

## **Programowanie niskopoziomowe**

### **Lab 1 – Wstęp do programowanie niskopoziomowego w NASM (Assembly x86)**

#### **Spis treści**

1. Wprowadzenie teoretyczne .....	2
1.1 Czym jest język asemblerowy? .....	2
1.2 Rejestry procesora .....	3
1.3 Podstawowe instrukcje NASM.....	4
1.4 Instrukcje warunkowe.....	5
1.4 Definiowanie danych i rezerwowanie przestrzeni.....	7
1.5 Wywołania systemowe (syscalls) i przerwania .....	8
1.6 Struktura programu w NASM .....	11
1.7 Rejestry w architekturze x86-64 – short intro .....	12
2. Przygotowanie środowiska .....	15
3. Zadania praktyczne.....	17
Część Fabularna .....	17
Przykłady i zadania .....	20
Przykład 1 – Hello World .....	20
Przykład 2 – Podstawowe dodawanie dwóch liczb .....	22
Zadanie 1 – Dodawanie dwóch liczb z wyświetlaniem wyniku .....	23
Zadanie 2 – Dodawanie dwóch liczb wpisanych przez użytkownika z wyświetlaniem wyniku .....	24
Przykład 3 – Integracja z biblioteką C .....	25
Przykład 4 – Wywołanie Assembly NASM w kodzie C .....	26
Zadanie 3 – Wypisanie liczb od 1 do 9 przy użyciu pętli .....	27
Zadanie 4 – Wyświetlanie trójkąta .....	28
Zadanie 5 – Potęgowanie.....	30
Zadania dodatkowe dla chętnych .....	31
Zadanie 6 – Obliczanie średniej z argumentów .....	31
Zadanie 7 – Rekurencyjna funkcja obliczająca silnię .....	31
Zadanie 8 – Fibbonaci.....	31

# 1. Wprowadzenie teoretyczne

## 1.1 Czym jest język asemblerowy?

Asembler (Assembly) jest językiem programowania niskiego poziomu, który umożliwia bezpośrednią komunikację z procesorem komputera. W przeciwieństwie do języków wysokiego poziomu (np. Python, Java czy C++), które są niezależne od sprzętu i kompilowane lub interpretowane do kodu maszynowego przez dodatkowe oprogramowanie, asembler jest ścisłe związany z architekturą konkretnego procesora (np. x86).

Kod asemblerowy zapisuje się za pomocą mnemoników (skrótnych oznaczeń instrukcji procesora), co zapewnia programistę pełną kontrolę nad operacjami sprzętowymi, takimi jak przenoszenie danych, operacje arytmetyczne, zarządzanie pamięcią oraz wykonywanie skoków warunkowych i bezwarunkowych. Dzięki temu programista może efektywnie zarządzać zasobami komputera oraz optymalizować programy pod kątem szybkości działania i zużycia pamięci.

Programowanie w asemblerze wymaga od programisty dobrej znajomości architektury komputera, szczególnie struktury rejestrów, sposobu adresowania pamięci oraz systemów liczbowych, takich jak binarny i heksadecymalny. Ze względu na złożoność i specyfikę tego języka, asembler najczęściej wykorzystuje się do pisania kodu systemowego, sterowników sprzętu, oprogramowania wbudowanego (embedded), optymalizacji wydajności oraz w dziedzinach, gdzie bezpośredni dostęp do sprzętu ma kluczowe znaczenie, takich jak cyberbezpieczeństwo czy inżynieria odwrotna (reverse engineering).

Podczas tych zajęć będziemy korzystać z NASM (Netwide Assembler), jednego z popularniejszych asemblerów dla architektury x86, charakteryzującego się przejrzystą składnią, szerokimi możliwościami oraz kompatybilnością z różnymi platformami i systemami operacyjnymi, takimi jak Linux, macOS czy Windows.

Manual NASM można znaleźć [tutaj](#).

## 1.2 Rejestry procesora

Operacje procesora polegają głównie na przetwarzaniu danych. Dane te mogą być przechowywane w pamięci i stamtąd odczytywane. Jednak odczyt danych z pamięci oraz ich zapisywanie spowalniają działanie procesora, ponieważ wiąże się to ze skomplikowanym procesem wysyłania żądania danych przez magistralę sterującą do jednostki pamięci i odbierania danych tą samą drogą. Aby przyspieszyć operacje procesora, zawiera on wewnętrzne obszary pamięci, zwane **rejestrami**. Rejestry przechowują elementy danych do przetwarzania bez konieczności dostępu do pamięci. W procesorze znajduje się ograniczona liczba rejestrów, które są wbudowane w jego układ scalony.

Rejestry to szybkie pamięci wewnętrz procesora, wykorzystywane do przechowywania danych tymczasowych:

### Rejestry ogólnego zastosowania:

- **EAX (Accumulator Register)** – używany do operacji arytmetycznych i logicznych, często przechowuje wyniki obliczeń.
- **EBX (Base Register)** – często wykorzystywany jako wskaźnik bazowy w dostępie do pamięci.
- **ECX (Counter Register)** – stosowany w pętlach i operacjach powtarzeniowych jako licznik iteracji.
- **EDX (Data Register)** – używany w operacjach mnożenia i dzielenia, przechowuje dodatkowe dane wynikowe.

### Rejestry specjalne (operacje na pamięci):

- **ESI (Source Index)** – wskaźnik źródła danych w operacjach kopiowania i przetwarzania łańcuchów.
- **EDI (Destination Index)** – wskaźnik miejsca docelowego w operacjach kopiowania danych.
- **ESP (Stack Pointer)** – wskaźnik stosu, przechowuje adres szczytu stosu (używany do zarządzania wywołaniami funkcji).
- **EBP (Base Pointer)** – wskaźnik bazowy stosu, często używany do dostępu do argumentów i zmiennych lokalnych funkcji.

