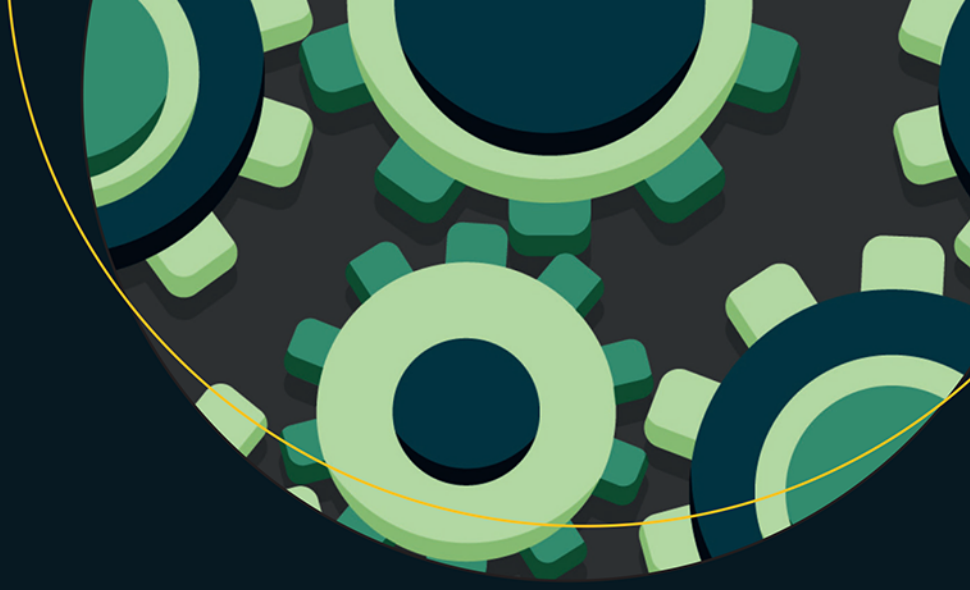

Programowanie w asemblerze x64

Od nowicjusza do znawcy AVX

Jo Van Hoey

Helion

Apress

Tytuł oryginału: Beginning x64 Assembly Programming: From Novice to AVX Professional

Tłumaczenie: Grzegorz Werner

ISBN: 978-83-289-0109-4

First published in English under the title Beginning x64 Assembly Programming; From Novice to AVX Professional by Jo Van Hoey, edition: 1

Helion S.A.
ul. Kościuszki 1c, 44-100 Gliwice
tel. 32 230 98 63
e-mail: *helion@helion.pl*
WWW: *https://helion.pl* (księgarnia internetowa, katalog książek)

Printed in Poland.

Spis treści

ROZDZIAŁ 1.

Twój pierwszy program

Pokolenia programistów rozpoczynały swoją karierę od nauczenia się, jak wyświetlić `witaj, świecie` na ekranie komputera. Jest to tradycja, którą zapoczątkował w latach 70. Brian W. Kernighan w napisanej wspólnie z Dennisem Ritchiem książce *The C Programming Language*. Kernighan opracował język programowania C w laboratoriach Bella. Od tego czasu wiele się zmieniło, ale C nadal jest językiem, który powinien znać każdy szanujący się programista. Większość „nowoczesnych" i „wyrafinowanych" języków programowania ma korzenie w C. Niekiedy C jest nazywany „przenośnym asemblerem", więc jako początkujący programista asemblera, powinieneś go poznać. Aby zadośćuczynić tradycji, zaczniemy od programu w asemblerze, który wyświetli `witaj, świecie` na Twoim ekranie. Na listingu 1.1 pokazano kod źródłowy asemblerowej wersji programu `witaj, świecie`, którą będziemy analizować w tym rozdziale.

Listing 1.1. *hello.asm*

```
; hello.asm
section .data
    msg     db      "witaj, świecie",0
section .bss
section .text
    global main
main:
    mov     rax, 1      ; 1 = wypisz
    mov     rdi, 1      ; 1 = na stdout
    mov     rsi, msg    ; łańcuch do wyświetlenia w rsi
    mov     rdx, 15     ; długość łańcucha, bez 0
    syscall             ; wyświetl łańcuch
    mov     rax, 60     ; 60 = wyjście
    mov     rdi, 0      ; 0 = wyjściowy kod sygnalizujący sukces
    syscall             ; zakończ
```

Edycja, asemblacja, konsolidowanie i uruchamianie (lub debugowanie)

Na rynku jest wiele dobrych edytorów tekstu, zarówno bezpłatnych, jak i komercyjnych. Poszukaj takiego, który obsługuje wyróżnianie składni dla 64-bitowej wersji NASM. Aby korzystać z wyróżniania składni, w większości przypadków będziesz musiał pobrać jakiś dodatek lub pakiet.

