prefiks `E-`. W ten o to sposób wygląda rejestr `RAX` / `EAX` / `AX` / `AL` w x86-64:

<table>
<tr><td colspan="8">Number bajtu:</td></tr>
<tr><td>7</td><td>6</td><td>5</td><td>4</td><td>3</td><td>2</td><td>1</td><td>0</td></tr>
<tr><td colspan="8">RAX^{x64}</td></tr>
<tr><td colspan="4"></td><td colspan="4">EAX</td></tr>
<tr><td colspan="6"></td><td colspan="2">AX</td></tr>
<tr><td colspan="6"></td><td>AH</td><td>AL</td></tr>
</table>

Funkcja `main()` zwraca wartość typu *int*, który w C/C++, dla większej kompatybilności, pozostał 32-bitowy. Właśnie dlatego na końcu funkcji `main()` zeruje się nie `RAX`, a `EAX`, czyli 32-bitową część rejestru. Dodatkowo, na stosie lokalnym jest zarezerwowanych 40 bajtów. Jest to tzw. „shadow space”, który będzie omawiany później: **??**.

GCC: x86-64

Przyjrzyjmy się wynikom kompilacji GCC na 64-bitowym systemie Linux:

Listing 1.23: GCC 4.4.6 x64

```
.string "hello, world\n"
main:
        sub     rsp, 8
        mov     edi, OFFSET FLAT:.LC0 ; "hello, world\n"
        xor     eax, eax  ; liczba użytych rejestrów wektorowych XMM0-XMM7
        call    printf
        xor     eax, eax
        add     rsp, 8
        ret
```

Na Linuksie, *BSD i Mac OS X w architekturze x86-64 argumenty funkcji także przekazuje się przez rejestry [Michael Matz, Jan Hubicka, Andreas Jaeger, Mark Mitchell, *System V Application Binary Interface. AMD64 Architecture Processor Supplement*, (2013)] [24].

6 pierwszych argumentów jest przekazywanych przez rejestry `RDI`, `RSI`, `RDX`, `RCX`, `R8` i `R9`, a reszta — przez stos. Zgodnie z przytoczonym standardem ABI, argumenty zmiennoprzecinkowe można przekazać również przez rejestry wektorowe `XMM0`-`XMM7`.

Wskaźnik na łańcuch znaków jest przekazywany przez `EDI` (32-bitową część rejestru). Dlaczego nie użyto 64-bitowego `RDI`?

Warto pamiętać, że w 64-bitowym trybie wszystkie instrukcje `MOV`, zapisujące cokolwiek do młodszej 32-bitowej części rejestru, zerują starsze 32 bity (jest to opisane w dokumentacji Intel: **??**). Z tego powodu instrukcja `MOV EAX, 011223344h` poprawnie zapisze tę wartość do `RAX`, a starsze bity się wyzerują.

[24] Dostęp także przez https://software.intel.com/sites/default/files/article/402129/mpx-linux64-abi.pdf

Jeśli byśmy podejrzeli w deasemblerze **IDA!** skompilowany plik (.o), to zobaczylibyśmy kody operacji (opcode) wszystkich instrukcji [25]:

Listing 1.24: GCC 4.4.6 x64

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

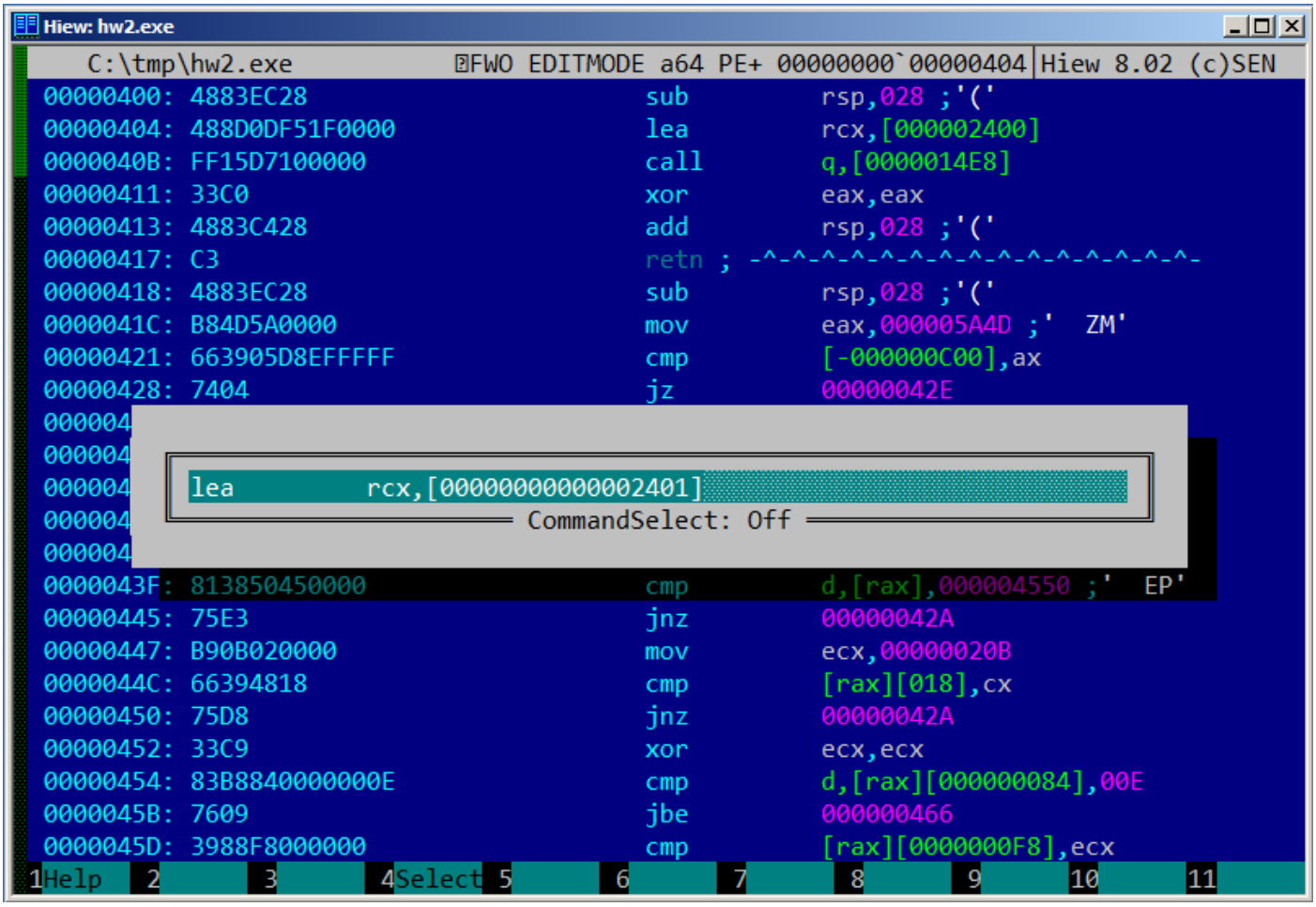
Jak widać, instrukcja `MOV` pod adresem `0x4004D4`, która zapisuje do `EDI`, zajmuje 5 bajtów. Ta sama instrukcja, zapisująca 64-bitową wartość do `RDI`, zajmuje 7 bajtów. Najwyraźniej GCC stara się zaoszczędzić trochę miejsca. GCC jest również pewne, że segment danych, w którym przechowywany jest łańcuch znaków, nigdy nie będzie zaalokowany pod adresem powyżej 4GiB.

Widać również, że rejestr `EAX` został wyzerowany przed wywołaniem funkcji `printf()`. Zgodnie ze standardem **ABI!**[26] opisanym wyżej, w *NIX dla x86-64 w `EAX` jest ustawiana liczba użytych rejestrów wektorowych do przekazania argumentów zmiennoprzecinkowych, jeśli funkcja może przyjmować zmienną liczbę argumentów. Nie został wykorzystany żaden taki rejestr, a `printf()` jest funkcją o zmiennej liczbie argumentów, więc należy ustawić `EAX` na 0.

Łatanie (patching) adresów (Win64)

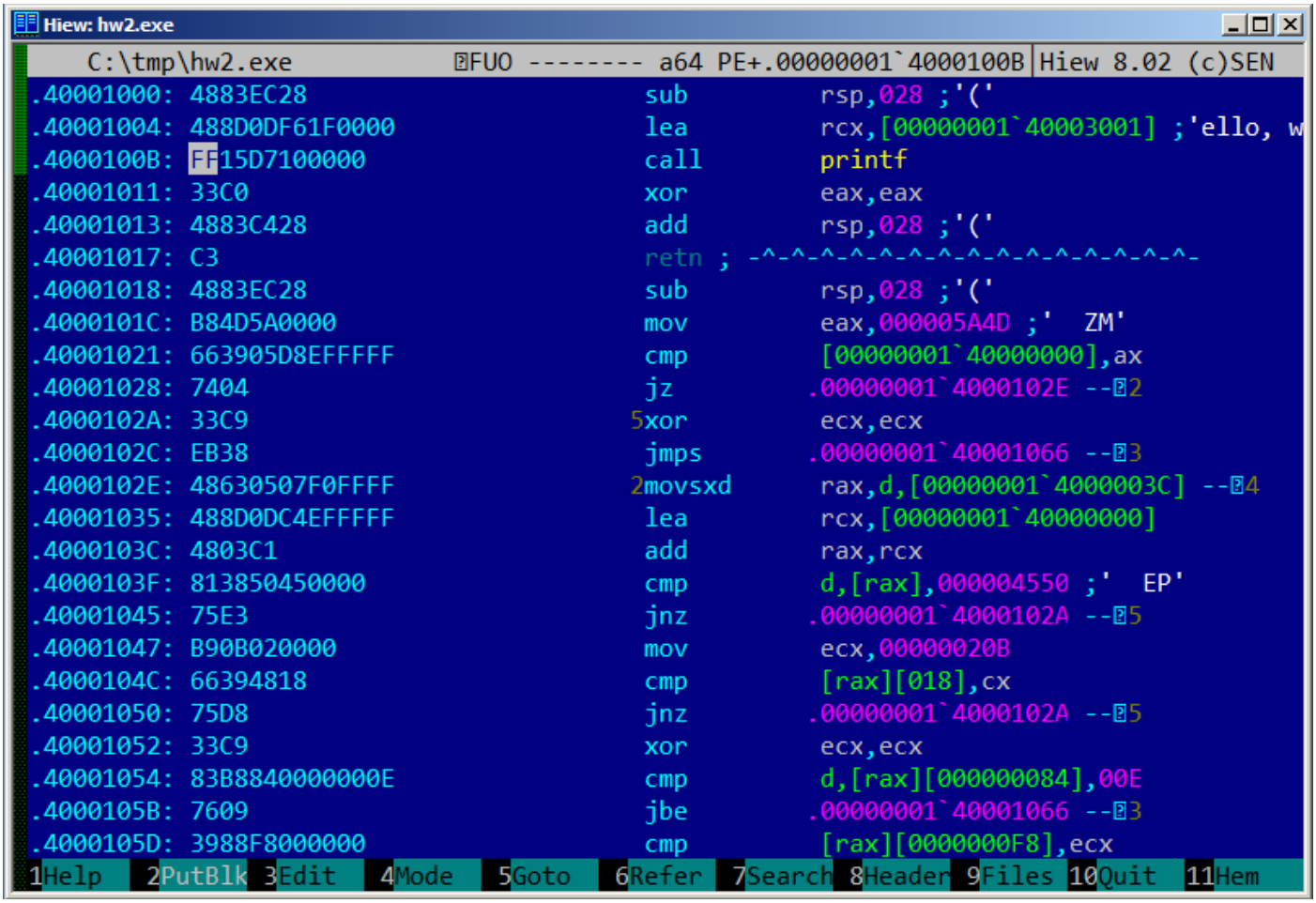
Jeśli nasz przykład skompilujemy za pomocą MSVC 2013 z opcją `/MD` (dynamiczne linkowanie, mniejszy plik wykonywalny dzięki zewnętrznym odwoływaniom do bibliotek `MSVCR*.DLL`), funkcja `main()` będzie łatwo do znalezienia, gdyż wystąpi jako pierwsza:

[25] Trzeba włączyć tę opcję w **Options** → **Disassembly** → **Number of opcode bytes**
[26] **ABI!**



Rysunek 1.4: Hiew

Możemy spróbować inkrementować adres:



Rysunek 1.5: Hiew

Hiew wyświetla teraz „ello, world” (obok instrukcji ‘`LEA RCX`...‘, która ładuje adres łańucha znaków, przekazywany jako argument do funkcji `printf()`). Kiedy uruchomimy plik wykonywalny, właśnie ten ciąg znaków zostnie wypisany na ekran.

Wyświetlanie różnych ciągów znaków z pliku wykonywalnego (Linux x64)

Plik wykonywalny po skompilowaniu za pomocą GCC 5.4.0 na systemie Linux x64, ma wiele innych łańcuchów znaków: głównie są to nazwy importowanych funkcjii oraz nazwy bibliotek.

Uruchamiamy objdump, żeby podejrzeć zawartość wszystkich sekcji skompilowanego pliku:

```
% 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.

...
```

Łatwo przekazać adres łańcucha znaków „/lib64/ld-linux-x86-64.so.2” do funkcji `printf()`:

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

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

Trudno uwierzyć, ale program wyświetli ten łańcuch znaków na ekran.

Jeśli zmienimy adres na `0x400260`, to wyświetli się napis „GNU”. Adres jest prawidłowy dla konkretnej wersji GCC, GNU toolset, etc. W waszym systemie plik wykonywalny może wyglądać trochę inaczej i wszystkie adresy także będą inne. Podobnie, usuwanie lub dodawanie kodu do kodu źródłowego może przesunąć wszystkie adresy w programie wykonywalnym do przodu lub do tyłu.

1.5.3 ARM

Do eksperymentów z ARM skorzystamy z kilku kompilatorów:

- Popularnego w systemach wbudowanych Keil Release 6/2013.
- Apple Xcode 4.6.3 IDE z kompilatorem LLVM-GCC 4.2 [27].
- GCC 4.9 (Linaro) (dla ARM64), jest dostępny w postaci pliku wykonywalnego dla win32 na http://go.yurichev.com/17325.

Wszędzie w tej książce, jeżeli nie jest zaznaczono inaczej, mówimy o 32-bitowym ARM (włączając tryb Thumb i Thumb-2). 64-bitowym ARM będzie oznaczony explicite jako ARM64.

Nieoptymalizujący Keil 6/2013 (tryb ARM)

Na początek skompilujmy nasz przykład za pomocą Keil:

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

Kompilator *armcc* generuje listing w asemblerze w składni Intela. Listing zawiera niektóre wysokopoziomowe makra, związane z ARM [28]. Nas interesują prawdziwe instrukcje, dlatego zobaczmy jak wygląda skompilowany kod w programie **IDA!**.

Listing 1.25: Nieoptymalizujący Keil 6/2013 (tryb ARM) **IDA!**

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

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

Widać, że każda instrukcja ma rozmiar 4 bajtów — zgodnie z oczekiwaniami, ponieważ kompilowaliśmy nasz kod dla trybu ARM, a nie Thumb.

Pierwsza instrukcja, `STMFD SP!, {R4,LR}` [29], działa podobnie jak instrukcja `PUSH` w x86: odkłada wartości dwóch rejestrów (`R4` i **LR!**) na stos.

W rzeczy samej, kompilator *armcc*, generując listing, wstawił tam dla uproszczenia instrukcję `PUSH {r4,lr}`.

Nie jest to do końca precyzyjne, ponieważ instrukcja `PUSH` dostępna jest w trybie Thumb. By uniknąć dezorientacji, wygenerowany kod maszynowy podglądamy w programie **IDA!**.

Instrukcja najpierw zmniejsza **SP!**[31], by wskazywał na miejsce na stosie dostępne do zapisu nowych wartości, następnie zapisuje wartości rejestrów `R4` i **LR!** pod adres w pamięci, na który wskazuje zmodyfikowany rejestr **SP!**.

Podobnie jak `PUSH` w trybie Thumb, ta instrukcja pozwala na odkładanie na stos wartości kilku rejestrów na raz, co może być bardzo wygodne.

[27] W rzeczywistości Apple Xcode 4.6.3 korzysta z GCC jako front-endu i z LLVM jako generatora kodu binarnego

[28] na przykład listing zawiera instrukcję `PUSH` / `POP`, których nie ma w trybie ARM

[29] **STMFD!**[30]

[31] **SP!**

Przy okazji, takie zachowanie nie ma swojego odpowiednika w x86.

Można zauważyć, że `STMFD` jest generalizacją instrukcji `PUSH` (czyli rozszerza jej możliwości), dlatego że może operować na różnych rejestrach, a nie tylko na **SP!**. Inaczej mówiąc, z `STMFD` można korzystać przy zapisie wartości kilku rejestrów we wskazane miejsce w pamięci.

Instrukcja `ADR R0, aHelloWorld` dodaje/odejmuje wartość w rejestrze **PC!**[32] (R0) do/od przesunięcia, w którym jest przechowywany łańcuch znaków `hello, world`. Dlaczego użyto tutaj rejestru **PC!**? Jest to tzw. „position-independent code” [33].

Taki kod można uruchomić z dowolnego miejsca w pamięci. Inaczej mówiąc jest to adresowanie względne, względem rejestru **PC!**. W kodzie operacji (opcode) instrukcji `ADR` jest zapisane przesunięcie (offset) między adresem tej instrukcji a adresem łańcucha znaków. Przesunięcie zawsze jest stałe, niezależnie od tego, w które miejsce **OS!** załadował nasz kod. Dlatego, wszystko czego potrzebujemy — to dodanie adresu bieżącej instrukcji (z **PC!**), żeby otrzymać adres bezwględny łańcucha znaków.

Instrukcja `BL __2printf`[34] wywołuje funkcję `printf()`. Działanie tej instrukcji przebiega w 2 krokach:

- zapisz adres występujący po instrukcji `BL` (`0xC`) do rejestru **LR!**,
- przekaż sterowanie do funkcji `printf()`, zapisując jej adres do rejestru **PC!**.

Kiedy funkcja `printf()` zakończy działanie, musi wiedzieć gdzie zwrócić sterowanie. Dlatego każda funkcja, kończąc pracę, zwraca sterowanie pod adres zapisany w rejestrze **LR!**.

Na tym polega główna różnica między „czystymi” procesorami **RISC!**, jak ARM, i procesorami **CISC!**[35] w rodzaju x86, gdzie adres powrotu zwykle jest odkładany na stos. Przeczytasz o tym więcej w kolejnym rozdziale (**??**).

Dodatkowo nie jest możliwe zakodowanie 32-bitowego adresu bezwzględnego (lub przesunięcia) w 32-bitowej instrukcji `BL`, ponieważ ma ona miejsce tylko dla 24 bitów. Jak zapewne pamiętasz, wszystkie instrukcje w trybie ARM mają długość 4 bajtów (32 bitów) i mogą się znajdować tylko pod adresem wyrównanym do krotności 4 bajtów. Oznacza to, że ostatnich 2 bitów (które są zawsze zerowe) można nie kodować. Ostatecznie zostaje nam 26 bitów na zakodowanie przesunięcia. Odpowiada to zakresowi $current\_PC \pm \approx 32M$.

Następna instrukcja `MOV R0, #0`[36] po prostu zapisuje 0 do rejestru `R0`. To dlatego, że nasza funkcja zwraca 0, a wartości zwracane z funkcji zapisywane są do `R0`.