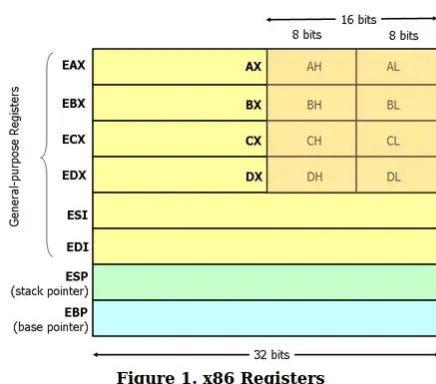


Figure 1. x86 Registers

Więcej informacji o rejestrach znajduje się [tutaj](#).

## 1.3 Podstawowe instrukcje NASM

Podstawowe instrukcje języka NASM:

- **mov** – przenosi wartość między rejestrami lub pamięcią. Przykład: mov eax, 5 umieszcza liczbę 5 w rejestrze eax.
- **and** – wykonuje logiczne „i” na bitach dwóch wartości. Przykład: and eax, ebx ustawia bity w eax, które są ustawione zarówno w eax, jak i ebx.
- **or** – wykonuje logiczne „lub” na bitach dwóch wartości. Przykład: or eax, ebx ustawia bity w eax, które są ustawione w eax lub w ebx.
- **xor** – wykonuje logiczne „wykluczające lub” na bitach dwóch wartości. Przykład: xor eax, eax zeruje rejestr eax.
- **add** – wykonuje dodawanie dwóch wartości. Przykład: add eax, ebx dodaje wartość rejestrze ebx do wartości w rejestrze eax.
- **sub** – odejmuje jedną wartość od drugiej. Przykład: sub eax, 2 odejmuje 2 od wartości rejestrze eax.
- **inc** – zwiększa wartość rejestrze lub pamięci o 1. Przykład: inc eax zwiększa wartość w eax o jeden.
- **dec** – zmniejsza wartość rejestrze lub pamięci o 1. Przykład: dec eax zmniejsza wartość w eax o jeden.
- **syscall** – wywołuje funkcję systemową w systemach operacyjnych typu Unix/Linux. Przykład: syscall
- **db** – definiuje bajty danych w pamięci. Przykład: db "Hello", 0.
- **mul** – mnoży wartości. Przykład: mul ebx mnoży wartość rejestrze eax przez wartość rejestrze ebx i wynik umieszcza w eax.
- **div** – dzieli wartości. Przykład: div ebx dzieli wartość rejestrze eax przez wartość rejestrze ebx.
- **jmp** – wykonuje skok do wskazanego miejsca w kodzie, np. do etykiety. Przykład: jmp start.

Instruction	Description
mov <i>x,y</i>	$x \leftarrow y$
and <i>x,y</i>	$x \leftarrow x \wedge y$
or <i>x,y</i>	$x \leftarrow x \vee y$
xor <i>x,y</i>	$x \leftarrow x \oplus y$
add <i>x,y</i>	$x \leftarrow x + y$
sub <i>x,y</i>	$x \leftarrow x - y$
inc <i>x</i>	$x \leftarrow x + 1$
dec <i>x</i>	$x \leftarrow x - 1$
syscall <i>n</i>	Invoke operating system routine <i>n</i>
db	A <a href="#">pseudo-instruction</a> that declares bytes that will be in memory when the program runs

## 1.4 Instrukcje warunkowe

### Notatki: Instrukcje warunkowe w asemblerze

#### Flagi statusu procesora

Po wykonaniu instrukcji arytmetycznej lub logicznej (np. cmp), procesor ustawia lub kasuje odpowiednie **bity flag w rejestrze rflags**. Najważniejsze flagi to:

- **S (sign)** – ustawiona, jeśli wynik operacji jest ujemny.
- **Z (zero)** – ustawiona, jeśli wynik operacji jest równy zero.
- **C (carry)** – ustawiona, jeśli wystąpiło przeniesienie lub pożyczka (np. w dodawaniu, odejmowaniu).
- **O (overflow)** – ustawiona, jeśli wystąpiło przepiętnienie operacji.

#### Instrukcje warunkowe

Po wykonaniu operacji arytmetycznej lub logicznej można wykonać skok (jump), przeniesienie (move) lub ustawienie wartości (set) na podstawie stanu flag.

#### Rodzaje instrukcji warunkowych

Instrukcje warunkowe w asemblerze występują w trzech głównych formach:

- **j** – skok warunkowy (jump)
- **cmov** – przeniesienie warunkowe (conditional move)
- **set** – ustawienie wartości warunkowo

#### Sufiksy instrukcji warunkowych

Instrukcje warunkowe wykorzystują **30 różnych sufiksów**, które określają warunek działania.

Przykłady:

- **Flagi znaku i zera:** s (negative), ns (non-negative), z (zero), nz (non-zero)
- **Flagi przeniesienia i przepiętnienia:** c (carry), nc (no carry), o (overflow), no (no overflow)
- **Flagi parzystości:** p (parity), np (no parity)
- **Porównania liczbowe:**
  - e (equal), ne (not equal)
  - l (less than), le (less or equal)
  - g (greater than), ge (greater or equal)
  - b (below), be (below or equal)
  - a (above), ae (above or equal)

## Zastosowanie instrukcji warunkowych

Instrukcje warunkowe pozwalają na:

- Realizację pętli (jz, jnz)
- Optymalizację warunkowych operacji (cmov, set)
- Obsługę wyjątków i błędów w programach assemblerowych

# Conditional Instructions

After an arithmetic or logic instruction, or the compare instruction, `cmp`, the processor sets or clears bits in its `rflags`. The most interesting flags are:

- `s` (sign)
- `z` (zero)
- `c` (carry)
- `o` (overflow)

So after doing, say, an addition instruction, we can perform a jump, move, or set, based on the new flag settings. For example:

Instruction	Description
<code>jz L</code>	Jump to label <code>L</code> if the result of the operation was zero
<code>cmovno x, y</code>	$x \leftarrow y$ if the last operation did not overflow
<code>setc x</code>	$x \leftarrow 1$ if the last operation had a carry, but $x \leftarrow 0$ otherwise ( <code>x</code> must be a byte-size register or memory location)

The conditional instructions have three base forms: `j` for conditional jump, `cmov` for conditional move, and `set` for conditional set. The suffix of the instruction has one of the 30 forms: `s ns z nz c nc o no p np pe po e ne l nl le nle g ng ge nge a na ae nae b nb be nbe`.