- W tej książce będziemy pisać kod przeznaczony dla Netwide Assembler (NASM). Istnieją inne asemblery, takie jak YASM, FASM, GAS i MASM firmy Microsoft. Jak to w świecie komputerów, często toczą się zażarte dyskusje, który jest najlepszy. Postanowiliśmy użyć asemblera NASM, ponieważ jest dostępny w wersjach dla Linuksa, Windows i macOS oraz ma dużą społeczność użytkowników. Podręcznik znajdziesz pod adresem *www.nasm.us*.

Będziemy używać edytora gedit z zainstalowanym plikiem wyróżniania składni asemblera. Gedit to standardowy edytor dostępny w Linuksie; my używamy dystrybucji Ubuntu Desktop 22.04 LTS. Plik wyróżniania składni znajdziesz pod adresem *https://wiki.gnome.org/action/show/Projects/GtkSourceView/LanguageDefinitions*. Pobierz plik *asm-intel.lang* i skopiuj go do katalogu */usr/share/gtksourceview*.0/language-specs/*, zastępując gwiazdkę (*) numerem wersji zainstalowanej w Twoim systemie. Kiedy otworzysz gedit, na dole okna edytora będziesz mógł wybrać język programowania, w tym przypadku Assembler (Intel).

Na naszym ekranie gedit plik *hello.asm* z listingu 1.1 wygląda tak, jak pokazano na rysunku 1.1.

```
; hello.asm
section .data
    msg db      "Witaj, świecie",0
section .bss
section .text
    global main
main:
    mov     rax, 1      ; 1 = wypisz
    mov     rdi, 1      ; 1 = na stdout
    mov     rsi, msg    ; łańcuch do wyświetlenia w rsi
    mov     rdx, 15     ; długość łańcucha, bez 0
    syscall             ; wyświetl łańcuch
    mov     rax, 60     ; 60 = wyjście
    mov     rdi, 0      ; 0 = wyjściowy kod sygnalizujący sukces
    syscall             ; zakończ
```

Rysunek 1.1. *Plik hello.asm w edytorze gedit*

Chyba się zgodzisz, że dzięki wyróżnianiu składni kod staje się nieco bardziej czytelny.

Kiedy piszemy programy w asemblerze, mamy otwarte dwa okna — okno gedit, zawierające kod źródłowy, oraz okno z wierszem poleceń w katalogu projektu. Dzięki temu możemy łatwo przełączać się między edycją plików projektu a manipulowaniem nimi (asemblacją i wykonywaniem programu, debugowaniem itd.). Zgadzamy się, że w bardziej skomplikowanych i większych projektach to za mało; będziesz potrzebował zintegrowanego środowiska

programistycznego (ang. *integrated development environment*, IDE). Ale na razie praca z prostym edytorem tekstu i wierszem poleceń (innymi słowy, z CLI) w zupełności wystarczy. Ma to tę zaletę, że możemy skupić się na asemblerze, a nie na „wodotryskach" oferowanych przez IDE. W późniejszych rozdziałach omówimy przydatne narzędzia i programy użytkowe, niektóre z graficznym interfejsem użytkownika, a inne uruchamiane z poziomu CLI. Jednak opis środowisk IDE wykracza poza ramy niniejszej książki.

W każdym ćwiczeniu używamy oddzielnego katalogu projektowego, który zawiera wszystkie pliki potrzebne w danym projekcie.

Oczywiście, oprócz edytora tekstu będzie potrzebnych kilka innych narzędzi, takich jak `GCC`, `GDB`, `make` i `NASM`. Najpierw sprawdź, czy masz zainstalowany pakiet `GCC`.