Ostatnia instrukcja to `LDMFD SP!, R4,PC`[37]. Pobiera ona wartość ze stosu (lub z pamięci), zapisuje do `R4` i **PC!** oraz zwiększa wskaźnik stosu **SP!**. Działa podobnie jak instrukcja `POP`.
Notabene pierwsza instrukcja `STMFD` odłożyła na stos wartości z rejestrów `R4` i **LR!**, ale teraz zostały one przywrócone do `R4` i **PC!**, dzięki instrukcji `LDMFD`.

Jak już wiemy, rejestr **LR!** zawiera adres w pamięci, pod który funkcja zwróci sterowanie po zakończeniu swojej pracy. Pierwsza instrukcja odkłada tę wartość na stos, ponieważ ten sam rejestr zostanie wykorzystany przez naszą funkcję `main()`, gdy wywoła ona funkcję `printf()`.

Na końcu funkcji ta wartość zapisywana jest do rejestru **PC!**, w ten sposób przekazując sterowanie tam, skąd została wywołana.

Z reguły funkcja `main()` jest funkcją główną w C/C++, więc zarządzanie zostanie zwrócone do loadera w **OS!**, lub gdzieś do **CRT!**, lub w jeszcze inne, podobne, miejsce.

Wszystko to pozwala pozbyć się ręcznego wywoływania `BX LR` (skok pod adres z rejestru **LR!**) na samym końcu funcji.

`DCB` — dyrektywa asemblera, opisująca tablicę bajtów bądź ciąg znaków ASCII, podobna do dyrektywy DB w x86

[32] **PC!**
[33] Jest to szerzej omówione w kolejnym rozdziale (**??**)
[34] Branch with Link
[35] **CISC!**
[36] oznacza MOV
[37] **LDMFD!**[38] jest instrukcją odwrotną do **STMFD!**

Nieoptymalizujący Keil 6/2013 (tryb Thumb)

Skompilujmy ten sam przykład za pomocą Keil w trybie Thumb:

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

Otrzymamy (listing z programu **IDA!**):

Listing 1.26: Nieoptymalizujący Keil 6/2013 (tryb Thumb) + **IDA!**

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

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

Od razu można zauważyć 2 bajtowe (16-bitowe) kody operacji (opcode) — jest to już wcześniej wspomniany tryb Thumb.

Wyjątkiem jest instrukcja `BL`, która składa się z dwóch 16-bitowych instrukcji. W jednym 16-bitowym kodzie operacji (opcode) jest za mało miejsca na przesunięcie, po którym znajduje się funkcja `printf()`. Dlatego pierwsza 16-bitowa instrukcja ładuje starsze 10 bitów przesunięcia, a druga — młodsze 11 bitów przesunięcia.

Skoro wszystkie instrukcje w trybie Thumb są 2-bajtowe (16-bitowe), to sytuacja, w której instrukcja zaczyna się pod adresem nieparzystym, jest niemożliwa.

Wniosek z tego jest taki, że ostatniego bitu adresu można nie kodować. Dzięki temu instrukcja `BL` w trybie Thumb może zakodować adres z przedziału $current\_PC \pm \approx 2M$.

Co do pozostałych instrukcji: `PUSH` i `POP` działają jak opisane wyżej `STMFD` / `LDMFD`, tyle że rejestr **SP!** nie jest tu jawnie wskazany. `ADR` działa dokładnie tak samo jak w poprzednim przykładzie. `MOVS` zapisuje wartość 0 do rejestru `R0`, by funkcja zwróciła zero.

Optymalizujący Xcode 4.6.3 (LLVM) (tryb ARM)

Xcode 4.6.3 bez włączonego trybu optymalizacji generuje za dużo zbędnego kodu, dlatego włączymy optymalizację (flaga `-O3`), dzięki czemu liczba instrukcji będzie tak mała, jak to możliwe.

Listing 1.27: Optymalizujący Xcode 4.6.3 (LLVM) (tryb ARM)

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

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

Instrukcje `STMFD` i `LDMFD` są już nam znane.

Instrukcja `MOV` zapisuje liczbę `0x1686` do rejestru `R0` — jest to przesunięcie, wskazujące na łańcuch znaków „Hello world!”.

Rejestr `R7` (wg standardu [*iOS ABI Function Call Guide*, (2010)][39]) przechowuje wskaźnik ramki stosu (frame pointer). Będzie to omówione później.

Instrukcja `MOVT R0, #0` (MOVe Top) zapisuje 0 do starszych 16 bitów rejestru. Zwykła instrukcja `MOV` w trybie ARM może zapisywać tylko do młodszych 16 bitów rejestru, z uwagi na długość instrukcji.

[39] Dostęp także przez http://go.yurichev.com/17276

Warto pamiętać, że w trybie ARM kody operacji (opcode) instrukcji są ograniczone do 32 bitów. Nie ma to oczywiście wpływu na przenoszenie wartości między rejestrami. Z tego powodu do zapisywania do starszych bitów (od 16 do 31, włącznie) istnieje dodatkowa instrukcja `MOVT`. Tutaj jej użycie jest zbędne, gdyż `MOV R0, #0x1686` i tak by wyzerowała starszą część rejestru. Możliwe, że jest to niedociągnięcie kompilatora.

Instrukcja `ADD R0, PC, R0` dodaje **PC!** do `R0` żeby wyliczyć adres bezwzględny łańcucha znaków „Hello world!”. Jak już wiemy, jest to „position-independent code”, dlatego taka korekta jest niezbędna.

Instrukcja `BL` wywołuje `puts()` zamiast `printf()`.

LLVM zamienił wywołanie `printf()` na `puts()`. Działanie `printf()` z jednym argumentem jest prawie równoznaczna `puts()`.

Prawie, gdyż obie funkcje zadziałają identycznie, jeśli łańcuch znaków nie będzie zawierał sekwencji opisujących format, zaczynających się od znaku %. Jeśli będzie, wtedy wyniki ich pracy będą różne. [40].

Dlaczego kompilator zamienił wywoływaną funkcję? Prawdopodobnie dlatego, że funkcja `puts()` jest szybsza [41]. Najwidoczniej dlatego, że `puts()` przekazuje znaki na stdout, nie porównując ich ze znakiem %.

Dalej jest już znana instrukcja `MOV R0, #0`, ustawiająca 0 jako wartość zwracaną.

Optymalizujący Xcode 4.6.3 (LLVM) (tryb Thumb-2)

Domyślnie Xcode 4.6.3 wygeneruje trybie Thumb-2 podobny kod:

Listing 1.28: Optymalizujący Xcode 4.6.3 (LLVM) (tryb Thumb-2)

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

...

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

Instrukcje `BL` i `BLX` w trybie Thumb są kodowane jako para 16-bitowych instrukcji. W Thumb-2 te zastępcze kody operacji (opcode) rozszerzono tak, by instrukcje mogły być zakodowane jako 32-bitowe.

Można to łatwo zauważyc, gdyż w w trybie Thumb-2 wszystkie 32-bitowe instrukcje zaczynają się od `0xFx` lub `0xEx`.

Jednak na listingu w programie **IDA!** bajty kodu operacji (opcode) są zamienione miejscami. W procesorze ARM instrukcje są kodowane w następujący sposób: najpierw podaje się ostatni bajt, potem pierwszy (dla trybów Thumb i Thumb-2), lub, dla trybu ARM, najpierw czwarty bajt, następnie trzeci, drugi i pierwszy (z uwagi na różną kolejność bajtów).

Bajty są wypisywane w listingach IDA w następującej kolejności:

- dla trybów ARM i ARM64: 4-3-2-1;
- dla trybu Thumb: 2-1;
- dla pary 16-bitowych instrukcji w trybie Thumb-2: 2-1-4-3.

Widzimy, że instrukcje `MOVW`, `MOVT.W` i `BLX` rzeczywiście zaczynają się od `0xFx`.

Jedną z tych instrukcji jest `MOVW R0, #0x13D8` — zapisuje ona 16-bitową liczbę do młodszych bitów rejestru `R0`, zerując starsze.

[40] Należy również zauważyć, że `puts()` nie potrzebuje znaku nowej linii ‘\n’ na końcu łańcucha, dlatego został on pominięty.
[41] ciselant.de/projects/gcc_printf/gcc_printf.html

Inną instrukcją jest `MOVT.W R0, #0` — działa tak jak `MOVT` z poprzedniego przykładu, ale przeznaczona jest dla trybu Thumb-2

W tym przykładzie wykorzystana została instrukcja `BLX` zamiast `BL`. Różnica polega na tym, że oprócz zapisania adresu powrotu (**RA!**) do rejestru **LR!** i przekazania sterowania do funkcji `puts()`, odbywa się zmiana trybu procesora z Thumb/Thumb-2 na tryb ARM (lub odwrotnie).

Jest to niezbędne dlatego, że instrukcja, do której zostanie przekazane sterowanie jest zakodowana w trybie ARM i wygląda następująco:

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

Jest to skok do miejsca, w którym, w sekcji importów, zapisany jest adres funkcji `puts()`. Można zadać pytanie: dlaczego nie można wywołać `puts()` bezpośrednio, tam gdzie jest to potrzebne?

Nie jest to efektywne, z punktu widzenia oszczędności miejsca.

Praktycznie każdy program korzysta z zewnętrznych bibliotek łączonych dynamicznie (jak DLL w Windows, .so w *NIX czy .dylib w Mac OS X). W bibliotekach dynamicznych znajdują się często wykorzystywane funkcje biblioteczne, w tym funkcja `puts()` ze standardu C.

W wykonywalnym pliku binarnym (Windows PE .exe, ELF lub Mach-O) istnieje sekcja importów. Jest to lista symboli (funkcji lub zmiennych globalnych) importowanych z modułów zewnętrznych wraz z nazwami tych modułów. Program ładujący **OS!** iterując po symbolach zaimportowanych w module głównym, ładuje niezbędne moduły i ustawia rzeczywiste adresy każdego z symboli.

W naszym przypadku, *__imp__puts* jest 32-bitową zmienną, w której program ładujący **OS!** umieści rzeczywisty adres funkcji z biblioteki zewnętrznej.

Następnie `LDR` odczytuje 32-bitową wartość z tej zmiennej, i zapisując ją do rejestru **PC!**, przekazuje tam sterowanie.

Żeby skrócic czas tej procedury programu ładującego, trzeba sprawić aby adres każdego symbolu zapisywał się tylko raz, do specjalnie przydzielonego miejsca.

Do tego, jak się już upewniliśmy, zapisywanie 32-bitowej liczby do rejestru jest niemożliwe bez odwoływania się do pamięci.

Optymalnym rozwiązaniem jest wydzielenie osobnej funkcji, pracującej w trybie ARM, której jedynym celem jest przekazywanie sterowania dalej, do bilioteki dynamicznie łączonej. Nastepnie można wywoływać tę jednoinstrukcyjną funkcję z kodu w trybie Thumb.

À propos, w poprzednim przykładzie (skompilowanym dla trybu ARM), instrukcja `BL` przekazuje sterowanie do takiej samej thunk-funkcji, lecz procesor nie przestawia się w inny tryb (stąd brak „X” w mnemoniku instrukcji).

Jeszcze o thunk-funkcjach

Thunk-funkcje są trudne do zrozumienia przede wszystkim przez brak spójności w terminologii. Najprościej jest myśleć o nich jak o adapterach-przejściówkach z jednego typu gniazdek na drugi. Na przykład, adapter pozwalający włożyć do gniazdka amerykańskiego wtyczkę brytyjską lub na odwrót. Thunk-funkcje również są czasami nazywane *wrapper-ami*. *Wrap* w języku angielskim to *owinąć*, *zawinąć*, *opakować*. Oto jeszcze kilka definicji tych funkcji:

> “Kawałek kodu, który dostarcza adres:”, według P. Z. Ingerman, który wymyślił *thunk* w 1961 roku, jako sposób na powiązanie parametrów rzeczywistych z ich formalnymi definicjami w wywołaniach procedur, w języku Algol-60. Jeśli procedura jest wywołana z wyrażeniem w miejscu parametru formalnego, kompilator generuje *thunk*, który oblicza wartość wyrażenia i pozostawia adres tego wyniku w pewnej standardowej lokalizacji.
> …
> Microsoft i IBM zdefiniowali w ich systemach, opartych na Intelu, “środowisko 16-bitowe“ (z odrażającymi rejestrami segmentowymi i 64k limitem adresów) i “środowisko 32-bitowe“ (z płaskim adresowaniem i półrzeczywistym trybem zarządzania pamięcią). Te dwa środowiska mogą działać równocześnie na tym samym komputerze i systemie operacyjnym (dzięki

> temu, co w świecie Microsoftu znane jest jako WOW, co jest skrótowcem od Windows On Windows). MS i IBM zdecydowali, że proces przechodzenia z trybu 16-bitowego do 32-bitowego (i odwrotnie), nazwany zostanie “thunk”; istnieje nawet narzędzie na system Windows 95, ”THUNK.EXE”, nazwane ”thunk kompilatorem”.

(The Jargon File)

Jeszcze jeden przykład możemy znaleźć w bibliotece LAPACK — (“Linear Algebra PACKage”) napisanej w języku FORTRAN. Deweloperzy C/C++ również chcą korzystać z LAPACK, ale przepisywanie jej na C/C++, a następnie utrzymywanie kilku wersji byłoby szaleństwem. Istnieją wobec tego krótkie funkcje w C, które są wywoływane ze środowiska C/C++, które z kolei wywołują funkcje FORTRAN i prawie nic oprócz tego nie robią:

```
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);
}
```

Takie funkcje również są nazywane ”wrapperami”.

ARM64

GCC

Skompilujmy przykład w GCC 4.8.1 dla ARM64:

Listing 1.29: Nieoptymalizujący 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:       97fffa0         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!..
```

W ARM64 nie ma trybów Thumb i Thumb-2, jest tylko ARM, więc wszystkie instrukcje są 32-bitowe.

Dysponujemy dwukrotnie większą liczbą rejestrów: **??**. 64-bitowe rejestry mają prefiks `X-`, a ich 32-bitowe części — `W-`.

Instrukcja `STP` (*Store Pair*) odkłada na stos jednocześnie 2 rejestry: `X29` i `X30`. Oczywiście może ona zapisać tę parę gdziekolwiek w pamięci, ale w tym przypadku miejscem docelowym jest rejestr **SP!**, a więc jest ona odkładana na stos.

Rejestry w ARM64 są 64-bitowe (8 bajtowe), dlatego do przechowywania 2 rejestrów potrzeba 16 bajtów.

Wykrzyknik (“!”) po operandzie oznacza, że najpierw od **SP!** będzie odjęte 16 i dopiero po tej czynności wartości z obu rejestrów będą odłożone na stos.

Jest to tak zwany *pre-index*. Więcej o różnicy między *post-index* a *pre-index*: **??**.

Posługując się terminologią x86 — pierwsza instrukcja jest analogiczna do pary instrukcji `PUSH X29` i `PUSH X30`. `X29` w ARM64 jest wykorzystywane jako **FP!**[42], a `X30` jako **LR!**, dlatego są one odkładane na stos w prologu funkcji.

[42] **FP!**

Druga instrukcja kopiuje **SP!** do `X29` (**FP!**). Jest to niezbędne do ustawienia ramki stosu (stack frame) funkcji.

Instrukcje `ADRP` i `ADD` ustawiają adres łańcucha znaków „Hello!” w rejestrze `X0`, ponieważ pierwszy argument funkcji jest przekazywany przez ten rejestr. Jednakże w ARM nie ma instrukcji, za pomocą których można zapisać do rejestru dużą liczbę (dlatego że długość instrukcji wynosi maksymalnie 4 bajty. Więcej informacji o tym można znaleźć tutaj: **??**). Dlatego trzeba skorzystać z kilku instrukcji. Pierwsza instrukcja (`ADRP`) zapisuje do `X0` adres strony o rozmiarze 4KiB, która zawiera łańcuch znaków, a druga (`ADD`) dodaje do tego adresu resztę (przesunięcie względem początku strony pamięci). Więcej o tym: **??**.

`0x400000 + 0x648 = 0x400648`, i możemy zobaczyć, że w segmencie danych `.rodata` pod tym adresem znajduje się nasz łańcuch znaków „Hello!”.

Następnie za pomocą instrukcji `BL` jest wywoływana funkcja `puts()`. Zostało to omówione wcześniej: **??**.

Instrukcja `MOV` zapisuje 0 do `W0`. `W0` to młodsze 32 bity 64-bitowego rejestru `X0`:

<table>
<tr><td>Starsze 32 bity</td><td>Młodsze 32 bity</td></tr>
<tr><td colspan="2">X0</td></tr>
<tr><td></td><td>W0</td></tr>
</table>

Wynik funkcji jest zwracany przez `X0` i `main()` zwraca 0.

Dlaczego 32-bitowa część? W ARM64, jak i w x86-64, typ *int* ma rozmiar 32 bitów, dla kompatybilności.

Skoro funkcja zwraca 32-bitowy *int*, to trzeba wypełnić tylko młodsze 32 bity 64-bitowego rejestru `X0`.

Żeby mieć pewność, zmieńmy przykład i skompilujmy go ponownie. Teraz `main()` zwraca 64-bitową wartość:

Listing 1.30: funkcja `main()` zwracająca wartość typu `uint64_t`

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

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

Wynik jest taki sam, tylko `MOV` w tej linii wygląda teraz:

Listing 1.31: Nieoptymalizujący GCC 4.8.1 + objdump

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

Następnie za pomocą instrukcji `LDP` (*Load Pair*) przywracane są rejestry `X29` i `X30`.

Nie ma wykrzyknika po instrukcji: oznacza to, że najpierw wartości są zdejmowane ze stosu a dopiero po tej czynności **SP!** jest zwiększany o 16.

Jest to tzw. *post-index*.

W ARM64 pojawia się nowa instrukcja: `RET`. Działa tak samo jak `BX LR`, ale zawiera specjalny bit (ang. *hint bit*), który podpowiada procesorowi, że jest to wyjście z funkcji, a nie kolejna instrukcja skoku - by procesor mógł zoptymalizować jej wykonanie.

Funkcja jest bardzo prosta, GCC z włączoną optymalizacją generuje dokładnie taki sam kod.

1.5.4 MIPS

O „wskaźniku globalnym” („global pointer”)

„Wskaźnik globalny” („global pointer”) jest bardzo ważną koncepcją MIPS. Jak już wiemy, każda instrukcja w MIPS ma długość 32 bitów, dlatego niemożliwe jest zakodowanie 32-bitowego adresu w jednej instrukcji. Zamiast tego trzeba wykorzystać parę instrukcji (jak to robiło GCC dla załadowania adresu łańcucha znaków).

Można załadować dane z dowolnego adresu w przedziale $register - 32768 ... register + 32767$, za pomocą jednej instrukcji, dlatego że można w niej zakodować 16-bitowe przesunięcie (przesunięcie może być ujemne,

stąd mówimy o zakresie liczby 16-bitowej ze znakiem). Możemy więc przydzielić do tego celu jakiś rejestr i zaalokować bufor 64KiB na najczęściej wykorzystywane dane.

Rejestr ten nazywamy „wskaźnikiem globalnym” („global pointer”) i wskazuje on na środek bufora 64KiB. Bufor zwykle zawiera zmienne globalne oraz adresy funkcji importowanych jak `printf()`, ponieważ deweloperzy GCC stwierdzili, że pobranie adresu pewnych funkcji musi być tak szybkie, jak to możliwe, i powinno zająć jedną instrukcję, a nie dwie.

W plikach ELF ten 64KiB bufor znajduje się częściowo w segmencie .sbss („small **BSS!**[43]”) dla danych niezainicjalizowanych i w segmencie .sbss („small **BSS!**”) dla danych zainicjalizowanych. To oznacza, że programista może wybrać do czego potrzebuje szybszego dostępu i umieścić to w segmentach .sdata/.sbss. Niektórzy programiści starej daty mogą pamiętać model pamięci w MS-DOS **??** lub w managery pamięci typu XMS/EMS, gdzie cała pamięć była podzielona na bloki o długości 64KiB.

Ten pomysł jest wykorzystywany również w innych architekturach, np. PowerPC.

Optymalizujący GCC

Popatrzmy na poniższy przykład, pokazujący wykorzystanie „wskaźnika globalnego”.

Listing 1.32: Optymalizujący GCC 4.4.5 (wyjście w asemblerze)