## 1.4 Definiowanie danych i rezerwowanie przestrzeni

W programowaniu oraz zarządzaniu pamięcią operacyjną, **definiowanie danych i rezerwowanie przestrzeni** oznacza alokację określonej ilości pamięci dla zmiennych, struktur danych lub buforów, które mają być używane w trakcie działania programu.

### 1. Definiowanie danych

Definiowanie danych polega na określeniu typu oraz wartości zmiennej lub struktury, które będą przechowywane w pamięci. W zależności od języka programowania, możemy to zrobić na różne sposoby:

```
db    0x55          ; just the byte 0x55
db    0x55,0x56,0x57 ; three bytes in succession
db    'a',0x55       ; character constants are OK
db    'hello',13,10,'$' ; so are string constants
dw    0x1234         ; 0x34 0x12
dw    'a'           ; 0x61 0x00 (it's just a number)
dw    'ab'          ; 0x61 0x62 (character constant)
dw    'abc'          ; 0x61 0x62 0x63 0x00 (string)
dd    0x12345678     ; 0x78 0x56 0x34 0x12
dd    1.234567e20    ; floating-point constant
dq    0x123456789abcdef0 ; eight byte constant
dq    1.234567e20     ; double-precision float
dt    1.234567e20     ; extended-precision float
```

Aby zarezerwować przestrzeń w pamięci bez jej inicjalizacji, można użyć następujących pseudo-instrukcji w asemblerze. Powinny one zostać umieszczone w sekcji .bss, ponieważ ich użycie w sekcji .text spowoduje błąd kompilacji.

```
buffer:      resb   64          ; reserve 64 bytes
wordvar:     resw   1           ; reserve a word
realarray:   resq   10          ; array of ten reals
```

## 1.5 Wywołania systemowe (syscalls) i przerwania

W nowoczesnych systemach operacyjnych aplikacje użytkownika działają w tzw. **przestrzeni użytkownika (user space)**, natomiast jądro systemu (kernel) operuje w **przestrzeni jądra (kernel space)**. Bezpośredni dostęp aplikacji do zasobów systemu, takich jak pamięć, urządzenia wejścia/wyjścia (klawiatura, ekran, pliki, sieć), jest ograniczony ze względów bezpieczeństwa i stabilności systemu. Aby programy mogły korzystać z tych zasobów, muszą komunikować się z jądrem systemu poprzez **wywołania systemowe (syscalls)**. Jest to specjalny mechanizm, który pozwala aplikacjom użytkownika wykonywać operacje systemowe bez uzyskania pełnego dostępu do przestrzeni jądra.

Wywołania systemowe są realizowane za pomocą:

- **Przerwań (int 0x80)** – stosowanych w architekturze **x86 (32-bit)**.
- **Instrukcji syscall** – stosowanej w architekturze **x86-64 (64-bit)**.

### Przerwania w systemie Linux (x86, 32-bit)

Przerwania (ang. **interrupts**) to sygnały przesyłane do procesora, które mogą zostać wygenerowane zarówno przez sprzęt (np. klawiaturę, mysz, zegar systemowy), jak i przez oprogramowanie. W przypadku systemu Linux, programy mogą wywoływać funkcje jądra poprzez **przerwanie int 0x80**, które przekazuje sterowanie do jądra systemu operacyjnego.

W **32-bitowej architekturze x86**, rejestry mają następujące role podczas wywołań systemowych:

#### Rejestr Zastosowanie

eax	Numer wywołania systemowego
ebx	Pierwszy argument wywołania systemowego
ecx	Drugi argument
edx	Trzeci argument
esi	Czwarty argument (opcjonalnie)
edi	Piąty argument (opcjonalnie)

## Wywołania systemowe (syscalls) w systemie Linux (x86-64, 64-bit)

W nowoczesnych systemach operacyjnych opartych na **architekturze x86-64**, przerwania int 0x80 zostały zastąpione przez bardziej wydajne **instrukcje syscall**. Wywołania systemowe w **x86-64** korzystają z innych rejestrów do przekazywania argumentów.

### Rejestry używane w syscall (x86-64)

#### Rejestr Zastosowanie

rax	Numer wywołania systemowego
rdi	Pierwszy argument
rsi	Drugi argument
rdx	Trzeci argument
r10	Czwarty argument
r8	Piąty argument
r9	Szósty argument

### Przykładowe wywołanie systemowe (sys\_write) przy użyciu syscall (x86-64)

```
section .data
msg db "Hello, world!", 0xa ; Tekst do wypisania
len equ $ - msg ; Obliczenie długości

section .text
global _start

_start:
    mov rax, 1 ; Numer syscall dla sys_write
    mov rdi, 1 ; Deskryptor pliku (1 - stdout)
    mov rsi, msg ; Adres wiadomości do wypisania
    mov rdx, len ; Długość wiadomości
    syscall ; Wywołanie systemowe

    mov rax, 60 ; Numer syscall dla sys_exit
    xor rdi, rdi ; Kod powrotu (0)
    syscall ; Zakończenie programu
```

## Porównanie int 0x80 i syscall

Cechy	int 0x80 (x86, 32-bit)	syscall (x86-64, 64-bit)
Mechanizm wywołania	Przerwanie (int 0x80)	Instrukcja syscall
Numer syscall	eax	rax
Rejestry argumentów	ebx, ecx, edx, esi, edi	rdi, rsi, rdx, r10, r8, r9
Wydajność	Wolniejszy	Szybszy

## Dlaczego syscall jest szybszy?

- int 0x80 wymaga przełączenia kontekstu i wykonania większej liczby operacji w jądrze systemu.
- syscall jest zoptymalizowaną instrukcją procesora, która szybciej przechodzi do przestrzeni jądra.