`GCC` (skrót od ang. *GNU Compiler Collection*) to domyślny kompilator i linker w Linuksie. (GNU, skrót od **G**NU to **N**ie **U**nix, to rekurencyjny akronim. Używanie rekurencyjnych akronimów jest starym żartem zapoczątkowanym w latach 70. przez programistów Lispa. Kiepskim starym żartem…)

W wierszu poleceń wpisz **`gcc -v`**. `GCC` wyświetli kilka komunikatów, pod warunkiem że jest zainstalowany. Jeśli nie jest, zainstaluj go poniższym poleceniem:

```
sudo apt install gcc
```

Sprawdź też, czy zainstalowane są pakiety `GDB` i `make`, przez wydanie poleceń `gdb -v` i `make -v`. Jeśli nie rozumiesz tych instrukcji, odśwież swoją wiedzę o Linuksie, zanim przejdziesz dalej.

Musisz zainstalować asembler `NASM` i pakiet `build-essential`, który zawiera kilka potrzebnych narzędzi. W Ubuntu Desktop 22.04 możesz to zrobić poleceniem:

```
sudo apt install build-essential nasm
```

W wierszu poleceń wpisz **`nasm -v`**, a `nasm` wyświetli numer wersji, jeśli jest prawidłowo zainstalowany. Po zainstalowaniu opisanych wyżej programów jesteś gotowy do napisania pierwszego programu w asemblerze.

Wpisz program **`witaj, świecie`** z listingu 1.1 w swoim ulubionym edytorze tekstu i zapisz go pod nazwą *hello.asm*. Jak wspomnieliśmy, do zapisania plików tego pierwszego projektu użyj oddzielnego katalogu. Wyjaśnimy każdy wiersz kodu dalej w tym rozdziale; zwróć uwagę na następujące cechy źródłowego kodu asemblera („kodem źródłowym" jest plik *hello.asm* z właśnie wpisanymi przez Ciebie instrukcjami programu):

- Możesz używać znaków tabulacji, spacji i nowego wiersza, żeby zwiększyć czytelność kodu.
- Jeden wiersz zawiera jedną instrukcję.
- Tekst następujący po średniku to komentarz, innymi słowy — wyjaśnienie przeznaczone dla człowieka. Komputery ignorują komentarze.

W edytorze tekstu utwórz kolejny plik zawierający wiersze pokazane na listingu 1.2.

Listing 1.2. *Plik makefile dla programu hello.asm*

```
# makefile dla hello.asm
hello: hello.o
    gcc -o hello hello.o -no-pie
hello.o: hello.asm
    nasm -f elf64 -g -F dwarf hello.asm -l hello.lst
```

Na rysunku 1.2 pokazano, jak wygląda ten plik w edytorze gedit.

```
1 # makefile dla hello.asm
2 hello: hello.o
3         gcc -o hello hello.o -no-pie
4 hello.o: hello.asm
5         nasm -f elf64 -g -F dwarf hello.asm -l hello.lst
```

Rysunek 1.2. *Plik makefile w programie gedit*

Zapisz ten plik pod nazwą *makefile* w tym samym katalogu, co *hello.asm* i wyjdź z edytora.

Plik *makefile* zostanie wykorzystany przez polecenie `make` do zautomatyzowanego zbudowania naszego programu. **Budowanie** programu to między innymi sprawdzanie, czy w kodzie źródłowym nie ma błędów, dodawanie niezbędnych usług systemu operacyjnego i przekształcanie kodu w sekwencję instrukcji czytelnych dla komputera. W tej książce będziemy używać prostych plików *makefile*. Jeśli chciałbyś dowiedzieć się o nich więcej, podręcznik znajdziesz na stronie:

https://www.gnu.org/software/make/manual/make.html

a samouczek jest dostępny pod adresem:

https://www.tutorialspoint.com/makefile/

Aby zrozumieć, co robi plik *makefile*, trzeba czytać go od góry w dół. Oto uproszczone wyjaśnienie: narzędzie `make` pracuje na drzewie zależności. Zauważa, że *hello* zależy od *hello.o*. Następnie zauważa, że *hello.o* zależy od *hello.asm*, a *hello.asm* nie zależy od niczego innego. Narzędzie `make` porównuje daty ostatniej modyfikacji plików *hello.asm* i *hello.o*, a jeśli data modyfikacji *hello.asm* jest późniejsza, wykonuje wiersz po *hello.o*, czyli *hello.asm*. Następnie `make` zaczyna czytać plik *makefile* od nowa i odkrywa, że data modyfikacji *hello.o* jest późniejsza niż data modyfikacji *hello*. Wykonuje więc wiersz po *hello*, czyli *hello.o*.