```
1  $LC0:
2  ; \000 to bajt zero w systemie ósemkowym:
3          .ascii  "Hello, world!\012\000"
4  main:
5  ; prolog funkcji
6  ; ustaw GP:
7          lui     $28,%hi(__gnu_local_gp)
8          addiu   $sp,$sp,-32
9          addiu   $28,$28,%lo(__gnu_local_gp)
10 ; odłóż RA na stos lokalny:
11         sw      $31,28($sp)
12 ; załaduj adres funkcji puts() z GP do $25:
13         lw      $25,%call16(puts)($28)
14 ; załaduj adres łańcucha znaków do $4 ($a0):
15         lui     $4,%hi($LC0)
16 ; skocz do puts(), zapisując adres powrotu do rejestru powrotu:
17         jalr    $25
18         addiu   $4,$4,%lo($LC0) ; branch delay slot
19 ; przywróć RA:
20         lw      $31,28($sp)
21 ; skopiuj 0 z $zero do $v0:
22         move    $2,$0
23 ; zwróć sterowanie, skacząc pod adres w RA:
24         j       $31
25 ; epilog funkcji:
26         addiu   $sp,$sp,32 ; branch delay slot + posprzątaj stos lokalny
```

Rejestr $GP w prologu funkcji ustawiany jest na środek tego obszaru. Na stos lokalny odkładany jest rejestr **RA!**. W tym przykładzie kompilator również zamienił wywołanie `printf()` na `puts()`. Adres funkcji `puts()` jest ładowany do $25 za pomocą instrukcji `LW` („Load Word”). Następnie adres łańcucha znaków jest ładowany do $4 za pomocą pary instrukcji `LUI` („Load Upper Immediate”) i `ADDIU` („Add Immediate Unsigned Word”). `LUI` ustawia starsze 16 bitów rejestru (stąd „upper” w nazwie instrukcji) i `ADDIU` dodaje młodsze 16 bitów do adresu.

`ADDIU` znajduje się po `JALR` (pamiętaj jednak o *branch delay slot*). Rejestr $4 ($A0) jest wykorzystywany do przekazywania pierwszego argumentu funkcji [44].

`JALR` („Jump and Link Register”) skacze pod adres z rejestru $25 (znajduje się w nim adres `puts()`), jednocześnie zapisując adres instrukcji, która zostanie wywołana jako następna (`LW`) w **RA!**. Widać podobieństwo do ARM. I jeszcze jedna bardzo ważna rzecz: adres zapisywany do **RA!** nie jest adresem kolejnej (`ADDIU`) instrukcji z listingu (dlatego, że jest to *delay slot* i wykonuje się przed instrukcją skoku), a instrukcji następującej po *delay slot*. W ten sposób, podczas wykonywania `JALR`, do **RA!** jest zapisywane $PC+8$. W naszym wypadku jest to adres instrukcji `LW`, kolejnej po `ADDIU`.

[43] **BSS!**

[44] Tabela rejestrów MIPS znajduje się w dodatku **??**

`LW` („Load Word”) w linii 20 przywraca **RA!** ze stosu lokalnego (ta instrukcja jest właściwie częścią epilogu funkcji).

`MOVE` w linii 22 kopiuje wartość z $0 ($ZERO) do $2 ($V0).

MIPS ma specjalny rejestr, który zawsze zawiera stałą zero. Najwyraźniej deweloperzy MIPS stwierdzili, że 0 jest najpowszechniejszą stałą w programowaniu, więc niech rejestr $0 będzie wykorzystywany za każdym razem, kiedy będzie potrzebne 0.

Inna ciekawostka: w MIPS nie ma instrukcji kopiującej wartość z rejestru do rejestru. `MOVE DST, SRC` to w rzeczywistości `ADD DST, SRC, $ZERO` (gdzie $DST = SRC + 0$). Najwidoczniej twórcy MIPS chcieli stworzyć jak najbardziej zwięzłą tablicę kodów operacji (opcode). To wcale nie znaczy, że dodawanie jest wykonywane podczas każdej instrukcji `MOVE`. Prawdopodobnie te pseudoinstrukcje są optymalizowane w **CPU!** i **ALU!**[45] nigdy nie jest wykorzystywane.

`J` w linii 24 skacze pod adres w **RA!**, co powoduje wyjście z funkcji. `ADDIU` poniżej `J` jest wykonywane przed `J` (pamiętasz o *branch delay slot*?) i należy do epilogu funkcji.

Poniżej listing z programu **IDA!**. Każdy rejestr posiada swoją pseudonazwę:

Listing 1.33: Optymalizujący GCC 4.4.5 (**IDA!**)

```
1  .text:00000000 main:
2  .text:00000000
3  .text:00000000 var_10          = -0x10
4  .text:00000000 var_4           = -4
5  .text:00000000
6  ; prolog funkcji
7  ; ustaw GP:
8  .text:00000000                 lui     $gp, (__gnu_local_gp >> 16)
9  .text:00000004                 addiu   $sp, -0x20
10 .text:00000008                 la      $gp, (__gnu_local_gp & 0xFFFF)
11 ; odłóż RA na stos lokalny:
12 .text:0000000C                 sw      $ra, 0x20+var_4($sp)
13 ; odłóż GP na stos lokalny:
14 ; z jakiegoś powodu tej instrukcji nie było na listingu z GCC:
15 .text:00000010                 sw      $gp, 0x20+var_10($sp)
16 ; załaduj adres funkcji puts() z GP do $t9:
17 .text:00000014                 lw      $t9, (puts & 0xFFFF)($gp)
18 ; wylicz adres łańcucha znaków i zapisz w $a0:
19 .text:00000018                 lui     $a0, ($LC0 >> 16)  # "Hello, world!"
20 ; skocz do puts(), zapisując adres powrotu do rejestru powrotu
21 .text:0000001C                 jalr    $t9
22 .text:00000020                 la      $a0, ($LC0 & 0xFFFF)  # "Hello, world!"
23 ; przywróć RA:
24 .text:00000024                 lw      $ra, 0x20+var_4($sp)
25 ; skopiuj 0 z $zero do $v0:
26 .text:00000028                 move    $v0, $zero
27 ; zwróć sterowanie, skacząc pod adres z RA:
28 .text:0000002C                 jr      $ra
29 ; epilog funkcji:
30 .text:00000030                 addiu   $sp, 0x20
```

Instrukcja w linii 15 odkłada GP na stos lokalny. Co ciekawe, brakuje jej na listingu z GCC — możliwe, że jest to spowodowane błędem w samym GCC[46]. Wartość GP musi zostać odłożona, ponieważ każda funkcja może używać swojego własnego 64KiB bufora. Rejestr zawierający adres `puts()` nazwany został $T9, dlatego że rejestry z prefiksem T określane są jako „tymczasowe” i ich zawartości nie musi być zachowywana - nie jest więc odkładana na stos.

Nieoptymalizujący GCC

Nieoptymalizujący GCC generuje nieco rozwlekły kod.

Listing 1.34: Nieoptymalizujący GCC 4.4.5 (wyjście w asemblerze)

```
1  $LC0:
```

[45] **ALU!**

[46] Najwidoczniej funkcje generujące listingi nie są krytyczne dla użytkowników GCC, dlatego pewne kosmetyczne błędy wciąż mogą być niepoprawione.

```
        .ascii  "Hello, world!\012\000"
main:
; prolog funkcji
; odłóż RA ($31) i FP na stos:
        addiu   $sp,$sp,-32
        sw      $31,28($sp)
        sw      $fp,24($sp)
; ustaw FP (stack frame pointer):
        move    $fp,$sp
; ustaw GP:
        lui     $28,%hi(__gnu_local_gp)
        addiu   $28,$28,%lo(__gnu_local_gp)
; załaduj adres łańcucha znaków
        lui     $2,%hi($LC0)
        addiu   $4,$2,%lo($LC0)
; załaduj adres funkcji puts() z GP:
        lw      $2,%call16(puts)($28)
        nop
; wywołaj puts():
        move    $25,$2
        jalr    $25
        nop  ; branch delay slot

; przywróć GP ze stosu lokalnego:
        lw      $28,16($fp)
; ustaw rejestr $2 ($V0) na zero:
        move    $2,$0
; epilog funkcji.
; przywróć SP:
        move    $sp,$fp
; przywróć RA:
        lw      $31,28($sp)
; przywróć FP:
        lw      $fp,24($sp)
        addiu   $sp,$sp,32
; skok pod adres z RA:
        j       $31
        nop  ; branch delay slot
```

FP jest wykorzystywany jako wskaźnik ramki stosu (stack frame pointer). Widać również 3 instrukcje **NOP!**. Drugi i trzeci **NOP!** występują po instrukcjach skoku. Prawdopodobnie kompilator GCC zawsze dodaje **NOP!**-y (przez *branch delay slot*) po instrukcjach skoku a następnie, jeśli optymalizacja jest włączona, może je wyeliminować. W tym przypadku instrukcje pozostały na swoim miejscu.

Poniżej ten sam plik wykonywalny w programie **IDA!**:

Listing 1.35: Nieoptymalizujący 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
; prolog funkcji
; odłóż RA i FP na stosie:
.text:00000000                 addiu   $sp, -0x20
.text:00000004                 sw      $ra, 0x20+var_4($sp)
.text:00000008                 sw      $fp, 0x20+var_8($sp)
; §ustaw FP (stack frame pointer)§:
.text:0000000C                 move    $fp, $sp
; ustaw GP:
.text:00000010                 la      $gp, __gnu_local_gp
.text:00000018                 sw      $gp, 0x20+var_10($sp)
; załaduj adres łańcucha znaków:
.text:0000001C                 lui     $v0, (aHelloWorld >> 16)  # "Hello, world!"
.text:00000020                 addiu   $a0, $v0, (aHelloWorld & 0xFFFF)  # "Hello, world!"
; załaduj adres funkcji puts() z GP:
.text:00000024                 lw      $v0, (puts & 0xFFFF)($gp)
.text:00000028                 or      $at, $zero ; NOP
; wywołaj puts():
```

```
24 .text:0000002C                 move    $t9, $v0
25 .text:00000030                 jalr    $t9
26 .text:00000034                 or      $at, $zero ; NOP
27 ; przywróć GP ze stosu lokalnego:
28 .text:00000038                 lw      $gp, 0x20+var_10($fp)
29 ; ustaw rejestr $2 ($V0) na zero:
30 .text:0000003C                 move    $v0, $zero
31 ; epilog funkcji.
32 ; przywróć SP:
33 .text:00000040                 move    $sp, $fp
34 ; przywróć RA:
35 .text:00000044                 lw      $ra, 0x20+var_4($sp)
36 ; przywróć FP:
37 .text:00000048                 lw      $fp, 0x20+var_8($sp)
38 .text:0000004C                 addiu   $sp, 0x20
39 ; skocz pod adres z RA:
40 .text:00000050                 jr      $ra
41 .text:00000054                 or      $at, $zero ; NOP
```

Co ciekawe, **IDA!** rozpoznała parę `LUI` / `ADDIU` i zebrała ją w jedną pseudoinstrukcję `LA` („Load Address”) w linii 15. Można zauważyć, że jej długość to 8 bajtów! Nazywamy to pseudoinstrukcją (lub *makrem*), ponieważ nie jest to prawdziwa instrukcja MIPS, tylko wygodna nazwa dla pary dwóch powiązanych instrukcji.

Widać również, że **IDA!** nie rozpoznała instrukcji **NOP!** w liniach 22, 26 i 41.

W rzeczywistości **NOP!** jest realizowany przez `OR $AT, $ZERO`. Jest to instrukcja przeprowadzająca operację *LUB* na rejestrze $AT i $ZERO (rejestr zawsze przechowujący stałą 0), co jest pustą instrukcją. MIPS, jak i inne **ISA!**, nie posiada oddzielnej instrukcji **NOP!**.

Rola ramki stosu (stack frame) w tym przykładzie

Adres łańcucha znaków jest przekazywany przez rejestr. Po co w takim razie ustawiać stos lokalny? Wartości rejestrów **RA!** i GP muszą być gdzieś zapisane (ponieważ wywołujemy funkcję - `printf()`) i w tym celu korzysta się ze stosu lokalnego.

Gdyby to była funkcja liść (ang. leaf function), to można by pozbyć się prologu i epilogu funkcji, zobacz przykład: **??**.

Optymalizujący GCC: sesja w debuggerze GDB

Listing 1.36: Przykład sesji w GDB

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

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

Breakpoint 1, 0x00400654 in main ()
(gdb) set step-mode on
(gdb) disas
Dump of assembler code for function main:
0x00400640 <main+0>:    lui     gp,0x42
0x00400644 <main+4>:    addiu   sp,sp,-32
0x00400648 <main+8>:    addiu   gp,gp,-30624
0x0040064c <main+12>:   sw      ra,28(sp)
0x00400650 <main+16>:   sw      gp,16(sp)
0x00400654 <main+20>:   lw      t9,-32716(gp)
0x00400658 <main+24>:   lui     a0,0x40
0x0040065c <main+28>:   jalr    t9
0x00400660 <main+32>:   addiu   a0,a0,2080
0x00400664 <main+36>:   lw      ra,28(sp)
0x00400668 <main+40>:   move    v0,zero
0x0040066c <main+44>:   jr      ra
```

```
0x00400670 <main+48>:   addiu   sp,sp,32
End of assembler dump.
(gdb) s
0x00400658 in main ()
(gdb) s
0x0040065c in main ()
(gdb) s
0x2ab2de60 in printf () from /lib/libc.so.6
(gdb) x/s $a0
0x400820:        "hello, world"
(gdb)
```

1.5.5 Wnioski

Główną różnicą między kodem w x86/ARM a x64/ARM64 jest to, że wskaźnik na łańcuch znaków stał się 64-bitowy. W rzeczy samej, współczesne **CPU!** stały się 64-bitowe przez niższy koszt pamięci oraz większe zapotrzebowania na nią przez współczesne aplikacje. Komputery mogą mieć więcej pamięci, niż można zaadresować za pomocą 32 bitów. W związku z tym, wszystkie wskaźniki są teraz 64-bitowe.

1.5.6 Ćwiczenia

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

1.6 Prolog i epilog funkcji

Prolog funkcji to sekwencja instrukcji rozpoczynająca funkcję. Zwykle przypomina poniższy fragment kodu:

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

Co te instrukcje robią: zapisują wartość rejestru `EBP` register na stos, ustawiają wartość w rejestrze `EBP` na wartość z `ESP` a następnie alokuję miejsce na stosie na zmienne lokalne

Wartość w `EBP` pozostaje niezmieniona przez całą funkcję i używana jest przy dostępie do zmiennych lokalnych i argumentów. Do tego samego celu można by użyć `ESP`, ale byłoby to niewygodne, gdyż wartość `ESP` zmienia się w czasie wykonywania funkcji.

Epilog funkcji zwalnia zaalokowane miejsce na stosie, przywraca wartość rejestru `EBP` do jego pierwotnego stanu i zwraca sterowanie do funkcji wywołującej:

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

Prolog i epilog zwykle jest wykrywany przez deasemblery i używany do wyodrębnienia pojedynczych funkcji.

1.6.1 Rekurencja

Prolog i epilog funkcji mają negatywny wpływ na wywołania rekurencyjne.

Więcej o rekurencji: **??**.

1.7 Pusta funkcja raz jeszcze

Wróćmy do przykładu z pustą funkcją **??**. Uzbrojeni w wiedzę o prologu i epilogu funkcji, spójrzmy na wynik kompilacji GCC bez optymalizacji:

Listing 1.37: Nieoptymalizujący GCC 8.2 x64 (wyjście w asemblerze)

```
f:
        push    rbp
        mov     rbp, rsp
        nop
        pop     rbp
        ret
```

Poza instrukcją `RET` jest tam jedynie prolog i epilog, które nie zostały zoptymalizowane.

`NOP` wygląda jak kolejna osobliwość kompilatora. Jedyną instrukcją, która realizuje działanie programu, jest tutaj `RET`. Wszystkie pozostałe można by usunąć (zoptymalizować).

1.8 Zwracanie wartości raz jeszcze

Znając pojęcie prologu i epilogu, skompilujmy raz jeszcze przykład z wartością zwracaną (**??**, **??**) za pomocą GCC z wyłączoną optymalizacją:

Listing 1.38: Nieoptymalizujący GCC 8.2 x64 (wyjście w asemblerze)