## Lista wybranych wywołań systemowych w Linux

### Syscall    Numer (x86) Numer (x86-64) Opis

sys_exit	1	60	Zakończenie procesu
sys_write	4	1	Wypisanie tekstu na ekran
sys_read	3	0	Odczyt wejścia użytkownika
sys_open	5	2	Otwarcie pliku
sys_close	6	3	Zamknięcie pliku

Piętna lista wywołań systemowych dostępna jest w plikach:

- /usr/include/asm/unistd\_32.h (dla x86, 32-bit)
- /usr/include/asm/unistd\_64.h (dla x86-64, 64-bit)

Dodatkowo warto zobaczyć:

<https://chromium.googlesource.com/chromiumos/docs/+/master/constants/syscalls.md>

<https://linux-kernel-labs.github.io/refs/heads/master/lectures/interrupts.html>

## 1.6 Struktura programu w NASM

Typowe sekcje w programie NASM:

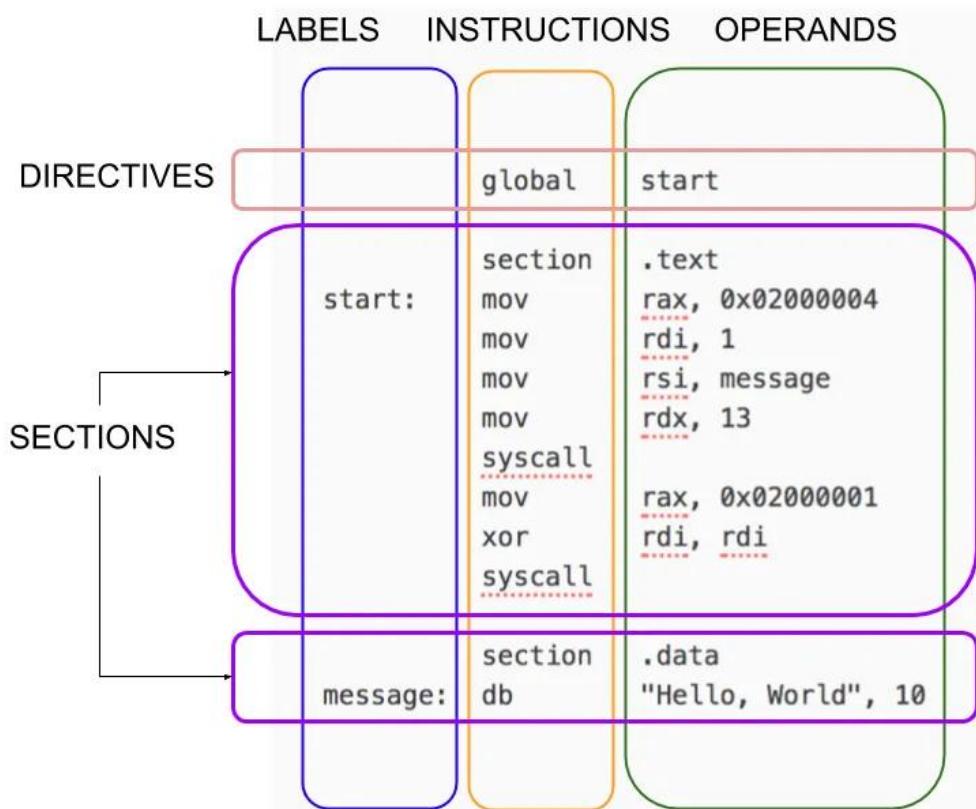
- .text - zawiera instrukcje wykonywalne (kod programu). Tutaj znajduje się logika aplikacji, instrukcje oraz skoki.
- .data - zawiera dane inicjalizowane, czyli zmienne zdefiniowane przez programistę, które mają przypisaną wartość początkową. Przykład:

```
section .data  
liczba dd 10  
tekst db "Hello", 0
```

- .bss - przeznaczona jest dla danych, które nie są inicjalizowane (nie mają przypisanej początkowej wartości). Rezerwuje jedynie przestrzeń w pamięci. Przykład:

```
section .bss  
bufor resb 64
```

Każdy program w NASM powinien zawierać sekcję .text, w której umieszcza się wykonywalny kod programu oraz instrukcję \_start, będącą punktem wejścia programu.



Struktura programu napisanego w języku niskopoziomowym

## 1.7 Rejestry w architekturze x86-64 – short intro

W architekturze **x86-64** mamy do dyspozycji zestaw rejestrów 64-bitowych. Poniżej omówione są najważniejsze z nich, z perspektywy programowania w asemblerze NASM w systemach Linux/Unix. Rejestry x86-64 omówimy dokładniej na kolejnych zajęciach.

### Rejestry ogólnego przeznaczenia (64-bitowe)

W trybie 64-bitowym każdy register ma 64 bity, a dodatkowo można się odwoływać do jego niższych części (32-, 16- i 8-bitowych).

#### 1. RAX

- „Accumulator” (akumulator).
- Historycznie służył do operacji arytmetycznych i logicznych.
- W wywołaniach funkcji w System V AMD64 ABI służy także do zwracania wartości (np. wynik funkcji).
- Niższe części:
  - EAX (32 bity)
  - AX (16 bitów)
  - AL (najniższe 8 bitów)

#### 2. RBX

- „Base register” dawniej używany do adresowania pamięci (w x86 16-bit).
- W 64-bitach może pełnić dowolną funkcję.
- Jest rejestrem typu *callee-saved* (funkcja wywoływana musi przywrócić jego wartość przed powrotem).
- Niższe części: EBX, BX, BL.

#### 3. RCX

- „Counter register”, często używany np. do zliczania pętli (choć w nowoczesnym kodzie można użyć dowolnego rejestru).
- W niektórych wywołaniach funkcji (np. syscall) służy do przekazywania argumentów.
- Niższe części: ECX, CX, CL.

#### 4. RDX

- Często używany w operacjach dzielenia i mnożenia (razem z RAX).
- W ABI AMD64 bywa 3. rejestrem do przekazywania argumentu do funkcji (po RDI, RSI).
- Niższe części: EDX, DX, DL.