W ostatnim wierszu naszego pliku *makefile* program NASM jest używany jako asembler. Po opcji `-f` następuje format wyjściowy, w naszym przypadku elf64 (ang. *Executable and Linkable Format for 64-bit* — format plików wykonywalnych i konsolidowalnych dla architektury 64-bitowej). Opcja `-g` oznacza, że chcemy dołączyć informacje na użytek debugowania w formacie określonym po opcji `-F`. Używamy formatu debugowania `dwarf`. Komputerowi maniacy, którzy wymyślili ten format, najwyraźniej lubili *Hobbita* i *Władcę pierścieni* J.J.R. Tolkiena i może dlatego doszli do wniosku, że DWARF (krasnolud) będzie dobrym uzupełnieniem ELF-a… A tak poważnie, DWARF to skrót od ang. ***D**ebug **W**ith **A**rbitrary **R**ecord **F**ormat* (debugowanie z arbitralnym formatem rekordu).

Innym formatem jest STABS, który nie ma nic wspólnego z dźganiem się mieczami w powieściach Tolkiena; nazwa jest skrótem od ang. ***S**ymbol **T**able **S**trings* (łańcuchy tablicy symboli). Nie będziemy używać formatu STABS, żebyś nie zrobił sobie krzywdy.

Opcja `-l` nakazuje programowi NASM wygenerować plik *.lst*. Będziemy używać plików *.lst* do badania wyników asemblacji. NASM utworzy plik obiektowy z rozszerzeniem *.o*. Plik ten zostanie następnie przetworzony przez program konsolidujący, popularnie zwany linkerem.

■ Może się zdarzyć, że NASM wyświetli kilka zagadkowych komunikatów o błędzie i odmówi utworzenia pliku obiektowego. Czasem NASM będzie narzekać tak uporczywie, że doprowadzi Cię na skraj szaleństwa. W takich przypadkach zachowaj spokój, wypij kolejną kawę i przejrzyj swój kod, ponieważ zrobiłeś coś źle. Im dłużej będziesz programować w asemblerze, tym szybciej zaczniesz wyłapywać błędy.

Kiedy w końcu przekonasz NASM, żeby wyprodukował plik obiektowy, kolejnym etapem będzie konsolidacja z wykorzystaniem linkera. Linker przetwarza Twój kod obiektowy i wyszukuje w systemie inne potrzebne pliki, zwykle usługi systemowe albo inne pliki obiektowe. Pliki te są następnie konsolidowane z wygenerowanym kodem obiektowym w celu utworzenia pliku wykonywalnego. Oczywiście, linker wykorzysta każdą możliwą okazję, aby się poskarżyć, że czegoś mu brakuje. W takim przypadku wypij kolejną kawę i sprawdź swój kod źródłowy oraz plik *makefile*.

W naszym przykładzie jako linkera używamy programu GCC (dla Twojej wygody poniżej jeszcze raz zamieszczamy odpowiedni wiersz pliku *makefile*):

```
hello: hello.o
    gcc -o hello hello.o -no-pie
```

Najnowszy linker i kompilator GCC domyślnie generują pliki wykonywalne niezależne od położenia (ang. *position-independent executable*, PIE). Ma to uniemożliwić hakerom zbadanie, jak program używa pamięci, aby wpłynąć na jego wykonywanie. W tym momencie nie będziemy budować plików PIE; bardzo skomplikowałoby to (celowo, ze względów bezpieczeństwa) analizę naszego programu. Dlatego do pliku *makefile* dodaliśmy parametr `-no-pie`.

Aby wstawić komentarz do pliku *makefile*, poprzedź go symbolem krzyżyka (`#`).

```
# makefile dla hello.asm
```

Używamy GCC ze względu na łatwy dostęp do biblioteki standardowej C z poziomu asemblera. Aby ułatwić sobie życie, od czasu do czasu w celu uproszczenia kodu asemblera będziemy używać funkcji C. Warto jednak wiedzieć, że innym popularnym programem konsolidującym w Linuksie jest `ld`, linker GNU.