```
f:
        push    rbp
        mov     rbp, rsp
        mov     eax, 123
        pop     rbp
        ret
```

Instrukcje realizujące działanie programu to `MOV` i `RET` – pozostałe to prolog i epilog.

1.9 Stos

Stos w informatyce jest jedną z najbardziej fundamentalnych struktur danych [47]. **AKA!**[48] **LIFO!**[49].

Technicznie rzecz biorąc, jest to blok pamięci w pamięci procesu + rejestr `ESP` w x86, `RSP` w x64, lub **SP!** w ARM, który przechowuje wskaźnik na miejsce w granicach tego bloku.

Najczęściej używanymi instrukcjami do operowania na stosie są `PUSH` i `POP` (w x86 i trybie Thumb w ARM). `PUSH` zmniejsza `ESP` / `RSP` /**SP!** o 4 w trybie 32-bitowym (lub o 8 w 64-bitowym), następnie zapisuje zawartość swojego operandu pod adres, na który wskazuje `ESP` / `RSP` /**SP!**, .

`POP` jest odwrotną operacją: najpierw pobiera dane z miejsca, na które wskazuje wskaźnik stos i umieszcza ją w miejscu wskazywanym przez operand docelowy (często jest to rejestr), a następnie zwiększa wskaźnik stosu o 4 (lub 8).

Po zaalokowaniu stosu wskaźnik stosu pokazuje na koniec tego obszaru pamięci. `PUSH` zmniejsza wskaźnik stosu, a `POP` — zwiększa. Koniec stosu znajduje się na początku zaalokowanego bloku pamięci. Może zabrzmieć to dziwnie, ale tak to działa.

ARM wspiera zarówno stosy rosnący w dół, jak i w górę.

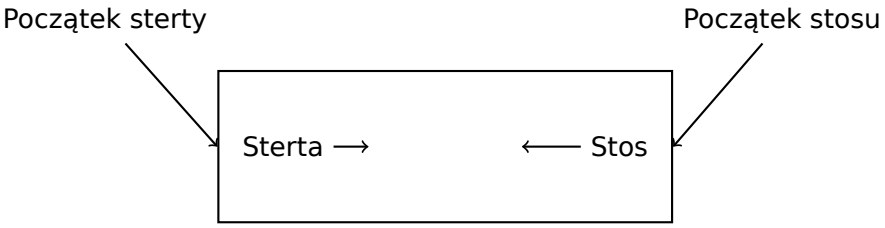
Na przykład instrukcje **STMFD!**/**LDMFD!**, **STMED!**[50]/**LDMED!**[51] są przeznaczone dla stosu malejącego (rosnącego w dół od adresów wysokich do adresów niskich).
Natomiast instrukcje **STMFA!**[52]/**LDMFA!**[53], **STMEA!**[54]/**LDMEA!**[55] są przeznaczone dla stosu rosnącego (rosnącego w górę od niskich adresów do adresów wysokich).

[47] wikipedia.org/wiki/Call_stack
[48] **AKA!**
[49] **LIFO!**
[50] **STMED!**
[51] **LDMED!**
[52] **STMFA!**
[53] **LDMFA!**
[54] **STMEA!**
[55] **LDMEA!**

1.9.1 Dlaczego stos rośnie w dół?

Intuicyjnie moglibyśmy pomyśleć, że jak każda inna struktura danych, stos mógłby rosnąć w górę, w kierunku adresów wysokich.

Prawdopodobnie stos rośnie w dół ze względów historycznych. Kiedy komputery były duże i zajmowały cały pokój, można było bardzo łatwo podzielić pamięć na dwa obszary: na stertę i na stos. Nie było wiadomo z góry jak duża może być sterta lub stos, dlatego takie rozwiązanie było najprostsze.



W [D. M. Ritchie and K. Thompson, *The UNIX Time Sharing System*, (1974)][56]można przeczytać:

> Dane użytkownika (część obrazu procesu) podzielone są na trzy logiczne segmenty. Segment kodu programu zaczyna się od adresu 0 w wirtualnej przestrzeni adresowej. W trakcie wykonania jest on zabezpieczony przed zapisem i tylko jedna jego kopia jest współdzielona przez wszystkie procesy, wykonujące ten sam program. Po pierwszej granicy 8KB ponad segmentem kodu programy, w wirtualnej przestrzeni adresowej, rozpoczyna się prywatny, zapisywalny segment danych, którego rozmiar może być zwiększany przez wywołanie systemowe. Od najwyższego adresu w wirtualnej przestrzeni adresowej zaczyna się segment stosu, który automatycznie rośnie w dół stosownie do zmian wskaźnika stosu.

To trochę przypomina sytuację, gdy uczeń prowadzi notatki z dwóch wykładów w jednym zeszycie. Pierwsze notatki zaczynają się konwencjonalnie, od początku zeszytu, ale drugie zapisywane są na końcu, po obróceniu zeszytu. Gdy zabraknie miejsca, notatki spotkają się gdzieś w środku.

1.9.2 Do czego wykorzystywany jest stos?

Zapisywanie adresu powrotu

x86

Przed wywołaniem funkcji za pomocą instrukcji `CALL`, na stos odkładany jest adres kolejnej instrukcji (tej bezpośrednio za `CALL`). Następnie następuje skok bezwarunkowy pod adres z operandu instrukcji `CALL`.

Instrukcja `CALL` jest równoważna parze instrukcji `PUSH adres_docelowy / JMP`.

`RET` zdejmuje adres ze stosu i przekazuje tam sterowanie — jest to równoważne parze instrukcji `POP tmp / JMP`

Bardzo łatwo przepełnić stos, poprzez nieskończoną rekurencję:

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

MSVC 2008 wyświetli ostrzeżenie:

```
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
```

...ale wygeneruje plik wykonywalny:

[56]Dostęp także przez http://go.yurichev.com/17270

```
?f@@YAXXZ PROC                          ; f
; Line 2
        push    ebp
        mov     ebp, esp
; Line 3
        call    ?f@@YAXXZ               ; f
; Line 4
        pop     ebp
        ret     0
?f@@YAXXZ ENDP                          ; f
```

...jeśli włączymy optymalizację (opcja `/0x`), to zoptymalizowany kod nie będzie powodował przepełnienia stosu i działał *poprawnie*[57]

```
?f@@YAXXZ PROC                          ; f
; Line 2
$LL3@f:
; Line 3
        jmp     SHORT $LL3@f
?f@@YAXXZ ENDP                          ; f
```

GCC 4.4.1 wygeneruje taki sam kod w obu przypadkach i nie wyświetli żadnego ostrzeżenia.

ARM

Programy na ARM również korzystają ze stosu do zapisywania adresu powrotu, ale trochę w inny sposób. Jak już było wspomniane w rozdziale „Hello, world!” (**??**), adres powrotu (**RA!**) jest zapisywany do rejestru **LR!** (rejestr powrotu). Jeśli zajdzie potrzeba wywołania kolejnej funkcji i ponownego użycia **LR!**, to jego zawartość będzie musiała być gdzieś zapisana.

Zwykle odbywa się to w prologu funkcji, często widzimy tam instrukcję jak `PUSH {R4-R7,LR}`, a w epilogu `POP {R4-R7,PC}` — rejestry, z których będzie korzystała bieżąca funkcja, w tym rejestr **LR!**, odkładane są na stos.

Jeśli jakaś funkcja nie wywołuje żadnych innych funkcji w trakcie swojej pracy, w terminologii **RISC!** nazywana jest *funkcją-liściem*[58]. Z tego powodu funkcja-liść nie odkłada rejestru **LR!** na stos, ponieważ go nie zmienia. Jeśli funkcja jest niewielkich rozmiarów i korzysta z małej liczby rejestrów, to może w ogóle nie korzystać ze stosu. Stąd w ARM możliwe jest wywoływanie małych funkcji-liści bez używania stosu. Jest to szybsze niż w starych x86, gdyż nie korzysta się z pamięci zewnętrznej RAM do korzystania ze stosu. [59]. Ten mechanizm przydaje się również, gdy pamięć pod stos nie została jeszcze zaalokowana albo jest niedostępna.

kilka przykładów takich funkcji: **??**, **??**, **??**, **??**, **??**, **??**, **??**, **??**.

Przekazywanie argumentów funkcji

Najpopularniejszy sposób na przekazywanie parametrów funkcji w x86 to „cdecl”:

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

Wywoływana funkcja pobiera swoje argumentu za pomocą wskaźnik stosu.

Stos, przed wykonaniem pierwszej instrukcji z `f()`, wygląda następująco:

[57] o ironio!

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

[59] Kiedyś, na PDP-11 i VAX, instrukcja CALL (wywołanie innych funkcji) była kosztowna; procesor spędzał na `CALL` nawet do 50% czasu wykonania programu. Z tego powodu posiadanie dużej liczby małych funkcji uchodziło za antywzorzec [Eric S. Raymond, *The Art of UNIX Programming*, (2003)Chapter 4, Part II].

ESP	adres powrotu
ESP+4	argument#1, oznaczony w programie **IDA!** jako `arg_0`
ESP+8	argument#2, oznaczony w programie **IDA!** jako `arg_4`
ESP+0xC	argument#3, oznaczony w programie **IDA!** jako `arg_8`
...	...

Opis innych konwencji wywoływania funkcji znajduje się tutaj: (**??**).

À propos, funkcja wywoływana nie posiada informacji o liczbie przekazywanych do niej argumentów. Funkcje w C o zmiennej liczbie argumentów (jak `printf()`) ustalają ich liczbę za pomocą specyfikatorów formatu (rozpoczynających się od znaku %).

Jeśli napiszemy:

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

`printf()` wypisze 1234, a następnie jeszcze dwie losowe[60] liczby, który przypadkowo znalazły się na stosie obok.

Dlatego nie ma znaczenia jak zapiszemy funkcję `main()`:
jak `main()`, `main(int argc, char *argv[])`
lub `main(int argc, char *argv[], char *envp[])`.

W rzeczywistości kod z **CRT!** wywołuje `main()` mniej więcej tak:

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

Jeśli zadeklarujesz `main()` bez argumentów, one i tak będą na stosie, lecz nie zostaną wykorzystane. Jeśli zadeklarujesz `main()` jako `main(int argc, char *argv[])`, to będziesz mógł skorzystać z pierwszych dwóch argumentów, a trzeci będzie dla funkcji „niewidoczny”. Co więcej, można nawet zadeklarować `main(int argc)` i to również zadziała.

Inny, podobny, przykład: **??**.

Alternatywne sposoby na przekazywanie argumentów

Warto zauważyć, że nic nie zmusza programisty do przekazywanie argumentów przez stos. Nie ma takiego wymagania, można to robić to zupełnie inaczej, nie korzystając ze stosu.

Dość popularnym sposobem wśród początkujących jest przekazywanie argumentów przez zmienne globalne, na przykład:

Listing 1.39: Kod w asemblerze

```
        ...

        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
```

[60] Tak na prawdę nie są one losowe, patrz: **??**

Ta metoda posiada oczywistą wadę: funkcja *do_something()* nie może wywołać sama siebie przez rekurencję (lub za pomocą innej funkcji), gdyż musiałaby nadpisać własne argumenty. To samo dotyczy zmiennych lokalnych, gdyby przechowywać je w zmiennych globalnych, to funkcja nie będzie mogła wywołać sama siebie. Co więcej, nie jest to bezpieczne w środowisku wielowątkowym[61]. Przechowywania danych na stosie wszystko upraszcza — stos może przechować tyle argumentów funkcji/zmiennych, na ile pozwoli jego rozmiar.

W [Donald E. Knuth, *The Art of Computer Programming*, Volume 1, 3rd ed., (1997), 189] można przeczytać o jeszcze dziwniejszych metodach przekazywania argumentów, szczególnie wygodnych na IBM System/360.

W MS-DOS istniała metoda przekazywania argumentów przez rejestry, na przykład ten fragment kodu na wiekowym 16-bitowym MS-DOS wypisze „Hello, world!”:

```
mov  dx, msg      ; adres wiadomości
mov  ah, 9        ; 9 oznacza funkcję "wypisz łańcuch znaków"
int  21h          ; wywołanie systemowe ("syscall") DOS

mov  ah, 4ch      ; funkcja "zakończ program"
int  21h          ; wywołanie systemowe ("syscall") DOS

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

Jest to całkiem podobne do metody **??**. Przypomina to również sposoby korzystania z wywołań systemowych (ang. syscall) na systemach Linuks (**??**) i Windows.

Jeżeli funkcja w MS-DOS zwraca wartość typu boolean (jeden bit, zwykle oznaczający wystąpienie błędu), to często wykorzystywana jest flaga `CF`.

Na przykład:

```
mov ah, 3ch       ; "3c" oznacza "stwórz plik
lea dx, filename
mov cl, 1
int 21h
jc  error
mov file_handle, ax
...
error:
...
```

W razie wystąpienia błędu flaga `CF` zostaje ustawiona. W przeciwnym razie uchwyt (ang. *file handle*) stworzonego pliku zwracany jest przez rejestr `AX`.

Ta metoda wciąż jest wykorzystywana przez programistów asemblera. W kodach źródłowych Windows Research Kernel (który jest bardzo podobny do Windows 2003) możemy znaleźć coś takiego (plik *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
```

[61] W poprawnej implementacji każdy wątek miałby własny stos lokalny, ze swoimi argumentami/zmiennymi.

```
        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
```

Przechowywanie zmiennych lokalnych

Funkcja może zaalokować miejsce na stosie dla własnych zmiennych lokalnych przez zmniejszenie wskaźnika stosu, w kierunku końca stosu (pamiętaj, że stos rośnie w dół, w kierunku niskich adresów!).

Jest to bardzo szybkie, niezależnie od liczby zmiennych lokalnych. Wiedz, że nie ma przymusu trzymania zmiennych lokalnych na stosie. Możesz je trzymać gdziekolwiek, ale tradycyjnie wykorzystuje się do tego stos.

x86: Funkcja alloca()

Ciekawym przypadkiem jest funkcja `alloca()` [62]. Działa ona jak `malloc()`, ale przydziela pamięć bezpośrednio na stosie. Nie ma potrzeby zwalniania tak zaalokowanego obszaru pamięci za pomocą `free()`, gdyż epilog funkcji (**??**) przywróci `ESP` do stanu początkowego i zaalokowana pamięć zostanie *porzucona*. Ciekawa jest również implementacja tej funkcji. Krótko mówiąc, przesuwa `ESP` w dół stosu o wymaganą liczbę bajtów, przez co `ESP` wskazuje na przydzielony obszar pamięci.

Sprawdźmy:

```
#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);
};
```

Funkcja `_snprintf()` działa tak samo jak `printf()`, tylko zamiast wypisywać tekst na standardowe wyjście (stdout), zapisuje do bufora `buf`. Z kolei funkcja `puts()` kopiuje zawartość bufora `buf` na standardowe wyjście. Oczywiście zamiast korzystać z tych dwóch funkcji, można by użyć `printf()`, ale chcemy zobaczyć wykorzystanie niewielkiego bufora.

[62] W MSVC implementację funkcji można podejrzeć w plikach `alloca16.asm` i `chkstk.asm` w `C:\Program Files (x86)\Microsoft Visual Studio 10.0\VC\crt\src\intel`

MSVC

Skompilujmy (MSVC 2010):

Listing 1.40: MSVC 2010

```
...

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

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

    push   esi
    call   _puts
    add    esp, 28

...
```

Jedyny parametr `alloca()` jest przekazywany przez `EAX` (zamiast przez stos) [63]

GCC + składnia Intela

GCC 4.4.1 generuje podobne wyjście, ale bez wywoływania zewnętrznych funkcji:

Listing 1.41: 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                ; wyrównaj wskaźnik do granicy 16 bajtów
        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 + składnia AT&T

Spójrzmy na ten sam kod, ale w składni AT&T:

Listing 1.42: GCC 4.7.3

[63] Dlatego, że alloca() to nie tyle funkcja, co raczej *compiler intrinsic* (**??**). Jedną z przyczyn, dla której potrzebujemy osobnej funkcji a nie kilku instrukcji w samym kodzie, jest to, że implementacja alloca() z **MSVC!**[64] zawiera kod, który czyta z właśnie zaalokowanej pamięci. W konsekwencji **OS!** mapuje pamięć fizyczną na ten region pamięci wirtualnej. Po wywołaniu funkcji `alloca()` `ESP` pokazuje na blok o długości 600 bajtów, który można użyć na tablicę `buf`.

```
.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
```

Kod jest taki sam jak na poprzednim listingu.

À propos, `movl $3, 20(%esp)` odpowiada `mov DWORD PTR [esp+20], 3` w składni Intela. W składni AT&T, sposób adresowania pamięci *rejestr+przesunięcie* zapisywany jest jako `przesunięcie(%rejestr)`.

(Windows) SEH

Na stosie są przechowywane wpisy **SEH!**[65] dla funkcji (jeśli są one obecne). Więcej o tym tutaj: (**??**).

Ochrona przed przepełnieniem bufora

Więcej o tym tutaj (**??**).

Automatyczne zwalnianie miejsca na stosie

Zmienne lokalne i wpisy SEH są trzymane na stosie prawdopodobnie dlatego, że kiedy funkcja kończy działanie są one zwalniane automatycznie. Odbywa się to za pomocą tylko jednej instrukcji, zmieniającej wartość wskaźnika stosu — często jest to instrukcja ADD. Argumenty funkcji, można tak powiedzieć, również są automatycznie zwalniane na końcu funkcji. Z kolei wszystko, co jest przechowywane na stercie(*heap*), trzeba zwalniać jawnie.

1.9.3 Struktura typowego stosu

Struktura typowego stosu w środowisku 32-bitowym, przed wykonaniem pierwszej instrukcji w funkcji, wygląda następująco:

...	...
ESP-0xC	zmienna lokalna#2, oznaczona w programie **IDA!** jako `var_8`
ESP-8	zmienna lokalna#1, oznaczona w programie **IDA!** jako `var_4`
ESP-4	odłożona wartość `EBP`
ESP	adres powrotu
ESP+4	argument#1, oznaczony w programie **IDA!** jako `arg_0`
ESP+8	argument#2, oznaczony w programie **IDA!** jako `arg_4`
ESP+0xC	argument#3, oznaczony w programie **IDA!** jako `arg_8`
...	...

[65] **SEH!**

1.9.4 Śmieci na stosie

> When one says that something seems random, what one usually means in practice is that one cannot see any regularities in it.
>
> Stephen Wolfram, A New Kind of Science.

Często w tej książce mówimy o „szumie” lub „śmieciach” na stosie czy w pamięci. Skąd one się biorą? Są to pozostałości po poprzednich wywołaniach funkcji.

Krótki przykład:

```
#include <stdio.h>

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

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

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

Kompilujemy...

Listing 1.43: Nieoptymalizujący 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
```

Kompilator się trochę oburzy...

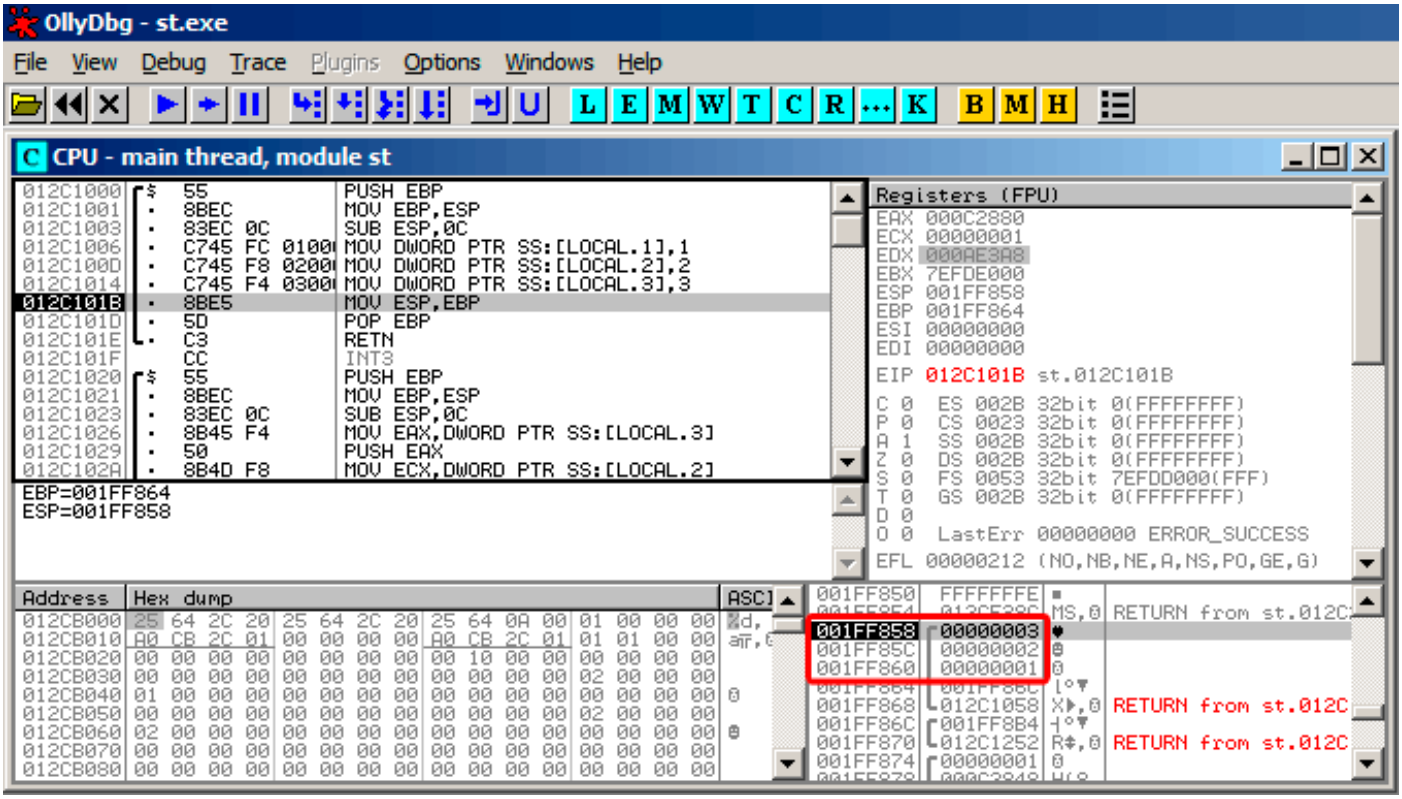
```
c:\Polygon\c>cl st.c /Fast.asm /MD
Microsoft (R) 32-bit C/C++ Optimizing Compiler Version 16.00.40219.01 for 80x86
Copyright (C) Microsoft Corporation.  All rights reserved.