## 5. RSI

- „Source index” – w 16/32-bitowym kodzie często używany do wskazywania źródła operacji na łańcuchach (instrukcje string).
- W ABI AMD64 jest 2. rejestrem do przekazywania argumentu do funkcji.
- Niższe części: ESI, SI, SIL.

## 6. RDI

- „Destination index” – w dawnych instrukcjach string do wskazywania docelowej lokacji.
- W ABI AMD64 jest 1. rejestrem do przekazywania argumentu do funkcji.
- Niższe części: EDI, DI, DIL.

## 7. RBP

- „Base pointer” (lub frame pointer).
- Często używany do przechowywania wskaźnika na ramkę stosu funkcji.
- Jest rejestrem *callee-saved*.
- Niższe części: EBP, BP, BPL.

## 8. RSP

- „Stack pointer” – zawsze wskazuje na wierzchołek stosu.
- Modyfikacje RSP (np. push, pop, call, ret) automatycznie go zmieniają.
- Nie dzieli się go zwykle na 32/16/8 bitów w praktyce (teoretycznie to możliwe, ale nieużyteczne).
- W ABI AMD64 musi być **wyrównany do 16 bajtów** przed wywołaniem funkcji w C, C++ itp.

## 9. R8, R9, R10, R11, R12, R13, R14, R15

- Dodatkowe rejesty 64-bitowe wprowadzone w trybie x86-64.
- R8 i R9 są odpowiednio 5. i 6. rejestrem do przekazywania argumentów w ABI AMD64.
- R10 i R11 to rejesty *caller-saved* (może je modyfikować funkcja wywoływana, nie musi przywracać).
- R12, R13, R14, R15 są rejestrami *callee-saved* (funkcja wywoływana ma je przywrócić).
- Każdy z nich ma niższe części (np. R8d, R8w, R8b itd.).

## **Rejestr instrukcji (Instruction Pointer)**

- **RIP**
  - Wskazuje na aktualnie wykonywaną instrukcję.
  - Nie można go modyfikować bezpośrednio za pomocą standardowych rozkazów (zamiast tego mamy skoki, wywołania, rozkazy jmp, call, ret itp.).
  - W 64-bitach dostępny jest tzw. *RIP-relative addressing*, co umożliwia wygodny dostęp do danych w sekcji .data stosunkowo do bieżącej instrukcji.

## **Rejestr flag (RFLAGS)**

- Zawiera bity flag, takie jak:
  - CF (Carry Flag),
  - ZF (Zero Flag),
  - SF (Sign Flag),
  - OF (Overflow Flag),
  - itp.
- W 64-bitach nazywany jest RFLAGS, w 32-bitach – EFLAGS, a w 16-bitach – FLAGS.
- Ustawiany/modyfikowany przez instrukcje arytmetyczne i logiczne (add, sub, cmp, test...) oraz przez instrukcje typu pushf, popf, lahf, sahf.

## **Podsumowanie**

- **RDI, RSI, RDX, RCX, R8, R9** – sześć pierwszych argumentów funkcji w trybie 64-bit (System V).
- **RSP** – wskaźnik stosu, zawsze aktualizowany przez push/pop/call/ret.
- **RAX** – wynik zwracany z funkcji.
- **RBX, RBP, R12–R15** – jeśli funkcja ich używa, musi je sama odtworzyć przed powrotem.
- **RIP** – rejestr instrukcji, do którego nie ma bezpośredniego zapisu (używamy rozkazów skoku/wywołania).

W NASM odwołania do rejestrów piszemy zawsze poprzedzone r, np. rax, rbx, rcx..., a w 32-bitowym asemblerze analogicznie: eax, ebx, ecx....

## 2. Przygotowanie środowiska

**NASM** (Netwide Assembler) jest popularnym asemblerem, czyli narzędziem służącym do konwertowania kodu napisanego w języku asemblerowym na kod maszynowy, który może być wykonany bezpośrednio przez procesor. NASM obsługuje architekturę x86 oraz x86-64, jest prosty w obsłudze, przenośny i zgodny z wieloma systemami operacyjnymi (Linux, macOS, Windows).

**LD** (Linker) to program służący do łączenia („linkowania”) osobnych fragmentów kodu – wygenerowanych przez asembler lub kompilator – w gotowy do uruchomienia plik wykonywalny. Linker odpowiada za prawidłowe umieszczenie kodu i danych w pamięci oraz rozwiązywanie zależności między różnymi fragmentami programu. Bez użycia linkera kod wygenerowany przez NASM nie byłby możliwy do uruchomienia jako samodzielna aplikacja.

W terminalu Linux (Ubuntu/Debian) wykonaj następujące komendy:

```
sudo apt-get update -y  
sudo apt-get install -y nasm
```

Aby upewnić się, że NASM i linker (ld) zostały poprawnie zainstalowane, wpisz:

```
nas -v  
ld -v
```

Dodatkowo aby kompliować programy x86 na Linux x64 musimy wykonać następujące komendy:

```
sudo dpkg --add-architecture i386  
sudo apt update  
sudo apt install gcc-multilib libc6-dev-i386
```

## Instalacja VS Code na Linux - Dystrybucje Debian i Ubuntu

Najłatwiejszym sposobem instalacji Visual Studio Code w dystrybucjach opartych na Debianie/Ubuntu jest pobranie i zainstalowanie pakietu [.deb package \(64-bit\)](#), zarówno poprzez graficzne centrum oprogramowania (jeśli jest dostępne), jak i za pomocą wiersza poleceń, używając następującego polecenia:

```
sudo apt install ./<file>.deb  
# If you're on an older Linux distribution, you will need to run this instead:  
# sudo dpkg -i <file>.deb  
# sudo apt-get install -f # Install dependencies
```