Jeśli poprzednie akapity były niezrozumiałe, nie przejmuj się — napij się kawy i kontynuuj lekturę; są to informacje uzupełniające, które na tym etapie nie mają większego znaczenia. Pamiętaj tylko, że plik *makefile* jest Twoim sojusznikiem i wykonuje za Ciebie mnóstwo pracy; w tym momencie jedyne, o co musisz się martwić, to to, żeby nie popełnić błędu podczas przepisywania kodu.

W wierszu poleceń przejdź do katalogu, w którym zapisałeś plik *hello.asm* i plik *makefile*. Wpisz **`make`**, aby zasemblować i zbudować program, a następnie wykonaj go przez wpisanie w wierszu poleceń `./hello`. Jeśli zobaczysz komunikat `hello, world` wypisany przed monitem polecenia, to znaczy, że wszystko zadziałało poprawnie. W przeciwnym razie zrobiłeś jakąś literówkę lub inny błąd i musisz sprawdzić kod źródłowy albo plik *makefile*. Uzupełnij kawę w swojej filiżance i wesołego debugowania!

Przykładowy wynik asemblacji i uruchomienia programu pokazano na rysunku 1.3.

```
grzegorz@Ubuntu22:~/x64/rozdzial 01$ make
nasm -f elf64 -g -F dwarf hello.asm -l hello.lst
gcc -o hello hello.o -no-pie
grzegorz@Ubuntu22:~/x64/rozdzial 01$ ./hello
witaj, świeciegrzegorz@Ubuntu22:~/x64/rozdzial 01$
```

Rysunek 1.3. *Wynik wykonania programu witaj, świecie*

Struktura programu w asemblerze

Nasz pierwszy przykład ilustruje podstawową strukturę programu w asemblerze. Oto jego główne części:

- sekcja `.data`,
- sekcja `.bss`,
- sekcja `.text`.

Sekcja .data

W sekcji `.data` deklaruje się i definiuje zainicjalizowane dane w następującym formacie:

<nazwa zmiennej> <typ> <wartość>

Kiedy w sekcji `.data` znajduje się jakaś zmienna, podczas asemblacji i konsolidacji kodu źródłowego przydzielane jest miejsce na tę zmienną. Nazwy zmiennych to nazwy symboliczne, a odwołania do pamięci lub zmiennych mogą zajmować jedną lub więcej lokacji pamięci. Nazwa zmiennej odnosi się do początkowego adresu zmiennej w pamięci.

Nazwy zmiennych zaczynają się od litery, po której następują inne litery, cyfry lub znaki specjalne. Dostępne typy danych wymieniono w tabeli 1.1.

Tabela 1.1. *Typy danych*

Typ	Długość	Nazwa
`db`	8 bitów	bajt
`dw`	16 bitów	słowo
`dd`	32 bity	podwójne słowo
`dq`	64 bity	poczwórne słowo

W przykładowym programie sekcja `.data` zawiera jedną zmienną, `msg`, która jest nazwą symboliczną wskazującą adres pamięci zajmowany przez `'h'`, pierwszy bajt łańcucha `"Witaj, świecie",0`. Zatem `msg` wskazuje literę `'h'`, `msg+1` wskazuje literę `'e'` itd. Taką zmienną nazywa się **łańcuchem**, czyli ciągłą listą znaków. Łańcuch jest „listą” lub „tablicą” znaków w pamięci. W rzeczywistości każdą ciągłą listę w pamięci można uważać za łańcuch; znaki mogą być czytelne dla człowieka albo nie, a łańcuch może, ale nie musi mieć znaczenia dla ludzi.

Łańcuch czytelny dla człowieka powinien być zakończony zerem. Kiedy pomijasz końcowe zero, robisz to na własne ryzyko. Końcowe zero, o którym tu mowa, nie jest cyfrą 0 w kodzie

ASCII; jest to liczbowe zero, co oznacza, że przechowująca je komórka pamięci zawiera 8 bitów o wartości zero. Jeśli nigdy nie słyszałeś o kodzie ASCII, zajrzyj do Google'a. Znajomość tego kodu jest ważna dla programisty. Oto krótkie wyjaśnienie: znakom używanym przez ludzi przypisano specjalne kody komputerowe. Wielka litera `A` ma kod 65, `B` ma kod 66 itd. Znak nowego wiersza ma kod 10, a znak `NULL` ma kod 0. Zatem łańcuch kończymy znakiem `NULL`. Jeśli w wierszu poleceń wpiszesz `man ascii`, Linux pokaże Ci tabelę kodów ASCII.