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
```

Ale kiedy uruchoimy skompilowany program...

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

Dziwne, przecież nie ustawialiśmy żadnych zmiennych w `f2()`. Te wartości to „duchy”, które wciąż znajdują się na stosie.

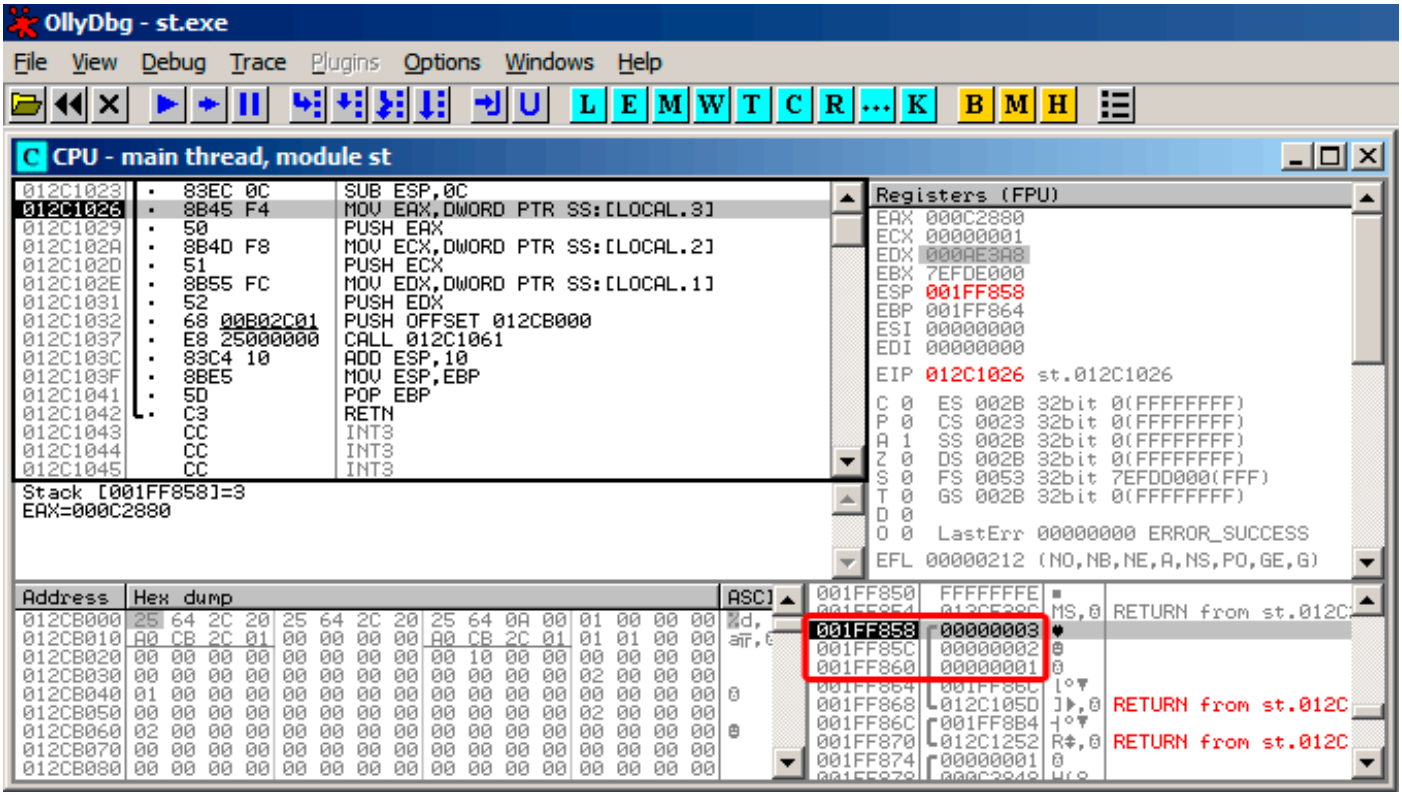
Spróbujmy uruchomić ten przykład w OllyDbg:



Rysunek 1.6: OllyDbg: `f1()`

Kiedy `f1()` ustawia zmienne a, b i c są one zapisywane pod adresem `0x1FF860`, itd.

A kiedy jest wykonywana `f2()`:



Rysunek 1.7: OllyDbg: `f2()`

... a, b i c w funkcji `f2()` znajdują się pod tymi samymi adresami! Nikt jeszcze nie nadpisał tych wartości, więc na razie pozostają one nietknięte. Taka dziwna sytuacja ma miejsce, kiedy kilka funkcji jest wykonywanych jedna po drugiej, a **SP!** jest taki sam (funkcje mają taką samą liczbę argumentów). Wtedy zmienne lokalne będą przechowywane w tych samych adresach na stosie. Podsumowując, wszystkie wartości na stosie (i ogólnie w pamięci) to wartości pozostałe po poprzednich funkcjach. Nie są one losowe, w ścisłym tego słowa znaczeniu, lecz nieprzewidywalne. Czy można coś z tym zrobić? Można by czyścić fragmenty stosu przed wykonywaniem funkcji, ale to za dużo zbędnej (i nieporzebnej) pracy.

MSVC 2013

Przykład był skompilowany w MSVC 2010. Jeden czytelnik tej książki spróbował skompilować to w MSVC 2013, uruchomił i zobaczył 3 liczby w odwrotnej kolejności:

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

Dlaczego? Również spróbowałem skompilować ten przykład w MSVC 2013 i otrzymałem:

Listing 1.44: 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
```

W odróżnieniu od MSVC 2010, MSVC 2013 rozmieścił zmienne a/b/c w funkcji `f2()` w odwrotnej kolejności. Jest to całkowicie poprawne, ponieważ w C/C++ nie ma zdefiniowanego standardu, który by wyznaczał kolejność zmiennych lokalnych na stosie. Przyczyną różnicy są zapewne zmiany w kodzie kompilatora, a więc nowsze MSVC zachowuje się nieco inaczej.

1.9.5 Ćwiczenia

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

1.10 Funkcja niemal pusta

Poniższy fragment kodu znalazłem w projekcie Boolector[66]:

```
// forward declaration. the function is residing in some other module:
int boolector_main (int argc, char **argv);

// executable
int main (int argc, char **argv)
{
        return boolector_main (argc, argv);
}
```

Dlaczego ktoś miałby tak robić? Nie wiem, ale przypuszczam, że `boolector_main()` może być kompilowane do biblioteki dynamicznej (jak np. DLL) i wywoływane w testach. Kod testowy również może przygotować argumenty argc/argv, tak jak to robi **CRT!**.

Ciekawy jest wynik kompilacji:

Listing 1.45: Nieoptymalizujący GCC 8.2 x64 (wyjście w asemblerze)

```
main:
        push    rbp
        mov     rbp, rsp
        sub     rsp, 16
        mov     DWORD PTR -4[rbp], edi
        mov     QWORD PTR -16[rbp], rsi
        mov     rdx, QWORD PTR -16[rbp]
        mov     eax, DWORD PTR -4[rbp]
        mov     rsi, rdx
        mov     edi, eax
        call    boolector_main
        leave
        ret
```

Jest to jak najbardziej poprawny kod, mamy: prolog, niepotrzebne (niezoptymalizowane) przetasowanie dwóch argumentów, `CALL`, epilog i `RET`.

Zobaczmy na efekt kompilacji GCC z włączoną optymalizacją:

Listing 1.46: Optymalizujący GCC 8.2 x64 (wyjście w asemblerze)

```
main:
        jmp     boolector_main
```

Bardzo prosty kod — rejestr i stos zostały nienaruszone, gdyż `boolector_main()` ma taki sam zestaw argumentów. Jedyne co należało zrobić, to przekazać sterowanie pod inny adres.

Przypomina to thunk funkcje.

Później zobaczymy nieco bardziej zaawanzowane przykłady: **??**, **??**.

[66] https://boolector.github.io/

1.11 printf() z wieloma argumentami

Spróbujmy rozszerzyć przykład *Hello, world!* (**??**):

```
#include <stdio.h>

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

1.11.1 x86

x86: 4 argumenty

MSVC

Gdy skompilujemy kod za pomocą MSVC 2010 Express, otrzymamy:

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

...

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

Kod jest niemal identyczny z tym, który widzieliśmy w *Hello, world!* (**??**), lecz teraz argumenty funkcji `printf()` zostały odłożone na stos w odwrotnej kolejności. Pierwszy argument jest zapisywany jako ostatni.

Przy okazji, zmienna typu *int* w środowisku 32-bitowym ma długość 32-bitów, czyli 4 bajty.

Mamy więc 4 argumenty. $4 * 4 = 16$ —zajmują dokładnie 16 bajtów na stosie: a 32-bit wskaźnik na łańcuch znaków i trzy liczby typu *int*.

Gdy wskaźnik stosu (rejestr `ESP`) jest przywracany za pomocą `ADD ESP, X` za wywołaniem funkcji, to często można określić liczbę argumentów, dzieląc X przez 4.

Oczywiście dotyczy to tylko konwencji wywołań *cdecl* i środowiska 32-bitowego!

O konwencjach wywołań przeczytasz (**??**).

Kompilator, gdy kilka funkcji jest wywoływanych jedna za drugą, może połączyć kilka instrukcji „ADD ESP, X” w jedną i umieścić ją po ostatnim wywołaniu:

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

Przykład prawdziwego kodu:

Listing 1.47: x86

```
.text:100113E7    push    3
.text:100113E9    call    sub_100018B0 ; wykorzystuje jeden argument (3)
.text:100113EE    call    sub_100019D0 ; funkcja bez argumentów
```

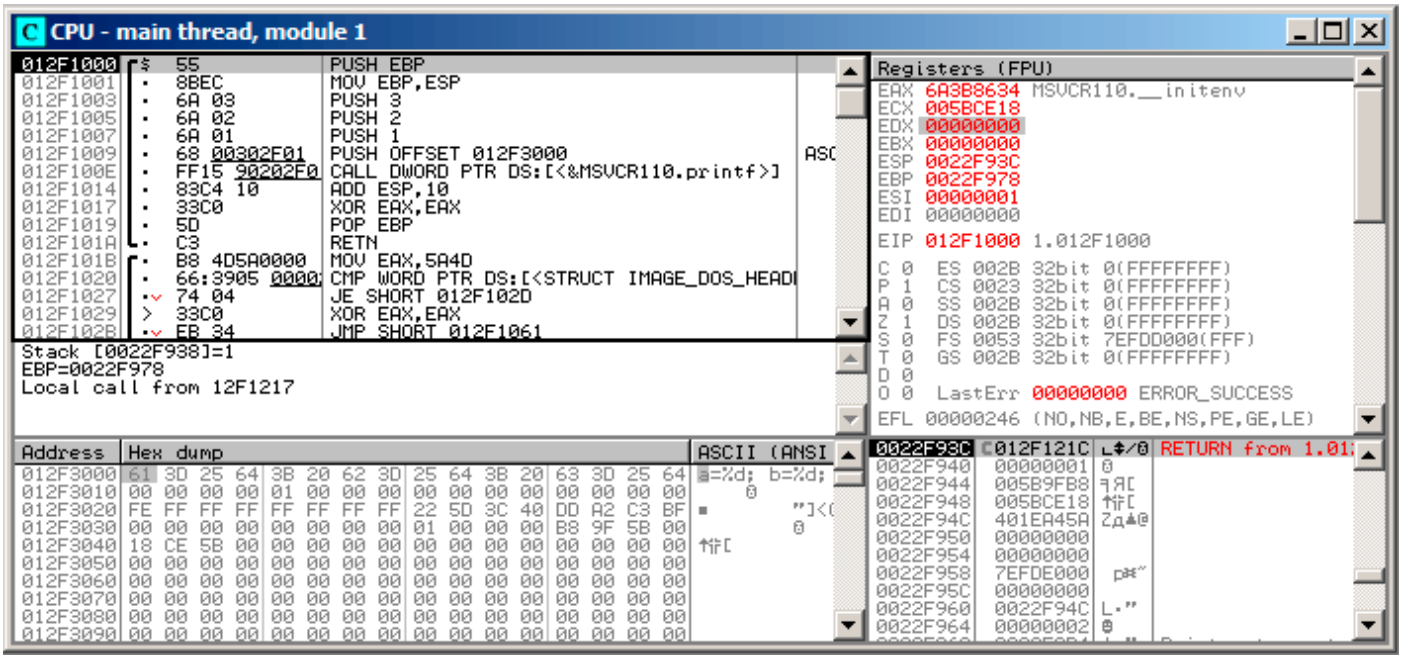
```
.text:100113F3    call    sub_10006A90 ; funkcja bez argumentów
.text:100113F8    push    1
.text:100113FA    call    sub_100018B0 ; wykorzystuje jeden argument (1)
.text:100113FF    add     esp, 8       ; jednocześnie sprząta dwa argumenty ze stosu
```

MSVC and OllyDbg

Skorzystajmy z OllyDbg. Jest to jeden z najpopularniejszych debuggerów pod win32, działających w trybie użytkownika. Przykład skompilujemy w MSVC 2012 z opcją `/MD`, która oznacza dynamiczne linkowanie do `MSVCR*.DLL`, by łatwo można było rozpoznać zaimportowane funkcje w debuggerze.

Możemy załadować plik wykonywalny do OllyDbg. Pierwszy breakpoint jest w `ntdll.dll`, naciskamy F9 (run). Drugi breakpoint jest w kodzie **CRT!**. Musimy odnaleźć funkcję `main()`.

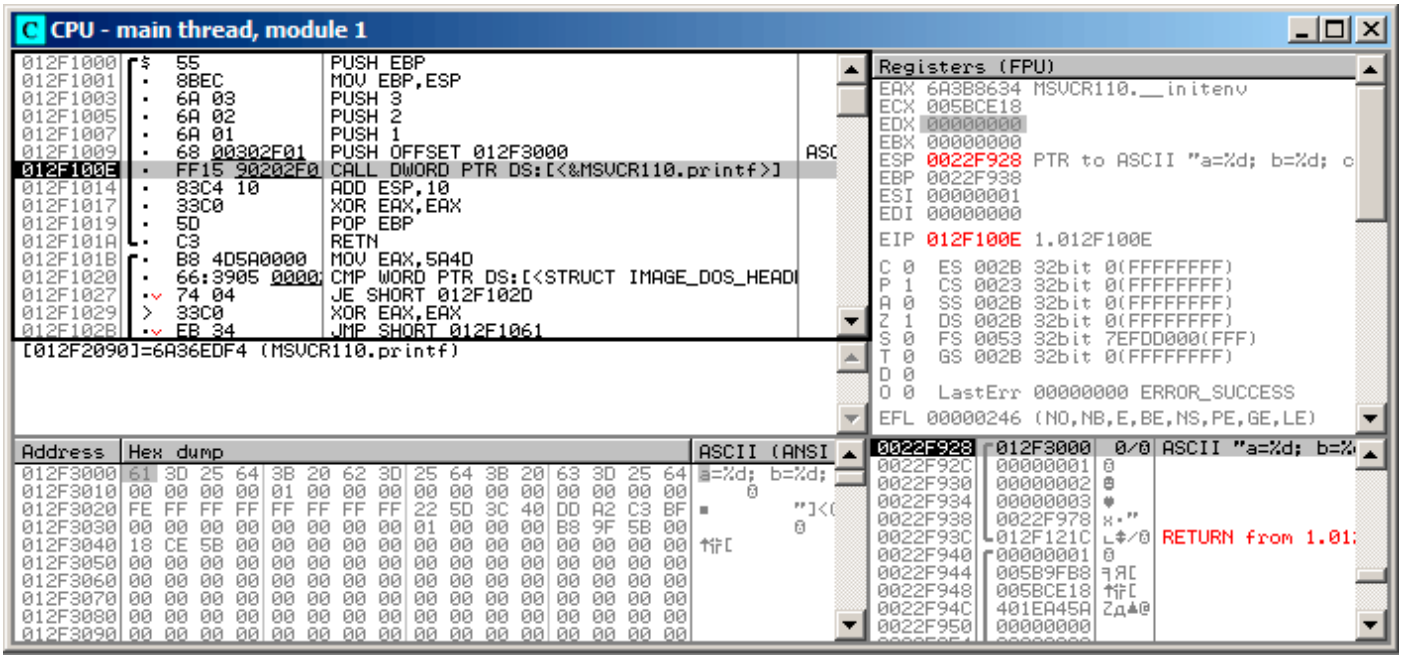
Można to zrobić scrollując na samą górę kodu (MSVC umieszcza funkcję `main()` na samym początku sekcji kodu):



Rysunek 1.8: OllyDbg: sam początek funkcji `main()`

Klikamy na instrukcję `PUSH EBP`, wciskamy F2 (ustaw breakpoint) i wciskamy F9 (run). Dzięki temu przeskoczymy kod z **CRT!**, którym nie jesteśmy na razie zainteresowani.

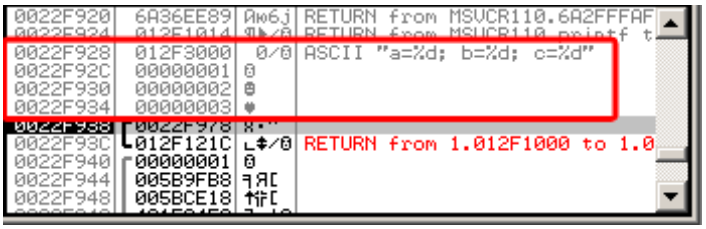
Naciśnij F8 (step over) 6 razy, by przeskoczyć 6 kolejnych instrukcji:



Rysunek 1.9: OllyDbg: przed wykonaniem `printf()`

PC! pokazuje teraz na instrukcję `CALL printf`. OllyDbg, jak inne debuggery, podświetla rejestry, które zostały zmienione. Za każdym razem, gdy naciskasz F8, `EIP` zmienia się i jego wartość jest wyświetlana na czerwono. `ESP` również się zmienia, gdyż argumenty są przekazywane przez stos.

Gdzie na stosie są nasze wartości? Spójrz na obszar w prawym, dolnym rogu okna:



Rysunek 1.10: OllyDbg: stos po odłożeniu argumentów (czerwona ramka została dodana przez autora w programie graficznym)

Widzimy 3 kolumny: adres na stosie, wartość i dodatkowo komentarz OllyDbg. OllyDbg rozumie wskaźniki na łańcuchy znaków, więc wypisuje wartość tego łańcucha jako komentarz .

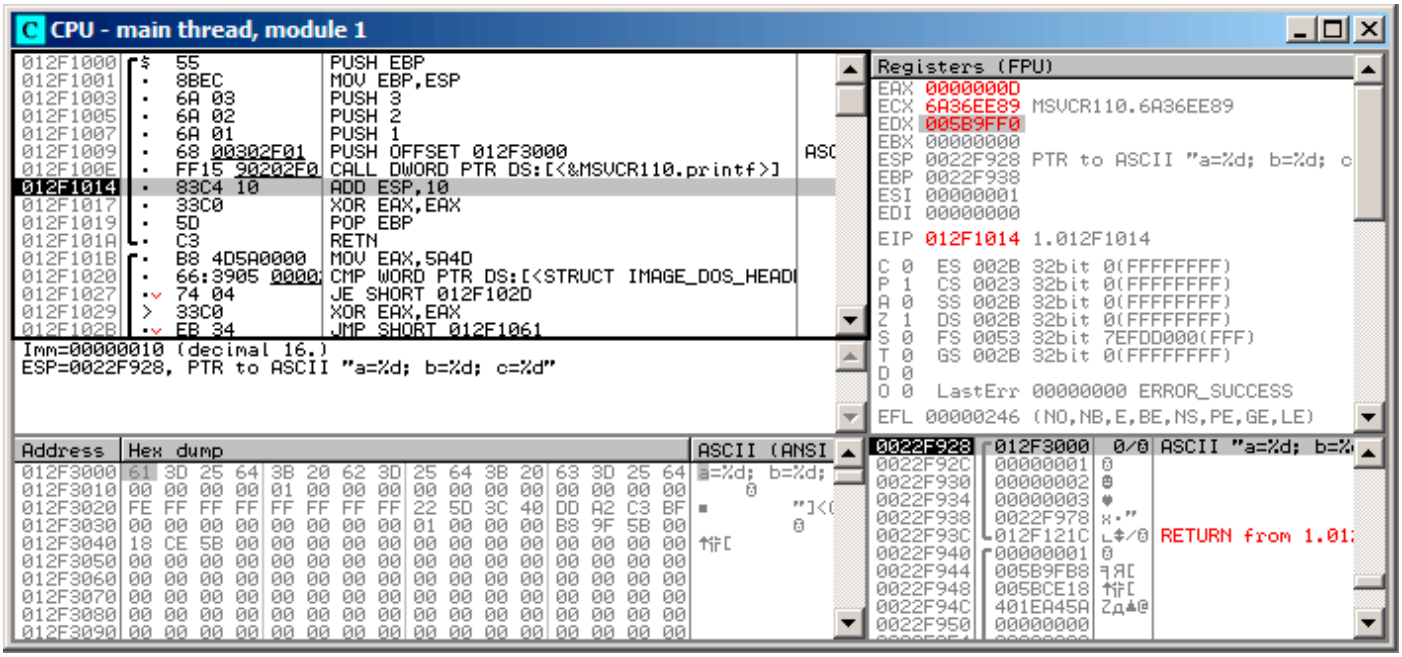
Możesz kliknąć prawym przyciskiem myszy na łańcuch znaków z formatem i wybrać „Follow in dump”. Łańcuch znaków pojawi się w lewym, dolnym oknie debuggera, wyświetlającym zawartość pamięci. Możesz edytować jej zawartość. Mógłbyś zmienić format łańcucha znaków, tak by na wyjście został wypisany inny tekst. W tym przykładzie nie jest to użyteczna funkcjonalność, ale możesz jej użyć w ramach ćwiczeń, by lepiej poznać opisane wcześniej mechanizmy.

Wciśnij F8 (step over).

Zobaczymy następujący rezultat w konsoli:

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

Sprawdźmy jak zmieniły się rejestry i stos:



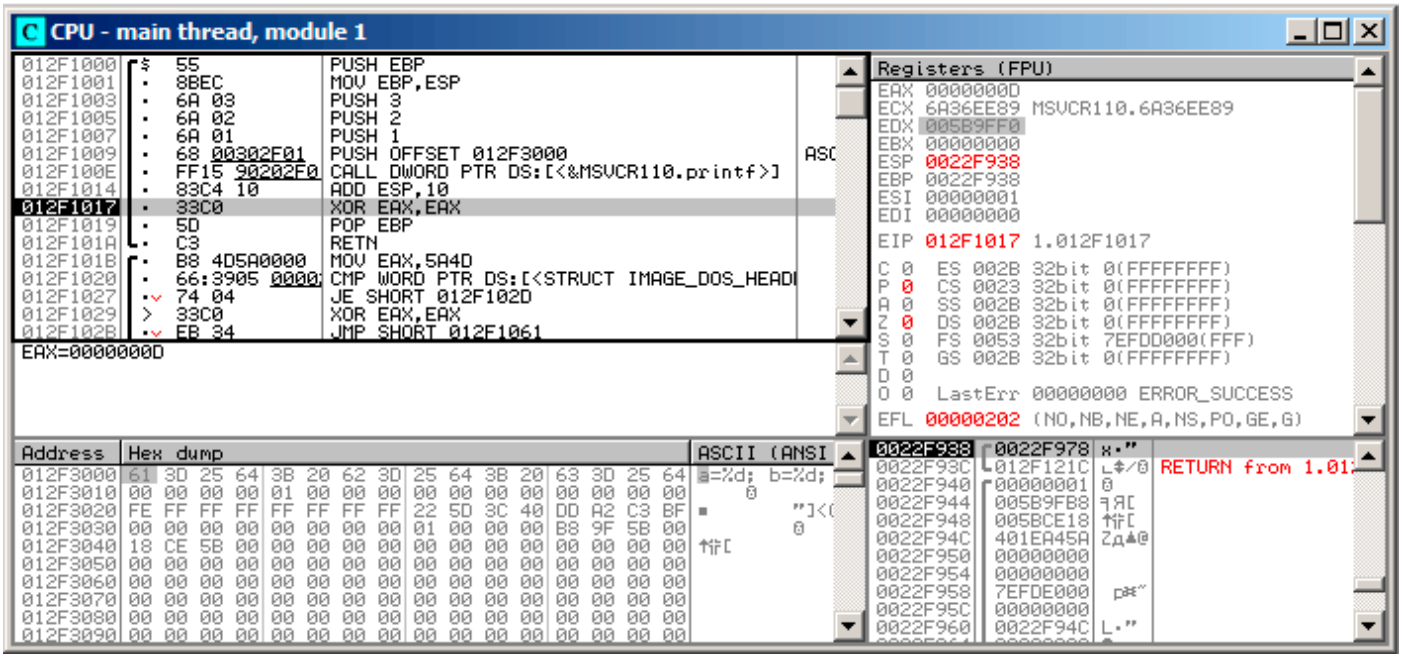
Rysunek 1.11: OllyDbg po wykonaniu funkcji `printf()`

Rejestr `EAX` zawiera teraz `0xD` (13). Jest to spodziewana wartość, ponieważ `printf()` zwraca liczbę wypisanych znaków. Zmieniła się wartość `EIP`: zawiera teraz adres kolejnej instrukcji, występującej bezpośrednio za `CALL printf`. `ECX` i `EDX` również się zmieniły. Najwyraźniej implementacja `printf()` wykorzystała je do własnych celów.

Ważnym faktem jest to, że ani wartość `ESP`, ani stan stosu się nie zmieniły!

Łatwo zauważyć, że łańcuch znaków z formatem oraz 3 powiązane z nim argumentu wciąż tam są. Jest to konwencja wywoływania *cdecl*: funkcja wywoływana nie przywraca `ESP` do pierwotnego stanu. Ta odpowiedzialność spoczywa na wywołującym.

Ponownie wciśnij F8, by wykonać instrukcję `ADD ESP, 10`:



Rysunek 1.12: OllyDbg: po wykonaniu instrukcji `ADD ESP, 10`

Zmieniła się wartość `ESP`, ale wartości na stosie zostały niezmienione! Można się tego spodziewać; nikt nie musi ich nadpisywać. Wszystko powyżej wskaźnika stosu (**SP!**) to *szum* czy *śmieci* i nie ma znaczenia. Czyszczenie nieużywanych wpisów na stosie byłoby stratą czasu i nikt tego nie potrzebuje.

GCC

Skompilujmy ten sam program na Linuksie, za pomocą GCC 4.4.1 i sprawdźmy wynik w programie **IDA!**:

```
main            proc near

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

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

Jedyną zauważalną różnicą jest inny sposób odkładania argumentów na stos. W tym przykładzie GCC explicite podaje adres w obrębie stosu i nie używa instrukcji `PUSH` / `POP`.

GCC i GDB

Wczytajmy plik wykonywalny do debuggera **GDB!**[67].

[67] **GDB!**

Opcja `-g` sprawia, że kompilator zapisuje do pliku wykonywalnego informacje użyteczne do debuggowania.

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

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

Listing 1.48: Ustawmy breakpoint w funkcji `printf()`

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

Kontynuujmy wykonywanie kodu. Gdybyśmy dysponowali kodem źródłowym funkcji `printf()`, **GDB!** mógłby go wyświetlić obok instrukcji asemblera.

```
(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.
```

Wypisz 10 wartości ze stosu. Skrajna lewa kolumna oznacza adres, pod którym wartości ze stosu znajdują się w pamięci.

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

Pierwszy element to adres powrotu (**RA!**) (`0x0804844a`). Możemy to sprawdzić, deasemblując pamięć pod tym adresem (instruujemy GDB, by wypisał 5 elementów ze stosu, interpretując je jako instrukcje asemblera):

```
(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
```

Pojawiająca się dwa razy `XCHG` działa jak pusta instrukcja **NOP!**, gdyż w tym przypadku ustawia wartość rejestr AX na jego bieżącą wartość.

Drugim elementem jest (`0x080484f0`) — to adres łańcucha znaków z formatem:

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

Kolejne 3 elementy (1,2,3) to argumenty funkcji `printf()`. Pozostałe elementy to prawdopodobnie „śmieci” znajdujące się na stosie, lecz mogłyby to być wartości używane przez inne funkcje, ich zmienne lokalnce, etc. Możemy je na razie zignorować.

Kontynuujmy za pomocą „finish”. Polecenie powoduje, że GDB wykona wszystkie instrukcje aż do końca funkcji. W tym przypadku do końca funkcji `printf()`.

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

GDB! pokazuje jaką wartość `printf()` zwróciła przez rejestr `EAX` (13). Jest to liczba znaków, które zostały wypisane i jest to dokładnie tyle, ile widzieliśmy w przykładzie z OllyDbg.

Widzimy również „return 0;” wraz z informacją, że to wyrażenie jest w pliku `1.c`, w linii 6. Plik `1.c` znajduje się w bieżącym katalogu i tam **GDB!** znalazł ten łańcuch znaków. Skąd debugger wiedział, która linia kodu w C jest właśnie wykonywana? Kompilatory, gdy generują informacje dla debuggera, zapisują również tablicę z zależnościami między liniami kodu źródłowego a adresami instrukcji, GDB jest w końcu debuggerem pracującym z kodem źródłowym.

Przyjrzyjmy się rejestrom. W `EAX` znajduje się wartość 13:

```
(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>
...
```

Deasemblujemy kolejne instrukcje. Strzałka pokazuje na instrukcję, która zostanie wykonana jako kolejna.

```
(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! używa domyślnie składni AT&T. Możemy jednak przełączyć się na składnię Intela.

```
(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.
```

Wykonajmy kolejną linię. **GDB!** pokazuje klamrowy nawias zamykający, który oznacza koniec bloku kodu.

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

Sprawdźmy rejestry po wykonaniu instrukcji `MOV EAX, 0`. Tak jak się spodziewamy, `EAX` zawiera teraz wartość 0.

```
(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: 9 argumentów

Zmieńmy nieco nasz przykład, by zobaczyć jak na x64 argumenty przekazywane są przez stos. Zwiększymy liczbę argumentów do 9 (łańcuch znaków z formatem + 8 zmiennych typu *int*):

```
#include <stdio.h>

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

MSVC

Jak wspomnieliśmy poprzednio — pierwsze 4 argumenty na Win64 przekazywane są przez rejestry `RCX`, `RDX`, `R8` i `R9` a pozostałe przez stos[68].

Widać to na wygenerowanym listingu. Stos został przygotowany przy pomocy instrukcji `MOV`, zamiast `PUSH` - wartości zostały odłożone ze wskazaniem adresu.

Listing 1.49: 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

        ; zwróć 0
        xor     eax, eax

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

Uważny czytelnik może się zastanawiać dlaczego dla wartości typu *int* zostało zaalokowanych 8 bajtów, skoro wystarczą tylko 4? Dla przypomnienia: 8 bajtów jest alokowanych dla każdego typu daych, krótszego niż 64 bity. Powodem jest wygoda, łatwo w ten sposób wyliczyć adres dowolnego argumentu. Poza tym, wszystkie są przechowywane w pamięci na wyrównanych adresach. W środowisku 32-bitowym jest podobnie - 4 bajty są zarezerwowane dla wszystkich typów nie dłuższych niż 4 bajty[69].

[68] przyp. tłum. - rzecz wygląda inaczej w przypadku przekazywania argumentów zmiennoprzecinkowych, gdy może zostać wykorzystany rejestr wektorowy `XMM0` - `XMM3`

[69] przyp. tłum. - np. zmienna typu long long zajmie 8 bajtów.

GCC

Kompilacja na x86-64 systemach *NIX daje podobny wynik jak MSVC, jednak teraz 6 pierwszych argumentów jest przekazywanych przez rejestry `RDI`, `RSI`, `RDX`, `RCX`, `R8` i `R9`, a cała reszta przez stos.

GCC generuje kod, który przechowuje wskaźnik stosu w `EDI` zamiast w `RDI` —co już zdążyliśmy zauważyć: **??**.

Widzieliśmy też, że rejestr `EAX` został wyzerowany przed wywołaniem funkcji `printf()`: **??**.

Listing 1.50: Optymalizujący 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 ; liczba użytych rejestrów wektorowych
        mov     DWORD PTR [rsp+16], 8
        mov     DWORD PTR [rsp+8], 7
        mov     DWORD PTR [rsp], 6
        call    printf

        ; zwróć 0

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

GCC + GDB

Wczytajmy plik wykonywalny do debuggera **GDB!**

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

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

Listing 1.51: Ustawiamy breakpoint na funkcji `printf()` i zaczynamy wykonanie

```
(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.
```

Rejestry `RSI` / `RDX` / `RCX` / `R8` / `R9` zostały ustawione na spodziewane wartości. W `RIP` przechowywany jest adres pierwszej instrukcji z funkcji `printf()`.

```
(gdb) info registers
rax            0x0      0
rbx            0x0      0
rcx            0x3      3
rdx            0x2      2
rsi            0x1      1
rdi            0x400628 4195880
```

```
rbp            0x7fffffffdf60   0x7fffffffdf60
rsp            0x7fffffffdf38   0x7fffffffdf38
r8             0x4      4
r9             0x5      5
r10            0x7fffffffdce0   140737488346336
r11            0x7ffff7a65f60   140737348263776
r12            0x400440 4195392
r13            0x7fffffffe040   140737488347200
r14            0x0      0
r15            0x0      0
rip            0x7ffff7a65f60   0x7ffff7a65f60 <__printf>
...
```

Listing 1.52: Łańcuch znaków z formatem

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

Wyświetlimy zawartość stosu za pomocą komendy x/g —*g* oznacza *giant word*, czyli wartość 64-bitową.

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

Jak w poprzednim przykładzie, pierwszy elementem na stosie jest adres powrotu **RA!**. 3 wartości przekazywane są przez stos: 6, 7, 8. Widać, że 8 przekazywane jest z wypełnionymi 32 starszymim bitami: `0x00007fff00000008`. Nie stanowi to problemu, ponieważ argument jest typu *int* type, który jest 32-bitowy. Starsze części rejestrów, albo elementów na stosie, mogą zawierać „losowe śmieci”.

Jeśli sprawdzimy gdzie zostanie zwrócone sterowanie po zakończeniu funkcji `printf()`, **GDB!** pokaże funkcję main:

```
(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.
```

Przeskoczmy na koniec funkcji `printf()` i wykonajmy jedną linię kodu z funkcji `main()`. W ramach tej linii rejestr `EAX` został wyzerowany. Debugger pokazuje teraz na koniec bloku kodu. Możemy się upewnić, że `EAX` rzeczywiście ma wartość 0, a `RIP` pokazuje teraz na instrukcję `LEAVE`, przedostatnią w funkcji `main()`.

```
(gdb) finish
Run till exit from #0  __printf (format=0x400628 "a=%d; b=%d; c=%d; d=%d; e=%d; f=%d; g=%d; h=%↙
    ↳ d\n") at printf.c:29
a=1; b=2; c=3; d=4; e=5; f=6; g=7; h=8
main () at 2.c:6
6               return 0;
```

```
Value returned is $1 = 39
(gdb) next
7       };
(gdb) info registers
rax            0x0      0
rbx            0x0      0
rcx            0x26     38
rdx            0x7ffff7dd59f0   140737351866864
rsi            0x7fffffd9       2147483609
rdi            0x0      0
rbp            0x7fffffffdf60   0x7fffffffdf60
rsp            0x7fffffffdf40   0x7fffffffdf40
r8             0x7ffff7dd26a0   140737351853728
r9             0x7ffff7a60134   140737348239668
r10            0x7fffffffd5b0   140737488344496
r11            0x7ffff7a95900   140737348458752
r12            0x400440 4195392
r13            0x7fffffffe040   140737488347200
r14            0x0      0
r15            0x0      0
rip            0x40057b 0x40057b <main+78>
...
```

1.11.2 ARM

ARM: 4 argumenty

Tradycyjny sposób w ARM na przekazywanie argumentów (tzw. konwencja wywoływania funkcji) wygląda następująco: pierwsze 4 argumenty są przekazywane przez rejestry `R0` - `R3`, a pozostałe przez stos. Przypomina to konwencję fastcall (**??**) czy win64 (**??**).

32-bitowy ARM

Nieoptymalizujący Keil 6/2013 (tryb ARM)

Listing 1.53: Nieoptymalizujący Keil 6/2013 (tryb ARM)

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

Pierwsze 4 argumenty zostały przekazane przez rejestry `R0` - `R3` w następującej kolejności: wskaźnik na łańcuch znaków z formatem w `R0`, następnie 1 w `R1`, 2 w `R2` i 3 w `R3`. Instrukcja z adresu `0x18` zapisuje 0 do rejestru `R0` —jest to część instrukcji *return 0* z kodu C. Nic nadzwyczajnego.

Optymalizujący Keil 6/2013 generuje taki sam kod.

Optymalizujący Keil 6/2013 (tryb Thumb)

Listing 1.54: Optymalizujący Keil 6/2013 (tryb Thumb)

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

```
.text:0000000E 00 20        MOVS    R0, #0
.text:00000010 10 BD        POP     {R4,PC}
```

Porównując do niezoptymalizowanego kodu w trybie ARM, nie widać wyraźnej różnicy.

Optymalizujący Keil 6/2013 (tryb ARM) + usunięty return

Zmieńmy nieco przykład, usuwając *return 0*:

```
#include <stdio.h>

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

Efekt jest dość nieoczekiwany:

Listing 1.55: Optymalizujący Keil 6/2013 (tryb ARM)

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

W zoptymalizowanej wersji w trybie ARM widzimy, że ostatnią instrukcją jest `B` zamiast spodziewanej `BL`. Inną różnicą jest brak prologu i epilogu (instrukcje zachowujące wartości rejestrów `R0` i **LR!**), które wystąpiły w wersji niezoptymalizowanej.

Instrukcja `B` skacze pod inny adres, bez zmiany rejestru **LR!**, podobnie jak `JMP` w x86. Dlaczego to działa? Nowy kod w działaniu jest równoważny poprzedniej wersji z dwóch powodów:

1) nie jest modyfikowany **SP!** (wskaźnik stosu),

2) wywołanie `printf()` jest ostatnią instrukcją, nic się dalej nie dzieje.

Funkcja `printf()` po zakończeniu pracy zwraca sterowanie do adresu z **LR!**. **LR!** przechowuje adres miejsca, z którego nasza funkcja została wywołana, a więc tam też zostanie zwrócone sterowanie. Nie musimy zapisywać **LR!**, ponieważ nie ma konieczności jego modyfikacji. W programie nie ma innych wywołań funkcji niż wywołanie `printf()` i to z tego powodu nie musimy modyfikować **LR!**. Co więcej, po wywołaniu funkcji nie musimy już nic robić! To własnie dzięki tym wszystkim okolicznościom optymalizacja była możliwa.