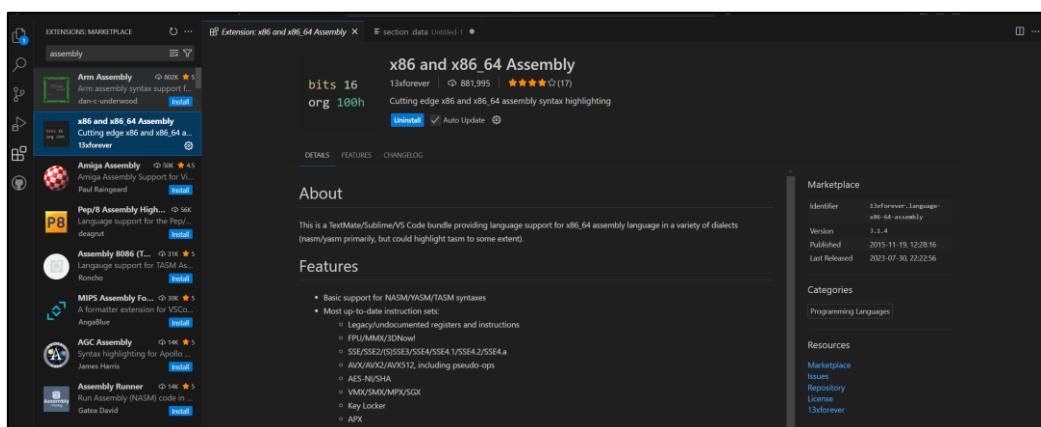
Podczas instalacji pakietu **.deb** zostanie wyświetlony monit o dodanie repozytorium **APT** oraz klucza podpisywania, co umożliwia automatyczne aktualizacje za pomocą menedżera pakietów systemowych.

## Metoda alternatywna

```
echo "code code/add-microsoft-repo boolean true" | sudo debconf-set-selections  
  
sudo apt-get install wget gpg  
wget -qO- https://packages.microsoft.com/keys/microsoft.asc | gpg --dearmor >  
packages.microsoft.gpg  
sudo install -D -o root -g root -m 644 packages.microsoft.gpg  
/etc/apt/keyrings/packages.microsoft.gpg  
echo "deb [arch=amd64,arm64,armhf signed-by=/etc/apt/keyrings/packages.microsoft.gpg]  
https://packages.microsoft.com/repos/code stable main" |sudo tee  
/etc/apt/sources.list.d/vscode.list > /dev/null  
rm -f packages.microsoft.gpg  
sudo apt install apt-transport-https  
sudo apt update  
sudo apt install code # or code-insiders
```

Pełna instrukcja instalacji dostępna [tutaj](#).

Następnie instalujemy rozszerzenie **x86 and x86\_64 Assembly**.



### 3. Zadania praktyczne

#### Część Fabularna

##### Pierwszy program w asemblerze x86 – "Hello, World!"

Pierwszym krokiem jest otwarcie pliku w edytorze tekstowym i utworzenie dwóch sekcji: **.text** i **.data**. W asemblerze sekcja jest najmniejszą jednostką kodu, którą można przenosić w plikach formatu **.elf** (Executable and Linkable Format – używany do programów wykonywalnych, bibliotek i innych zasobów).

Sekcje mogą zawierać różne typy danych, np.:

- **.text** – sekcja kodu wykonywalnego
- **.data** – dane tylko do odczytu
- **.bss** – dane do odczytu i zapisu, ale niezainicjalizowane

Tworzymy sekcje za pomocą słowa kluczowego section:

```
section .text ; Tworzy sekcję kodu
section .data ; Tworzy sekcję danych
```

##### Deklaracja punktu startowego programu

Następnie dodajemy punkt startowy programu, który jest wymagany dla linkera:

```
section .text ; Tworzy sekcję kodu
global _start ; Deklaracja punktu startowego
_start:       ; Sekcja startowa
section .data ; Tworzy sekcję danych
```

##### Definiowanie danych w sekcji .data

Teraz deklarujemy dane, które będziemy wykorzystywać w programie. Do wyświetlenia komunikatu "Hello, World!" potrzebujemy dwóch rzeczy:

- Treści wiadomości
- Długości wiadomości

W asemblerze musimy jawnie podać długość tekstu, co daje nam pełną kontrolę nad operacjami pamięci. Tworzymy nasz komunikat za pomocą db (define byte):

```
section .data
msg db "Hello, World!", 0xa ; Deklaracja wiadomości (0xa = nowa linia)
len equ $ - msg      ; Obliczenie długości wiadomości
```

## Przenoszenie danych do sekcji .text i ustawienie deskryptora pliku

Aby wyświetlić tekst, musimy:

1. Załadować długość wiadomości do rejestru edx
2. Załadować adres wiadomości do rejestru ecx
3. Ustawić deskryptor pliku w rejestrze ebx (stdout = 1)
4. Wywołać systemową funkcję zapisu (sys\_write)

```
section    .text
global     _start
_start:
    mov edx, len ; Przenosi długość wiadomości do EDX
    mov ecx, msg ; Przenosi adres wiadomości do ECX
    mov ebx, 1   ; Ustawia deskryptor pliku (stdout)
```

## Wywołanie systemowej funkcji do wyświetlania tekstu

W systemie Linux funkcje systemowe są wywoływane za pomocą **przerwania int 0x80**.

Każda funkcja systemowa ma swój numer ID – do wyświetlania tekstu służy sys\_write (kod: 4), więc ustawiamy eax = 4 i wykonujemy przerwanie:

```
mov eax, 4 ; Kod syscall dla sys_write
int 0x80 ; Wywołanie przerwania do jądra systemu
```

## Zakończenie programu

Po zakończeniu operacji należy zakończyć program, przekazując sterowanie do jądra systemu. Funkcja sys\_exit (kod 1) wymaga wartości 0 jako kodu zakończenia:

```
mov eax, 1 ; Kod syscall dla sys_exit
xor ebx, ebx ; Ustawia kod wyjścia na 0
int 0x80 ; Wywołanie przerwania kończącego program
```