Sekcja `.data` może również zawierać stałe, czyli wartości, które nie mogą zostać zmienione przez program. Deklaruje się je w następujący sposób:

```
<nazwa stałej>    equ    <wartość>
```

Oto przykład:

```
pi equ 3.1416
```

Sekcja .bss

Akronim `bss` oznacza *Block Started by Symbol* (blok rozpoczęty przez symbol), a jego historia sięga lat 50., kiedy deklaracja ta była częścią asemblera opracowanego dla komputera IBM 704. W sekcji `bss` umieszcza się niezainicjalizowane zmienne w następującym formacie:

```
<nazwa zmiennej>    <typ>    <liczba>
```

Dostępne typy danych `bss` wymieniono w tabeli 1.2.

Tabela 1.2. *Typy danych bss*

Typ	Długość	Nazwa
`resb`	8 bitów	bajt
`resw`	16 bitów	słowo
`resd`	32 bity	podwójne słowo
`resq`	64 bity	poczwórne słowo

Na przykład poniższa instrukcja deklaruje miejsce na tablicę 20 podwójnych słów:

```
dArray resd 20
```

Zmienne w sekcji `.bss` nie mają żadnych wartości; wartości zostaną im przypisane później, po uruchomieniu programu. Pamięć nie jest rezerwowana w czasie asemblacji, ale w czasie wykonania. Zastosowanie sekcji `.bss` pokażemy w późniejszych przykładach. Kiedy Twój program zaczyna się wykonywać, prosi system operacyjny o przydzielenie miejsca na zmienne z sekcji `.bss` i wstępne wypełnienie go zerami. Jeśli w czasie wykonania w pamięci zabraknie miejsca na zmienne `.bss`, program przedwcześnie zakończy działanie.

Sekcja .txt

W sekcji `.txt` rozgrywa się cała akcja. Sekcja ta zawiera kod programu i zaczyna się od następującej deklaracji:

```
    global main
main:
```

Część `main:` jest nazywana etykietą. Kiedy etykieta znajduje się w oddzielnym wierszu i nie następuje po niej żaden tekst, dodaj do niej dwukropek; w przeciwnym razie asembler wyświetli ostrzeżenie. A nie należy ignorować ostrzeżeń! Kiedy po etykiecie następują inne instrukcje, dwukropek nie jest potrzebny, ale lepiej wyrobić sobie nawyk kończenia wszystkich etykiet dwukropkiem. Dzięki temu Twój kod będzie bardziej czytelny.

W kodzie *hello.asm* po etykiecie `main:` przygotowujemy rejestry, takie jak `rdi`, `rsi` i `rax`, do wypisania komunikatu na ekranie. Więcej informacji o rejestrach znajdziesz w rozdziale 2. W tym przykładzie wyświetlamy łańcuch na ekranie za pomocą wywołania systemowego, czyli prosimy system operacyjny, aby zrobił za nas całą robotę.

- Kod wywołania systemowego 1, który oznacza „zapis”, umieszczamy w rejestrze `rax`.
- Aby umieścić jakąś wartość w rejestrze, używamy instrukcji `mov`. W rzeczywistości instrukcja ta niczego nie przenosi (ang. *move* — przenieść), ale kopiuje źródło do celu. Jej format to:

  `mov` *`cel`*`,` *`źródło`*
- Instrukcji `mov` można używać w następujący sposób:
  - `mov` rejestr, wartość bezpośrednia,
  - `mov` rejestr, pamięć,
  - `mov` pamięć, rejestr,
  - **operacja nielegalna**: `mov` pamięć, pamięć.
- W naszym kodzie docelowe miejsce zapisu określamy w rejestrze `rdi`, a wartość 1 reprezentuje standardowe wyjście (w tym przypadku oznacza to wypisanie danych na ekranie).
- Adres łańcucha, który ma zostać wyświetlony, umieszczamy w rejestrze `rsi`.
- W rejestrze `rdx` umieszczamy długość komunikatu. Policz znaki w łańcuchu `witaj, świecie`[1]. Nie licz znaków cudzysłowu ani końcowego zera. Jeśli policzysz końcowe zero, program spróbuje wyświetlić bajt `NULL`, co nie ma większego sensu.
- Następnie wykonujemy wywołanie systemowe, `syscall`, a łańcuch `msg` jest wypisywany na standardowym wyjściu. Instrukcja `syscall` oznacza wywołanie funkcji systemu operacyjnego.
- Aby uniknąć komunikatu o błędzie po zakończeniu działania programu, trzeba „czysto” wyjść z programu. Zaczynamy od zapisania wartości `60` (wskazującej „wyjście”)