Podobna optymalizacja pojawia się często w funkcjach, których ostatnią instrukcją jest wywołanie innej funkcji. Podobny przykład widać tutaj: **??**.

Nieco prostszy przykład opisywaliśmy już wcześniej: **??**.

ARM64

Nieoptymalizujący GCC (Linaro) 4.9

Listing 1.56: Nieoptymalizujący GCC (Linaro) 4.9

```
.LC1:
        .string "a=%d; b=%d; c=%d"
f2:
; zapisz FP i LR w ramce stosu:
        stp     x29, x30, [sp, -16]!
; ustaw wskaźnik ramki stosu (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
; przywróć FP i LR
        ldp     x29, x30, [sp], 16
        ret
```

Pierwsza instrukcja `STP` (*Store Pair*) zapisuje na stos **FP!** (X29) i **LR!** (X30). Kolejna, `ADD X29, SP, 0`, tworzy ramkę stosu — to po prostu zapisanie wartości **SP!** do X29.

Następnie widać już znaną parę instrukcji `ADRP` / `ADD`, która konstruuje wskaźnik na łańcuch znaków. *lo12* oznacza młodsze 12 bitów — linker umieści młodsze 12 bitów adresu LC1 w kodzie operacji (opcode) instrukcji `ADD`. Trzy ostatnie argumenty funkcji `printf()` to literały typu *int*, więc są ładowane do 32-bitowych części rejestru.

Optymalizujący GCC (Linaro) 4.9 generuje taki sam kod.

ARM: 9 argumentów

Użyjmy ponownie przykładu z 9 argumentami: **??**.

```
#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;
};
```

Optymalizujący Keil 6/2013: tryb ARM

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

Kod można podzielić na kilka części.

- Prolog.

  Pierwsza instrukcja — `STR LR, [SP,#var_4]!` — zapisuje **LR!** na stosie, ponieważ użyjemy tego rejestru przy wywołaniu funkcji `printf()`. Wykrzyknik na końcu oznacza *pre-index*.

Pre-index oznacza, że **SP!** zostanie najpierw zmniejszony o 4, a następnie pod tym zmodyfikowanym adresem zostanie zapisana wartość **LR!**. Podobnie działa instrukcja `PUSH` na x86. Więcej przeczytasz o tym tutaj: **??**.

Druga instrukcja — `SUB SP, SP, #0x14` — zmniejsza **SP!** (wskaźnik stosu) by zaalokować na stosie `0x14` (20) bajtów. Będziemy przekazywać do funkcji `printf()` 5 32-bitowych wartości, każda z nich zajmuje 4 bajty — razem zajmą więc dokładnie 20 bajtów. Pozostałe 4 32-bitowe wartości zostaną przekazane przez rejestry.

- Przekazania argumentów 5, 6, 7 i 8 przez stos.

  Najpierw argumenty zapisywane są do rejestrów, odpowiednio: `R0`, `R1`, `R2` and `R3`. Następnie instrukcja `ADD R12, SP, #0x18+var_14` zapisuje do rejestru `R12` adres stosu, pod którym te wartości zostaną umieszczone. *var_14* to makro równe -0x14, stworzone przez program **IDA!** by poprawić czytelność kodu pracującego ze stosem. Makra *var_?* wygenerowane w programie **IDA!** odpowiadają zmiennym lokalnym umieszczonym na stosie. Zatem ostatecznie do rejestru `R12` trafi adres `SP+4`.

  Kolejna instrukcja — `STMIA R12, R0-R3` — zapisuje zawartość rejestrów `R0`-`R3` w pamięci, pod adres z `R12`. `STMIA` to skrót od *Store Multiple Increment After*. *Increment After* oznacza, że `R12` będzie zwiększany o 4, po każdej zapisanej wartości rejestru.

- Przekazanie argumentu 4 przez stos.

  Najpierw 4 jest zapisywana w `R0`, a następnie za pomocą instrukcji `STR R0, [SP,#0x18+var_18]` odkładana jest na stos. *var_18* to -0x18, a więc ostateczne przesunięcie (offset) wynosi 0, a więc wartość z `R0` (4) trafi pod adres z **SP!**.

- Przekazanie argumentów 1, 2, 3 przez rejestry.

  3 pierwsze argumenty — 1, 2, 3 (a, b, c w łańcuchu formatującym) — są przekazywane przez rejestry `R1`, `R2` i `R3` tuż przed wywołaniem funkcji `printf()`.

- Wywołanie `printf()`.

- Epilog.

  Instrukcja `ADD SP, SP, #0x14` przywraca wskaźnik stosu **SP!** do poprzedniej wartości, porzucając wszystkie tam zapisane dane. Oczywiście odłożone wartości wciąż tam są, ale zostaną nadpisane przy wywołaniach kolejnych funkcji.

  Instrukcja `LDR PC, [SP+4+var_4],#4` wczytuje do **PC!** wcześniej odłożoną na stosie wartość **LR!**, powodując wyjście z funkcji. Tym razem na końcu instrukcji nie ma wykrzyknika — tak, najpierw **PC!** jest ładowany z adresu przechowywanego w **SP!** ($4 + var\_4 = 4 + (-4) = 0$), a następnie **SP!** jest zwiększany o 4.

  Dlaczego **IDA!** w taki sposób wyświetla instrukcje? W ten sposób łatwiej pokazać układ danych na stosie, widać tutaj, że zmienna `var_4` została stworzona do zapisu wartości **LR!** na stosie lokalnym.

  Instrukcje jest na swój sposób podobna do `POP PC` w x86[70].

Optymalizujący Keil 6/2013: tryb Thumb

```
.text:0000001C             printf_main2
.text:0000001C
.text:0000001C             var_18 = -0x18
.text:0000001C             var_14 = -0x14
.text:0000001C             var_8  = -8
.text:0000001C
.text:0000001C 00 B5        PUSH    {LR}
.text:0000001E 08 23        MOVS    R3, #8
.text:00000020 85 B0        SUB     SP, SP, #0x14
.text:00000022 04 93        STR     R3, [SP,#0x18+var_8]
.text:00000024 07 22        MOVS    R2, #7
.text:00000026 06 21        MOVS    R1, #6
.text:00000028 05 20        MOVS    R0, #5
.text:0000002A 01 AB        ADD     R3, SP, #0x18+var_14
```

[70] Niemożliwe jest ustawienie wartości w `IP/EIP/RIP` za pomocą `POP` w x86, ale poza tym to trafne porównanie.

```
.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}
```

Wyjście jest podobne do poprzedniego przykładu. Tym razem jest to kod w trybie Thumb i wartości są umieszczane na stosie w innym porządku: najpierw odkładana jest wartość 8, jako druga odkładana jest grupa wartości 5, 6, 7 a jako trzecia odkładana jest wartość 4.

Optymalizujący Xcode 4.6.3 (LLVM): tryb ARM

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

Niemal to samo widzieliśmy poprzednio, za wyjątkiem istrukcji `STMFA` (Store Multiple Full Ascending), która jest synonimem `STMIB` (Store Multiple Increment Before). Instrukcja najpierw zwiększa wartość rejestru **SP!**, a po tym zapisuje wartości rejestrów (z drugiego operandu) do pamięci. Te dwa kroki odbywają się w odwrotnej kolejności niż w instrukcji `STMIA`.

Można również zwrócić uwagę, że instrukcje zostały rozrzucone jakby losowo. Na przykład wartość w rejestrze `R0` jest ustawiana w trzech różnych miejscach: `0x2918`, `0x2920` i `0x2928`, a można by to zrobić w jednym.

Musimy pamiętać, że ten porządek jest pozornie losowy, kompilator ma swoje powodu do takiego szeregowania instrukcji, ponieważ kieruje się efektywnością kodu w trakcie wykonania.

Procesor z reguły próbuje równolegle wykonywać instrukcje położone obok siebie. Na przykład, instrukcje jak `MOVT R0, #0` i `ADD R0, PC, R0` nie mogę być wykonywane równocześnie, gdyż obie modyfikują ten sam rejestr `R0`. Z drugiej strony, `MOVT R0, #0` i `MOV R2, #4` mogą być wykonywane równolegle, gdyż nie ma konfliktu między wynikami ich pracy. Prawdopodobnie kompilator starał się tak uszeregować instrukcje, by mogły być wykonywane równolegle tam, gdzie to możliwe.

Optymalizujący Xcode 4.6.3 (LLVM): tryb Thumb-2

```
__text:00002BA0              _printf_main2
__text:00002BA0
__text:00002BA0              var_1C = -0x1C
__text:00002BA0              var_18 = -0x18
__text:00002BA0              var_C  = -0xC
__text:00002BA0
__text:00002BA0 80 B5          PUSH     {R7,LR}
__text:00002BA2 6F 46          MOV      R7, SP
__text:00002BA4 85 B0          SUB      SP, SP, #0x14
__text:00002BA6 41 F2 D8 20    MOVW     R0, #0x12D8
__text:00002BAA 4F F0 07 0C    MOV.W    R12, #7
__text:00002BAE C0 F2 00 00    MOVT.W   R0, #0
__text:00002BB2 04 22          MOVS     R2, #4
__text:00002BB4 78 44          ADD      R0, PC  ; char *
__text:00002BB6 06 23          MOVS     R3, #6
__text:00002BB8 05 21          MOVS     R1, #5
__text:00002BBA 0D F1 04 0E    ADD.W    LR, SP, #0x1C+var_18
__text:00002BBE 00 92          STR      R2, [SP,#0x1C+var_1C]
__text:00002BC0 4F F0 08 09    MOV.W    R9, #8
__text:00002BC4 8E E8 0A 10    STMIA.W  LR, {R1,R3,R12}
__text:00002BC8 01 21          MOVS     R1, #1
__text:00002BCA 02 22          MOVS     R2, #2
__text:00002BCC 03 23          MOVS     R3, #3
__text:00002BCE CD F8 10 90    STR.W    R9, [SP,#0x1C+var_C]
__text:00002BD2 01 F0 0A EA    BLX      _printf
__text:00002BD6 05 B0          ADD      SP, SP, #0x14
__text:00002BD8 80 BD          POP      {R7,PC}
```

Wyjście jest prawie takie samo jak w poprzednim przykładzie, różnicą jest użycie instrukcji z trybu Thumb/Thumb-2.

ARM64

Nieoptymalizujący GCC (Linaro) 4.9

Listing 1.57: Nieoptymalizujący GCC (Linaro) 4.9

```
.LC2:
        .string "a=%d; b=%d; c=%d; d=%d; e=%d; f=%d; g=%d; h=%d\n"
f3:
; przydziel miejsce na stosie:
        sub     sp, sp, #32
; zapisz FP i LR w ramce stosu:
        stp     x29, x30, [sp,16]
; ustaw wskaźnik ramki stosu (FP=SP+16):
        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           ; 9. argument
        str     w1, [sp]        ; zapisz 9. argument na stos
        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
; przywróć FP i LR
        ldp     x29, x30, [sp,16]
        add     sp, sp, 32
        ret
```

Pierwsze 8 argumentów przekazywanych jest w rejestrach X- oraz W-: [*Procedure Call Standard for the ARM 64-bit Architecture (AArch64)*, (2013)][71]. Wskaźnik na łańcuch znaków wymaga 64-bitowego rejestru, trafia więc do `X0`. Wszystkie pozostałe wartości są typu *int*, mają szerokość 32 bitów i umieszczane są w młodszych 32-bitowych częściach rejestrów (W-). Dziewiąty argument (8) jest przekazywany przez stos. Nie można przekazać dużej liczby argumentów przez rejestry, gdyż ich liczba jest ograniczona.

Optymalizujący GCC (Linaro) 4.9 generuje taki sam kod.

1.11.3 Wnioski

Tak wygląda szkielet wywołania funkcji:

Listing 1.58: x86

```
...
PUSH trzeci argument
PUSH drugi argument
PUSH pierwszy argument
CALL funkcja
; zmodyfikować wskaźnik stosu jeśli trzeba (§jeśli§ §trzeba§)
```

Listing 1.59: x64 (MSVC)

```
MOV RCX, pierwszy argument
MOV RDX, drugi argument
MOV R8, trzeci argument
MOV R9, 4-y argument
...
PUSH 5-y, 6-y argument, itd (jeśli trzeba)
CALL funkcja
; §zmodyfikować wskaźnik stosu (jeśli trzeba)§
```

Listing 1.60: x64 (GCC)

```
MOV RDI, pierwszy argument
MOV RSI, drugi argument
MOV RDX, trzeci argument
MOV RCX, czwarty argument
MOV R8, 5-y argument
MOV R9, 6-y argument
...
PUSH 7-y, 8-y argument, itd (jeśli trzeba)
CALL funkcja
; §zmodyfikować wskaźnik stosu (jeśli trzeba)§
```

Listing 1.61: ARM

```
MOV R0, pierwszy argument
MOV R1, drugi argument
MOV R2, trzeci argument
MOV R3, czwarty argument
; §przekazać 5-y, 6-y argument, itd, przez stos (jeśli trzeba)§
BL funkcja
; §zmodyfikować wskaźnik stosu (jeśli trzeba)§
```

Listing 1.62: ARM64

```
MOV X0, pierwszy argument
MOV X1, drugi argument
MOV X2, trzeci argument
MOV X3, 4-y argument
MOV X4, 5-y argument
MOV X5, 6-y argument
MOV X6, 7-y argument
MOV X7, 8-y argument
; §przekazać 9-y, 10-y argument, itd, przez stos (jeśli trzeba)§
BL funkcja
; §zmodyfikować wskaźnik stosu (jeśli trzeba)§
```

[71] Dostęp także przez http://go.yurichev.com/17287

Listing 1.63: MIPS

```
LI $4, pierwszy argument ; AKA $A0
LI $5, drugi argument ; AKA $A1
LI $6, trzeci argument ; AKA $A2
LI $7, 4-y argument ; AKA $A3
; §przekazać 5-y, 6-y argument, itd, przez stos (jeśli trzeba)§
LW temp_reg, adress funkcji
JALR temp_reg
```

1.11.4 À propos

À propos, różnica między przekazywaniem argumentów funkcji w x86, x64, fastcall, ARM i MIPS demonstruje, że procesorowi, ogólnie, jest obojętnie w kjaki sposób będą przekazywane parametry funkcji. Można stworzy ć hipotetyczny kompilator, który będzie je przekazywał za pomocą wskaźnika na strukturę z prametrami, nie korzystając ze stosu w ogóle.

Rejestry $A0...$A3 w MIPS są nazwane w ten sposób tylko dla wygody (jest to standard przy wywołaniach 032). Programiści mogą korzystać z jakichkolwiek innych rejestrów (może oprócz $ZERO) do przekazywania danych.

Początkujący programiści w asemblerze przekazują parametry do innych funkcji zwykle przez rejestry, lub nawet przez zmienne globalne. I wszystko to normalnie działa.

1.12 scanf()

Polish text placeholder

1.12.1 Prosty przykład

```
#include <stdio.h>

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

        scanf ("%d", &x);

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

        return 0;
};
```

Teraz używanie `scanf()` do interakcji z użytkownikiem w programach nie jest zbyt popularne, jednak mimo wszystko funkcja ta jest dobrym przykładem użycia wskaźnika na zmienną typu *int*.

O wskaźnikach

Wskaźniki są jedną z podstawowych koncepcji informatycznych. Często tworząc dużą tablicę, strukturę albo objekt jako argument jakiejś funkcji, przekazywanie wskaźnika do argumentu jest mniej pamięciożerne niż gdyby przekazać całą tablicę/strukturę, np. kiedy chcesz wypisać tekst w konsoli, najprościej będzie wskazać jego adres w pamięci.

W dodatku jeśli wywoływana funkcja potrzebuje zmodyfikować cokolwiek w dużej tablicy lub strukturze danych przekazanej jako parametr i zwrócić potem tą tablicę/strukturę, kopiowanie tylu danych byłoby prawie absurdalne. Dlatego najprościej będzie przekazać adres tej tablicy/struktury do wywoływanej funkcji i wtedy zmodyfikować to co wymagało modyfikacji.

Wskaźnik w C/C++—jest adresem pewnego miejsca w pamięci.

W x86, adresy są reprezentowane przy pomocy 32-bitowych liczb (czyli 4 bajtowych), a w x86-64 jako liczby 64-bitowe (czyli 8 bajtowe). Przy okazji jest to powód dlaczego niektórych ludzi oburza przeskok na x86-64- wszystkie wskaźniki w architekturze x64 wymagają dwa razy więcej miejsca, włączając pamięć cache, co jest bardzo ”kosztownym” zużyciem pamięci.

Możliwa jest praca z tylko nietypowanymi wskaźnikami, wymagająca nieco wysiłku, np. funkcja z biblioteki standardowej C- `memcpy()`, która kopiuje blok z jednej lokalizacji w pamięci do innej, jako argumenty przyjmuje 2 wskaźniki typu `void*`, co umożliwia kopiowanie dowolnych typów danych. Typy danych nie są istotne, znaczenie mają tylko rozmiary bloków pamięci.

Wskaźniki są także często używane kiedy funkcja potrzebuje zwrócić więcej niż jedną wartość (wrócę do tego później (**??**)).

Funkcja *scanf()* jest takim przypadkiem.

Oprócz tego faktu, funkcja scanf() wymaga podania w argumencie ile wartości ma wczytać, żeby później móc je zwrócić.

W C/C++ typ wskaźnika jest potrzebny tylko do sprawdzania typów podczas kompilacji.