### Końcowy kod programu "Hello, World!" w NASM (32-bit x86, Linux)

```
section    .text
global     _start

_start:
    mov edx, len ; Przenosi długość wiadomości do EDX
    mov ecx, msg ; Przenosi adres wiadomości do ECX
    mov ebx, 1   ; Ustawia deskryptor pliku (stdout)
    mov eax, 4   ; Syscall: sys_write
    int 0x80    ; Wywołanie systemowe

    mov eax, 1   ; Syscall: sys_exit
    xor ebx, ebx ; Kod wyjścia = 0
    int 0x80    ; Zakończenie programu

section    .data
msg      db "Hello, World!", 0xa ; Deklaracja wiadomości
len      equ $ - msg           ; Obliczenie długości wiadomości
```

### Jak skompilować i uruchomić kod?

1. **Kompilacja** (NASM + linker ld):
  2. nasm -f elf32 hello.asm -o hello.o
  3. ld -m elf\_i386 hello.o -o hello
4. **Uruchomienie programu:**
  5. ./hello

## Przykłady i zadania

### Przykład 1 – Hello World

Napisz program, który wypisze na ekran tekst „Hello world”:

Przykład programu „Hello World”

```
section .data
msg db "Hello world!", 0xa
len equ $ - msg

section .text
global _start
_start:
    mov edx, len
    mov ecx, msg
    mov ebx, 1
    mov eax, 4
    int 0x80

    mov eax, 1
    int 0x80
```

Kompilacja programu:

```
nasm -f elf32 hello.asm -o hello.o
ld -m elf_i386 hello.o -o hello
./hello
```

Alternatywnie:

```
nasm -f elf32 -o hello.o hello.asm && ld -m elf_i386 -o hello hello.o
```

Kod wraz z komentarzami:

```
section .data          ; Sekcja danych inicjalizowanych
    msg db "Hello world!", 0xa ; Definicja tekstu do wypisania („Hello world!”) zakończonego
                                znakiem nowej linii (0xa)
    len equ $ - msg          ; Obliczenie długości tekstu do wypisania (automatycznie obliczana
                                przez asembler)

section .text
    global _start           ; Deklaracja punktu wejścia programu

_start:
    mov edx, len            ; Umieszcza długość tekstu (len) w rejestrze edx (liczba znaków do
                                wypisania)
    mov ecx, msg             ; Umieszcza adres tekstu do wypisania w rejestrze ecx
    mov ebx, 1                ; Numer deskryptora pliku (stdout), czyli standardowe wyjście
    mov eax, 4                ; Numer funkcji systemowej dla operacji „write” (wypisywanie danych
                                na ekran)
    int 0x80                 ; Wywołanie przerwania systemowego, wypisuje tekst na ekran

    mov eax, 1                ; Numer funkcji systemowej dla zakończenia działania programu (exit)
    int 0x80                 ; Wywołanie jądra systemu Linux, aby zakończyć działanie programu

; Poniższe instrukcje są zbędne (powtórne zakończenie), można je usunąć:
; mov eax, 1
; int 0x80
```

## Przykład 2 – Podstawowe dodawanie dwóch liczb

Prosty program dodający dwie liczby:

```
section .data
    num1 dd 5
    num2 dd 3
section .text
    global _start
_start:
    mov eax, [num1]
    mov ebx, [num2]
    add eax, ebx

    mov eax, 1
    int 0x80
```

### Kompilacja:

```
nasm -f elf32 -o add.o add.asm && ld -m elf_i386 -o add add.o
```

### Kod z dodatkowymi komentarzami:

```
section .data
    num1 dd 5      ; deklaracja zmiennej num1 (4 bajty - double word), przypisanie wartości
    początkowej 5
    num2 dd 3      ; deklaracja zmiennej num2 (4 bajty - double word), przypisanie wartości
    początkowej 3

section .text
    global _start   ; deklaracja punktu wejścia, niezbędnego dla linkera

_start:
    ; Wczytanie wartości do rejestrów z pamięci
    mov eax, [num1] ; przeniesienie wartości zmiennej num1 do rejestru eax (eax = 5)
    mov ebx, [num2] ; przeniesienie wartości zmiennej num2 do rejestru ebx (ebx = 3)

    ; Wykonanie operacji dodawania
    add eax, ebx    ; dodanie zawartości rejestrów eax i ebx (eax = eax + ebx, czyli eax = 8)

    ; Zakończenie działania programu (system call „exit”)
    mov eax, 1      ; kod systemowy funkcji exit (1) – kończy program
    int 0x80        ; wywołanie przerwania systemowego w celu zakończenia programu
```

### Zadanie 1 – Dodawanie dwóch liczb z wyświetleniem wyniku

Wykorzystując kod z przykładu 2 napisać program który dodaje 2 liczby (np. 9 i 6) i wyświetla wynik w terminalu użytkownika. Wynik działania może wyglądać następująco:

```
mateusz@DESKTOP-JRP3SMH:~$ ./add2  
Wynik: 15
```

## Zadanie 2 – Dodawanie dwóch liczb wpisanych przez użytkownika z wyświetlaniem wyniku

Wykorzystując kod z zadania 1 napisać program który dodaje 2 liczby wprowadzone przez użytkownika i wyświetla wynik w terminalu użytkownika. Wynik działania może wyglądać następująco:

```
Enter the first number  
3  
Enter the 2nd number :  
5  
The sum is: 8
```

### Przykład 3 – Integracja z biblioteką C

Ten przykład przedstawia, jak wywoływać funkcję **puts** z biblioteki standardowej C w programie napisanym w języku asemblera NASM dla **architektury x86 (32-bit)**. Program wypisuje na ekran ciąg znaków "Hello World" przy użyciu funkcji puts, a następnie kończy swoje działanie.

```
global main
extern puts

section .text
main:
    push ebp          ; zachowanie ramki stosu (standard dla x86)
    mov ebp, esp

    push message      ; przekazanie adresu tekstu na stos jako argument do puts
    call puts          ; wywołanie funkcji bibliotecznej puts
    add esp, 4         ; przywrócenie stosu (usunięcie argumentu)

    mov eax, 0          ; zwrócenie wartości 0 z main
    leave              ; odtworzenie ramki stosu i powrót do wywołującego
    ret

section .data
message:
    db "Hello World", 0 ; ciąg znaków zakończony bajtem zerowym (zgodnie ze standardem C)
```