[1] Polskie znaki w standardowym kodowaniu UTF-8 zajmują 2 bajty, więc każdy z nich trzeba policzyć dwukrotnie; dlatego w przykładowym programie ustawiamy liczbę znaków na 15, a nie 14 — *przyp. tłum.*

w rejestrze `rax`. Wyjściowy kod `0` (oznaczający „sukces”) umieszczamy w rejestrze `rdi`, po czym wykonujemy wywołanie systemowe. Dzięki temu program kończy działanie bez skarżenia się na błędy.

Wywołania systemowe nakazują systemowi wykonanie określonych operacji. Każdy system operacyjny ma inną listę wywołań systemowych, a wywołania w Linuksie różnią się od tych używanych przez Windows lub macOS. W tej książce używamy wywołań systemowych Linuksa dla architektury x64; szczegóły znajdziesz pod adresem *http://blog.rchapman.org/posts/Linux_System_Call_Table_for_x86_64/*.

Pamiętaj, że 32-bitowe wywołania systemowe różnią się od 64-bitowych. Kiedy czytasz kod, zawsze sprawdź, czy napisano go dla systemu 32-bitowego czy 64-bitowego.

Wróć teraz do wiersza poleceń systemu operacyjnego i poszukaj pliku *hello.lst*. Plik ten został wygenerowany podczas asemblacji, przed konsolidacją, jak określono w pliku *makefile*. Otwórz plik w edytorze, a zobaczysz listing swojego kodu; w lewej kolumnie znajdują się względne adresy, a w następnej kolumnie — kod przetłumaczony na język maszynowy (w notacji szesnastkowej). Zawartość pliku *hello.lst* pokazano na rysunku 1.4.

```
1     1                                  ; hello.asm
2     2                                  section .data
3     3 00000000 776974616A2C20C59B-         msg db      "witaj, świecie",0
4     3 00000009 77696563696500
5     4                                  section .bss
6     5                                  section .text
7     6                                      global main
8     7                                  main:
9     8 00000000 B801000000                  mov     rax, 1      ; 1 = wypisz
10    9 00000005 BF01000000                  mov     rdi, 1      ; 1 = na stdout
11   10 0000000A 48BE-                       mov     rsi, msg    ; łańcuch do wyświetlenia w rsi
12   10 0000000C [0000000000000000]
13   11 00000014 BA0F000000                  mov     rdx, 15     ; długość łańcucha, bez 0
14   12 00000019 0F05                        syscall             ; wyświetl łańcuch
15   13 0000001B B83C000000                  mov     rax, 60     ; 60 = wyjście
16   14 00000020 BF00000000                  mov     rdi, 0      ; 0 = wyjściowy kod sygnalizujący sukces
17   15 00000025 0F05                        syscall             ; zakończ
18   16
```

Rysunek 1.4. *Plik hello.lst*

Po kolumnie z numerami wierszy następuje kolumna z ośmioma cyframi. Kolumna ta reprezentuje lokacje pamięci. Kiedy asembler budował plik obiektowy, nie było jeszcze wiadomo, jakie zostaną użyte lokacje pamięci. Dlatego poszczególne sekcje zaczynają się od lokacji 0. Sekcji `.bss` nie przydzielono pamięci.

W drugiej kolumnie widzimy wyniki konwersji instrukcji asemblera na kod szesnastkowy. Na przykład instrukcja `mov rax` została przekształcona w `B8`, a `mov rdi` — w `BF`. Są to szesnastkowe reprezentacje instrukcji maszynowych. Zwróć też uwagę na konwersję łańcucha `msg` na szesnastkowe kody ASCII. (Później dowiesz się więcej o notacji szesnastkowej). Pierwsza instrukcja do wykonania zaczyna się od adresu `00000000` i zajmuje pięć bajtów: `B8 01 00 00 00`. Podwójne zera służą do wypełniania i wyrównywania pamięci. Wyrównywanie pamięci to funkcja używana przez asemblery i kompilatory do optymalizacji kodu. Możesz podawać asemblerom i kompilatorom różne flagi, aby uzyskać jak najkrótszy kod, jak najszybszy kod albo jakiś kompromis między jednym a drugim. W dalszych rozdziałach omówimy optymalizację kodu pod kątem przyspieszania jego działania.