Wewnątrz skompilowanego kodu nie ma żadnej informacji jakiego typu są wskaźniki.

x86

MSVC

Tutaj znajduje się wynik kompilacji programu w MSVC 2010:

```
CONST    SEGMENT
$SG3831    DB    'Enter X:', 0aH, 00H
$SG3832    DB    '%d', 00H
$SG3833    DB    'You entered %d...', 0aH, 00H
CONST    ENDS
PUBLIC    _main
EXTRN    _scanf:PROC
EXTRN    _printf:PROC
; Function compile flags: /Odtp
_TEXT    SEGMENT
_x$ = -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

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

`x` jest zmienną lokalną.

Według standardu C/C++ zmienna lokalna może być widoczna tylko w konkretnej funkcji. Tradycyjnie zmienne lokalne są przechowywane na stosie. Prawdopodnie są inne moliwości przechowywania tych zmiennych, ale tak akurat jest w x86.

Zadaniem instrukcji rozpoczynającej funkcję, `PUSH ECX`, nie jest zapisanie stanu `ECX` (można zauważyć brak odpowiadającej instrukcji POP ECX na końcu funkcji).

Tak naprawdę instrukcja ta alokuje 4 bajty na stosie do przechowania zmiennej x.

Dostęp do `x` odbywa się z asystującym makrem `_x$` (równym -4) i rejestrem `EBP` rwskazującym bieżącą ramkę.

We fragmencie wykonującym funkcje, `EBP` wskazuje bieżącą Polish text placeholder umożliwiając dostęp do zmiennych lokalnych i argumentów funkcji poprzez `EBP+offset`.

Możliwe jest także użycie `ESP` w takim samym celu, le nie jest to zbyt wygodne, ponieważ wartość tego rejestru często się zmienia. Wartość `EBP` może być postrzegana jako *frozen state* wartości w `ESP` z początku wykonania funkcji.

Tutaj znajduje się typowa ramka stosu w układzie środowiska 32-bitowego:

...	...
EBP-8	zmienna lokalna #2, oznaczony w programie **IDA!** jako `var_8`
EBP-4	zmienna lokalna #1, oznaczony w programie **IDA!** jako `var_4`
EBP	zapisana wartość `EBP`
EBP+4	adres powrotu
EBP+8	argument#1, oznaczony w programie **IDA!** jako `arg_0`
EBP+0xC	argument#2, oznaczony w programie **IDA!** jako `arg_4`
EBP+0x10	argument#3, oznaczony w programie **IDA!** jako `arg_8`
...	...

Funkcja `scanf()` w naszym przykładzie ma dwa argumenty.

Pierwszy jest wskaźnikiem na string `%d` a drugi jest adresem zmiennej `x`.

Na początku adres zmiennej `x` jest ładowany do rejestru `EAX` przy pomocy instrukcji `lea eax, DWORD PTR _x$[ebp]`.

`LEA` oznacza *load effective address* i jest często używana do formowania adresów (**??**).

Można powiedzieć, że w tym przypadku `LEA` po prostu umieszcza sumę rejestru `EBP` i makra `_x$` w rejestrze `EAX`.

To jest to samo co `lea eax, [ebp-4]`.

Więc od rejestru `EBP` jest odejmowane 4 i wynik zostaje umieszczony w rejestrze `EAX`. Następnie wartość rejestru `EAX` jest odkładana na stosie i funkcja `scanf()` zostaje wywołana.

`printf()` wywołuje się z pierwszym argumentem- wskaźnikiem na string: `You entered %d...\n`.

Drugi argument jest przygotowywany za pomocą: `mov ecx, [ebp-4]`. Instrukcja kopiuje zmienną `x` (nie jej adres) do rejestru `ECX`.

Następnie wartość z `ECX` jest odkładana na stos, a na koniec zostaje wywołana funkcja `printf()`.

GCC

Tak wygląda skompilowany kod w GCC 4.4.1 w systemie Linux:

```
main            proc near

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

                push    ebp
                mov     ebp, esp
                and     esp, 0FFFFFFF0h
                sub     esp, 20h
                mov     [esp+20h+var_20], offset aEnterX ; "Enter X:"
                call    _puts
                mov     eax, offset aD  ; "%d"
                lea     edx, [esp+20h+var_4]
                mov     [esp+20h+var_1C], edx
```

```
                mov     [esp+20h+var_20], eax
                call    ___isoc99_scanf
                mov     edx, [esp+20h+var_4]
                mov     eax, offset aYouEnteredD___ ; "You entered %d...\n"
                mov     [esp+20h+var_1C], edx
                mov     [esp+20h+var_20], eax
                call    _printf
                mov     eax, 0
                leave
                retn
main            endp
```

GCC zamienia wywołanie funkcji `printf()` na wywołanie funkcji `puts()`. Powód tego został wyjaśniony w (**??**).

Jak w przykładzie MSVC—argumenty funkcji są umieszczane na stosie przy użyciu instrukcji `MOV`.

By the way

Ten prosty przykład pokazuje jak faktycznie kompilatory tłumaczą listy wyrażeń w C/C++-block na sekwencyjne listy instrukcji. Nie ma nic pomiędzy wyrażeniami w C/C++ a wynikowym kodem maszynowym.

1.12.2 Popularny błąd

Bardzo popularnym błędem jest podanie jako argumentu zmiennej *x* zamiast wskaźnika na zmienną *x*:

```
#include <stdio.h>

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

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

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

        return 0;
};
```

Więc co tutaj się stanie? *x* jest niezainicjalizowany i zawiera losowe śmieci z lokalnego stosu. Kiedy funkcja `scanf()` jest wywoływana, pobiera ciąg znaków od użytkownika, parsuje go do liczby i próbuje go zapisać w *x*, traktując wartość *x* jako adres w pamięci. Jednak skoro tam są losowe śmieci ze stosu, `scanf()` będzie próbować uzyskać dostęp do losowego adresu. Najczęściej wtedy proces zawiesza działanie.

Co ciekawe, niektóre biblioteki **CRT!**w wersji do debugowania, umieszczają w pamięci, która jest alokowana, widoczne wzory takie jak 0xCCCCCCCC albo 0x0BADF00D itp. W tym przypadku, *x* może zawierać 0xCCCCCCCC, więc `scanf()` będzie próbować dokonać zapisu pod adresem 0xCCCCCCCC. Jeśli program wyświetli komunikat, że gdzieś w procesie wystąpi próba zapisu pod adresem 0xCCCCCCCC, to znaczy, że została użyta niezainicjalizowana zmienna (lub wskaźnik). Jest to lepsze rozwiązanie niż gdyby nowo alokowana pamięć była po prostu wyczyszczona.

1.12.3 Zmienne globalne

Co jeśli zmienna `x` z poprzedniego przykładu nie będzie zmienną lokalną ale globalną? Wtedy będzie dostępna z każdego miejsca w programie, nie tylko wewnątrz funkcji. Zmienne globalne są uważane za anti-pattern, ale w celu eksperymentu możemy tak zrobić:

```
#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
```

In this case the `x` variable is defined in the `_DATA` segment and no memory is allocated in the local stack. It is accessed directly, not through the stack. Uninitialized global variables take no space in the executable file (indeed, why one needs to allocate space for variables initially set to zero?), but when someone accesses their address, the **OS!** will allocate a block of zeros there[72].

Now let's explicitly assign a value to the variable:

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

We got:

```
_DATA   SEGMENT
_x      DD      0aH

...
```

Here we see a value `0xA` of DWORD type (DD stands for DWORD = 32 bit) for this variable.

If you open the compiled .exe in **IDA!**, you can see the *x* variable placed at the beginning of the `_DATA` segment, and after it you can see text strings.

If you open the compiled .exe from the previous example in **IDA!**, where the value of *x* hasn't been set, you would see something like this:

[72] That is how a **VM!**[73] behaves

Listing 1.64: **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` is marked with `?` with the rest of the variables that do not need to be initialized. This implies that after loading the .exe to the memory, a space for all these variables is to be allocated and filled with zeros [*ISO/IEC 9899:TC3 (C C99 standard)*, (2007)6.7.8p10]. But in the .exe file these uninitialized variables do not occupy anything. This is convenient for large arrays, for example.

GCC: x86

The picture in Linux is near the same, with the difference that the uninitialized variables are located in the `_bss` segment. In **ELF!**[74] file this segment has the following attributes:

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

If you, however, initialize the variable with some value e.g. 10, it is to be placed in the `_data` segment, which has the following attributes:

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

MSVC: x64

Listing 1.65: 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

_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
```

[74] **ELF!**

The code is almost the same as in x86. Please note that the address of the x variable is passed to `scanf()` using a `LEA` instruction, while the variable's value is passed to the second `printf()` using a `MOV` instruction. `DWORD PTR` —is a part of the assembly language (no relation to the machine code), indicating that the variable data size is 32-bit and the `MOV` instruction has to be encoded accordingly.

ARM: Optymalizujący Keil 6/2013 (tryb Thumb)

Listing 1.66: 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
```

So, the `x` variable is now global and for this reason located in another segment, namely the data segment (*.data*). One could ask, why are the text strings located in the code segment (*.text*) and `x` is located right here? Because it is a variable and by definition its value could change. Moreover it could possibly change often. While text strings has constant type, they will not be changed, so they are located in the *.text* segment.

The code segment might sometimes be located in a **ROM!**[75] chip (keep in mind, we now deal with embedded microelectronics, and memory scarcity is common here), and changeable variables —in **RAM!**[76].

It is not very economical to store constant variables in RAM when you have ROM.

Furthermore, constant variables in RAM must be initialized, because after powering on, the RAM, obviously, contains random information.

Moving forward, we see a pointer to the `x` (`off_2C`) variable in the code segment, and that all operations with the variable occur via this pointer.

That is because the `x` variable could be located somewhere far from this particular code fragment, so its address must be saved somewhere in close proximity to the code.

The `LDR` instruction in Thumb mode can only address variables in a range of 1020 bytes from its location,

[75] **ROM!**
[76] **RAM!**

and in ARM-mode —variables in range of ± 4095 bytes.

And so the address of the `x` variable must be located somewhere in close proximity, because there is no guarantee that the linker would be able to accommodate the variable somewhere nearby the code, it may well be even in an external memory chip!

One more thing: if a variable is declared as *const*, the Keil compiler allocates it in the `.constdata` segment.

Perhaps thereafter, the linker could place this segment in ROM too, along with the code segment.

ARM64

Listing 1.67: Nieoptymalizujący GCC 4.9.1 ARM64

```
1         .comm   x,4,4
2  .LC0:
3         .string "Enter X:"
4  .LC1:
5         .string "%d"
6  .LC2:
7         .string "You entered %d...\n"
8  f5:
9  ; save FP and LR in stack frame:
10        stp     x29, x30, [sp, -16]!
11 ; set stack frame (FP=SP)
12        add     x29, sp, 0
13 ; load pointer to the "Enter X:" string:
14        adrp    x0, .LC0
15        add     x0, x0, :lo12:.LC0
16        bl      puts
17 ; load pointer to the "%d" string:
18        adrp    x0, .LC1
19        add     x0, x0, :lo12:.LC1
20 ; form address of x global variable:
21        adrp    x1, x
22        add     x1, x1, :lo12:x
23        bl      __isoc99_scanf
24 ; form address of x global variable again:
25        adrp    x0, x
26        add     x0, x0, :lo12:x
27 ; load value from memory at this address:
28        ldr     w1, [x0]
29 ; load pointer to the "You entered %d...\n" string:
30        adrp    x0, .LC2
31        add     x0, x0, :lo12:.LC2
32        bl      printf
33 ; return 0
34        mov     w0, 0
35 ; restore FP and LR:
36        ldp     x29, x30, [sp], 16
37        ret
```

In this case the x variable is declared as global and its address is calculated using the `ADRP` / `ADD` instruction pair (lines 21 and 25).

1.12.4 Ćwiczenie

- http://challenges.re/53

1.13 switch()/case/default

1.13.1

```
#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
};
```

Wnioski

listing.**??**.

1.13.2 Ćwiczenia

Ćwiczenie#1

Polish text placeholder

1.14 Loops

1.14.1 Ćwiczenia

- http://challenges.re/54
- http://challenges.re/55
- http://challenges.re/56
- http://challenges.re/57

1.15 More about strings

1.15.1 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

1.16 Replacing arithmetic instructions to other ones

1.16.1 Ćwiczenie

- http://challenges.re/59

1.17 Arrays

1.17.1

```
#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;
};
```

1.17.2

1.17.3 Ćwiczenia

- http://challenges.re/62
- http://challenges.re/63
- http://challenges.re/64
- http://challenges.re/65
- http://challenges.re/66

1.18 Structures

1.18.1 UNIX: struct tm

1.18.2

1.18.3 Ćwiczenia

- http://challenges.re/71
- http://challenges.re/72

1.19

1.19.1

Rozdział 2

Polish text placeholder

Rozdział 3

Rozdział 4

Java

4.1 Java

4.1.1

4.1.2

4.1.3

Rozdział 5

5.1 Linux

5.2 Windows NT

5.2.1 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].

5.3

5.4

Pierre Capillon – Black-box cryptanalysis of home-made encryption algorithms: a practical case study.

How to Hack an Expensive Camera and Not Get Killed by Your Wife.

[1] Dostęp także przez http://go.yurichev.com/17293
[2] Dostęp także przez http://go.yurichev.com/17294

Rozdział 6

Rozdział 7

Książki/blogi warte przeczytania

7.1 Książki i inne materiały

7.1.1 Inżynieria wsteczna

- 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)
- Reginald Wong, *Mastering Reverse Engineering: Re-engineer your ethical hacking skills*, (2018)

(Stara, ale wciąż interesująca) Pavol Cerven, *Crackproof Your Software: Protect Your Software Against Crackers*, (2002).

Oraz książki Krisa Kaspersky'ego.

7.1.2 Windows

- Mark Russinovich, *Microsoft Windows Internals*
- Peter Ferrie – The "Ultimate" Anti-Debugging Reference[1]

Blogi:

- Microsoft: Raymond Chen
- nynaeve.net

7.1.3 C/C++

- Brian W. Kernighan, Dennis M. Ritchie, *The C Programming Language*, 2ed, (1988)
- *ISO/IEC 9899:TC3 (C C99 standard)*, (2007)[2]
- Bjarne Stroustrup, *The C++ Programming Language, 4th Edition*, (2013)
- C++11 standard[3]
- Agner Fog, *Optimizing software in C++* (2015)[4]
- Marshall Cline, *C++ FAQ*[5]
- Dennis Yurichev, *C/C++ programming language notes*[6]

[1] http://pferrie.host22.com/papers/antidebug.pdf
[2] Dostęp także przez http://go.yurichev.com/17274
[3] Dostęp także przez http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2013/n3690.pdf.
[4] Dostęp także przez http://agner.org/optimize/optimizing_cpp.pdf.
[5] Dostęp także przez http://go.yurichev.com/17291
[6] Dostęp także przez http://yurichev.com/C-book.html

- JPL Institutional Coding Standard for the C Programming Language[7]

7.1.4 x86 / x86-64

- manuale Intela[8]
- manuale AMD [9]
- Agner Fog, *The microarchitecture of Intel, AMD and VIA CPUs*, (2016)[10]
- Agner Fog, *Calling conventions* (2015)[11]
- *Intel® 64 and IA-32 Architectures Optimization Reference Manual*, (2014)
- *Software Optimization Guide for AMD Family 16h Processors*, (2013)

Trochę stare, ale wciąż interesujące:

Michael Abrash, *Graphics Programming Black Book*, 1997[12] (znany z pracy nad niskopoziomową optymalizacją w takich projektach jak Windows NT 3.1 i id Quake).

7.1.5 ARM

- manuale ARM[13]
- *ARM(R) Architecture Reference Manual, ARMv7-A and ARMv7-R edition*, (2012)
- [*ARM Architecture Reference Manual, ARMv8, for ARMv8-A architecture profile*, (2013)][14]
- Advanced RISC Machines Ltd, *The ARM Cookbook*, (1994)[15]

7.1.6 Język maszynowy

Richard Blum — Professional Assembly Language.

7.1.7 Java

[Tim Lindholm, Frank Yellin, Gilad Bracha, Alex Buckley, *The Java(R) Virtual Machine Specification / Java SE 7 Edition*] [16].

7.1.8 UNIX

Eric S. Raymond, *The Art of UNIX Programming*, (2003)

7.1.9 Programowanie

- Brian W. Kernighan, Rob Pike, *Practice of Programming*, (1999)
- Henry S. Warren, *Hacker's Delight*, (2002). Niektórzy twierdzą, że sztuczki z tej książki nie mają dzisiaj znaczenia, ponieważ miały zastosowanie wyłącznie w procesorach **RISC!**, gdzie instrukcje typu branch są kosztowne. Niemniej jednak, wszystko to znacząco ułatwia zrozumienie algebry Boole'a i całej matematyki wokół tego.

[7] Dostęp także przez https://yurichev.com/mirrors/C/JPL_Coding_Standard_C.pdf
[8] Dostęp także przez http://www.intel.com/content/www/us/en/processors/architectures-software-developer-manuals.html
[9] Dostęp także przez http://developer.amd.com/resources/developer-guides-manuals/
[10] Dostęp także przez http://agner.org/optimize/microarchitecture.pdf
[11] Dostęp także przez http://www.agner.org/optimize/calling_conventions.pdf
[12] Dostęp także przez https://github.com/jagregory/abrash-black-book
[13] Dostęp także przez http://infocenter.arm.com/help/index.jsp?topic=/com.arm.doc.subset.architecture.reference/index.html
[14] Dostęp także przez http://yurichev.com/mirrors/ARMv8-A_Architecture_Reference_Manual_(Issue_A.a).pdf
[15] Dostęp także przez http://go.yurichev.com/17273
[16] Dostęp także przez https://docs.oracle.com/javase/specs/jvms/se7/jvms7.pdf; http://docs.oracle.com/javase/specs/jvms/se7/html/

7.1.10

- Bruce Schneier, *Applied Cryptography*, (John Wiley & Sons, 1994)
- (Free) lvh, *Crypto 101*[17]
- (Free) Dan Boneh, Victor Shoup, *A Graduate Course in Applied Cryptography*[18].

[17] Dostęp także przez https://www.crypto101.io/
[18] Dostęp także przez https://crypto.stanford.edu/~dabo/cryptobook/

Użyte akronimy

OS System operacyjny (Operating System)

PL Język programowania (Programming Language)

ROM Pamięć tylko do odczytu (Read-Only Memory)

ALU Jednostka arytmetyczno-logiczna (Arithmetic Logic Unit)

LIFO Ostatni na wejściu, pierwszy na wyjściu (Last In First Out)

ABI Interfejs binarny aplikacji (Application Binary Interface)

RA adres powrotu

PE Portable Executable (format plików wykonywalnych w systemach Windows)

SP wskaźnik stosu. SP/ESP/RSP w x86/x64. SP w ARM.

PC Program Counter. IP/EIP/RIP w x86/64. PC w ARM.

LR Link Register

IDA Interaktywny deasembler i debugger rozwijany przez Hex-Rays

MSVC Microsoft Visual C++

AKA Also Known As — znany również jako

CRT C Runtime library

CPU Central Processing Unit

CISC Complex Instruction Set Computing

RISC Reduced Instruction Set Computing

BSS Block Started by Symbol

DBMS Database Management Systems

ISA Instruction Set Architecture (architektura listy rozkazów)

SEH Structured Exception Handling

ELF Polish text placeholder(Executable File Format) Format plików wykonywalnych używany w systemach z rodziny *NIX, w szczególności na Linuksie

NOP No Operation

RAM Random-Access Memory

GCC GNU Compiler Collection

VM Virtual Memory

GPR General Purpose Registers (rejestry ogólnego przeznaczania)

GDB GNU Debugger

FP Frame Pointer

STMFD Store Multiple Full Descending ()

LDMFD Load Multiple Full Descending ()

STMED Store Multiple Empty Descending ()

LDMED Load Multiple Empty Descending ()

STMFA Store Multiple Full Ascending ()

LDMFA Load Multiple Full Ascending ()

STMEA Store Multiple Empty Ascending ()

LDMEA Load Multiple Empty Ascending ()

URL Uniform Resource Locator

Słownik terminów

Polish text placeholder Polish text placeholder. 61

anti-pattern coś powszechnie uznanego jako zła praktyka. 32, 63

callee funkcja wywoływana. 46

caller funkcja wywołująca. 6–8, 29, 46

endianess kolejność bajtów. 21

funkcja liść Funkcja, która nie wywołuje żadnej innej. 28, 32

GiB gibibajt: 2^{10} (1024) mebibajtów, 2^{20} (1048576) kibibajtów lub 2^{30} (1073741824) bajtów. 15

heap (kopiec, sterta) - przeważnie duży kawałek pamięci, zapewniony aplikacji przez **OS!** na jej własne potrzeby. malloc()/free() pracują ze stertą. 30

inkrementować zwiększać o 1. 16

inżynieria wsteczna proces odkrywania jak dana rzecz działa, czasami w celu jej sklonowania. iii

rejestr powrotu (RISC) Rejestr, w który zwykle przechowywany jest adres powrotu. Dzięki temu można wywoływać funkcje-liście (leaf functions) bez używania stosu - a więc szybciej. 32

stdout standardowe wyjście. 21, 35

thunk function prosta funkcja, której jedynym zadaniem jest wywołanie innej funkcji. 22, 41

wskaźnik stosu rejestr pokazujący na miejsce na stosie. 9, 11, 19, 30, 34, 42, 54, 55, 79

Indeks