#### Kompilacja:

```
nasm -f elf32 -o useC.o useC.asm && ld -m elf_i386 -o useC useC.o
```

## Przykład 4 – Wywołanie Assembly NASM w kodzie C

Znajdowanie maksimum z trzech liczb (w tym przypadku program napisany został w architekturze x64)

```
; -----
; A 64-bit function that returns the maximum value of its three 64-bit integer
; arguments. The function has signature:
;
; int64_t maxofthree(int64_t x, int64_t y, int64_t z)
;
; Note that the parameters have already been passed in rdi, rsi, and rdx. We
; just have to return the value in rax.
;

global maxofthree
section .text
maxofthree:
    mov    rax, rdi      ; result (rax) initially holds x
    cmp    rax, rsi      ; is x less than y?
    cmovl rax, rsi      ; if so, set result to y
    cmp    rax, rdx      ; is max(x,y) less than z?
    cmovl rax, rdx      ; if so, set result to z
    ret                 ; the max will be in rax
```

Przykład programu w C który wywoła powyższy program:

```
/* * A small program that illustrates how to call the maxofthree function we wrote in *
assembly language. */

#include <stdio.h>
#include <inttypes.h>

int64_t maxofthree(int64_t, int64_t, int64_t);

int main() {
    printf("%ld\n", maxofthree(1, -4, -7));
    printf("%ld\n", maxofthree(2, -6, 1));
    printf("%ld\n", maxofthree(2, 3, 1));
    printf("%ld\n", maxofthree(-2, 4, 3));
    printf("%ld\n", maxofthree(2, -6, 5));
    printf("%ld\n", maxofthree(2, 4, 6));
    return 0;
}
```

### Kompilacja:

```
nasm -felf64 maxofthree.asm && gcc callmaxofthree.c maxofthree.o -o callmaxofthree
```

### Zadanie 3 – Wypisanie liczb od 1 do 9 przy użyciu pętli

Napisać program który w pętli wypisze liczby od 1 do 9. Przykładowy wynik działania programu może wyglądać następująco.

```
mateusz@DESKTOP-JRP3SMH:~$ ./loop
123456789:mateusz@DESKTOP-JRP3SMH:~$ █
```

Podpowiedź: [https://www.tutorialspoint.com/assembly\\_programming/assembly\\_loops.htm](https://www.tutorialspoint.com/assembly_programming/assembly_loops.htm)

#### Zadanie 4 – Wyświetlanie trójkąta

Napisać program generujący trójkąt z gwiazdek. Wynik działania programu powinien wyglądać następująco:

```
mateusz@DESKTOP-JRP3SMH:~$ ./triangle
*
**
***
****
*****
*****
*****
*****
```

## Przykład 5 – Wyświetlanie argumentów z linii poleceń

Ten program w asemblerze NASM dla architektury **x86-64** pobiera i wyświetla argumenty przekazane do programu w wierszu poleceń, wypisując każdy argument w osobnej linii. Program korzysta z funkcji puts z biblioteki standardowej C do wyświetlania argumentów.

```
; A 64-bit program that displays its command line arguments, one per line.  
; On entry, rdi will contain argc and rsi will contain argv.  
  
global main  
extern puts  
section .text  
  
main:  
    push rdi          ; save registers that puts uses  
    push rsi  
    sub  rsp, 8       ; must align stack before call  
  
    mov   rdi, [rsi]   ; the argument string to display  
    call  puts         ; print it  
  
    add   rsp, 8       ; restore %rsp to pre-aligned value  
    pop   rsi          ; restore registers puts used  
    pop   rdi  
  
    add   rsi, 8       ; point to next argument  
    dec   rdi          ; count down  
    jnz  main          ; if not done counting keep going  
  
ret
```

- Po uruchomieniu programu w **rejestrze rdi** znajduje się liczba argumentów (argc).
- W **rejestrze rsi** znajduje się wskaźnik na tablicę argumentów (argv), czyli lista stringów przekazanych do programu.
- Program zapisuje rdi i rsi na stosie (push), aby zachować wartości.
- **sub rsp, 8** wyrównuje stos (stack alignment) przed wywołaniem puts.
- **mov rdi, [rsi]** – pobiera aktualny argument (wskaźnik do stringa) i umieszcza go w rdi, który jest pierwszym argumentem dla puts.
- **call puts** – wywołuje funkcję puts, która wypisuje argument.
- **add rsp, 8** – przywraca stos do poprzedniego stanu.
- **pop rsi, pop rdi** – przywraca wartości rsi i rdi.
- **add rsi, 8** – przesuwa wskaźnik rsi do następnego argumentu (argv[i+1]).
- **dec rdi** – zmniejsza licznik argc (ilości argumentów).
- **jnz main** – jeśli argc > 0, powtarza pętlę.
- **ret** – powrót z funkcji, kończy program.

## Zadanie 5 – Potęgowanie

Napisać program który wykonuje operację potęgowania liczby  $x^y$ . Przykładowe wykonanie programu może wyglądać następująco:

```
mateusz@DESKTOP-JRP3SMH:~$ ./pow
```

```
Podaj podstawa x: 3
```

```
Podaj wykładnik y: 5
```

```
Wynik = 243
```

```
mateusz@DESKTOP-JRP3SMH:~$ ./pow 2
```

## Zadania dodatkowe dla chętnych

### Zadanie 6 – Obliczanie średniej z argumentów

Napisać program, który traktuje wszystkie swoje argumenty wiersza poleceń jako liczby całkowite i wyświetla ich średnią jako liczbę zmienoprzecinkową.

### Zadanie 7 – Rekursywna funkcja obliczająca silnię

Napisać funkcję rekursywną która oblicza silnię.

### Zadanie 8 – Fibbonaci

Napisać program wypisujący pierwsze 90 liczb Fibonacciego.

Podpowiedź: warto poszukać rozwiązań dla architektury x64

Więcej przykładowych programów i ćwiczeń można znaleźć pod adresem [Kurs ASM](#). 😊

Warto zobaczyć również: <https://www.youtube.com/@olivestemlearning/videos>