Następna instrukcja zaczyna się od adresu `00000005` itd. Adresy pamięci składają się z 8 cyfr (tzn. 8 bajtów); każdy bajt liczy 8 bitów. Zatem adresy mają 64 bity; rzeczywiście, używamy asemblera 64-bitowego. Przyjrzyj się też odwołaniu do zmiennej `msg`. Ponieważ adres zmiennej `msg` nie jest jeszcze znany, odwołanie to ma postać `[0000000000000000]`.

Zgodzisz się zapewne, że mnemoniki asemblera i symboliczne nazwy adresów pamięci są znacznie łatwiejsze do zapamiętania niż wartości szesnastkowe — istnieją setki mnemoników z wieloma różnymi operandami, z których każdy przekłada się na jeszcze więcej szesnastkowych instrukcji. We wczesnej erze komputerów programiści używali języka maszynowego, języka programowania pierwszej generacji. Asembler, z jego „łatwiejszymi do zapamiętania" mnemonikami, jest językiem programowania drugiej generacji.

Podsumowanie

W tym rozdziale omówiliśmy następujące zagadnienia:

- Podstawowa struktura programu w asemblerze, z podziałem na różne sekcje
- Pamięć, z symbolicznymi nazwami adresów
- Rejestry
- Instrukcja asemblera: `mov`
- Jak używać wywołania systemowego (`syscall`)
- Różnica między kodem maszynowym a kodem asemblera

Skorowidz

P

R

S

T

V

W

Z

Chcesz zrozumieć, jak naprawdę działa procesor? Naucz się asemblera!

Nauka programowania w asemblerze bywa frustrująca. Język ten jest trudny i nie wybacza błędów. Jednak przebrnięcie przez te niedogodności przyniesie Ci wiele korzyści. Zdobędziesz bezcenną wiedzę o działaniu procesora, zyskasz też skuteczne narzędzie do badania złośliwego oprogramowania. Staniesz się o wiele lepszym programistą, a wiedza o instrukcjach AVX pozwoli Ci na spektakularne optymalizowanie kodu napisanego w językach wyższego poziomu.

Z tą książką stopniowo nauczysz się prostego, podstawowego kodu, a potem bardziej złożonych instrukcji AVX. Nabierzesz wprawy w czytaniu kodu asemblera i zaczniesz łączyć go z kodem w językach wyższego poziomu. Co ważniejsze, teorię ograniczono tu do niezbędnego minimum, za to dokładnie opisano dostępne narzędzia, omówiono sposób ich użytkowania i możliwe problemy. Kod natomiast został zaprezentowany w postaci kompletnych programów asemblera, co pozwoli Ci na dowolne testowanie, zmienianie i inne eksperymenty. W ten sposób przygotujesz się do samodzielnego badania różnych obszarów AVX i korzystania z oficjalnych podręczników Intela.

Dzięki książce:

- zrozumiesz, jak działa procesor i na czym polega praca systemu operacyjnego
- dowiesz się, jak kompilatory generują kod maszynowy
- poznasz skuteczniejsze sposoby poprawiania swoich programów
- nauczysz się uruchamiania programów w asemblerze
- zaczniesz badać złośliwe oprogramowanie i podejmować niezbędne działania

Jo Van Hoey — jest emerytowanym inżynierem informatyki. Przez 40 lat pracował w branży IT, na różnych stanowiskach; w IBM zajmował się oprogramowaniem mainframe. Od zawsze interesował się bezpieczeństwem IT i zastosowaniem asemblera do zabezpieczania infrastruktury IT przed atakami i złośliwym oprogramowaniem.

Helion
helion.pl
HELION SA
ul. Kościuszki 1c
44-100 Gliwice
tel.: 32 230 98 63
helion@helion.pl
KOD KORZYŚCI
Sięgnij po więcej!
ISBN 978-83-289-0109-4
9 788328 901094
Cena: 77,00 zł
Apress