

# ASSEMBLER X86 I X86-64

ŁUKASZ GUT, GRZEGORZ LITAROWICZ

# AGENDA

---

- Co to jest Assembly i Assembler?
- Rodzaje Assembly i Assemblerów
- Hello, world!
- Proces kompilacji
- Podstawy składni
- Krótko o segmentach pamięci
- Krótko o rejestrach
- Tryby adresowania
- Deklarowanie zmiennych oraz stałych
- Arytmetyka
- Operatory logiczne
- Instrukcje warunkowe
- Pętle
- Tablice
- Procedury
- Analiza mikro optymalizacji robionych przez kompilator

# CO TO JEST ASSEMBLY I ASSEMBLER

---

**Assembly** - niskopoziomowy język programowania silnie skorelowany z architekturą procesora i instrukcjami rozumianymi przez ten procesor.

**Assembler** - program komputerowy przekształcający instrukcje napisane w niskopoziomowym języku assembly w kod maszynowy.

# RODZAJE ASSEMBLY I ASSEMBLERÓW

---

Istnieje wiele rodzajów assembly, między innymi:

- x86 assembly
- x86-64 assembly
- ARM assembly
- i więcej ...

Istnieje także wiele rodzajów assemblerów, między innymi:

- NASM
- GNU Assembler
- MASM
- i więcej ...

# HELLO, WORLD!

---

```
section    .text
global _start

_start:
    mov     rdx, len
    mov     rsi, msg
    mov     rdi, 1
    mov     rax, 1
    syscall

    mov     rax, 60
    xor     rdi, rdi
    syscall

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

# PROCES KOMPILACJI

---

Aby skompilować i zlinkować assembly w wykorzystaniem assemblera NASM i linkera GNU wpisujemy następujące komendy:

```
nasm -f elf64 hello-world.asm #tworzenie pliku obiektowego  
ld hello-world.o #linkowanie
```

# PODSTAWY SKŁADNI - SEKCJE

---

Program w assembly dzieli się na 3 sekcje:

- **data**
- **bss**
- **text**

# PODSTAWY SKŁADNI - SEKCJA DATA

---

Sekcja **data** jest wykorzystywana do deklarowania zainicjalizowanych stałych. Dane w tej sekcji nie zmieniają się w trakcie działania programu.

Deklaracja sekcji data wygląda następująco:

```
section .data
```



# PODSTAWY SKŁADNI - SEKCJA BSS

---

Sekcja **bss** jest wykorzystywana do deklarowania zmiennych.

Deklaracja sekcji bss wygląda następująco:

```
section .bss
```

# PODSTAWY SKŁADNI - SEKCJA TEXT

---

Sekcja **text** jest wykorzystywana do trzymania kodu programu. Sekcja ta musi zaczynać się deklaracją *global \_start*, która mówi kernelowi gdzie rozpocząć wykonywanie programu.

Deklaracja sekcji text wygląda następująco:

```
section .text  
    global _start  
_start:
```

# PODSTAWY SKŁADNI - POLECENIA

---

Polecenia w języku assembly wprowadzamy po jednym na linię. Każde polecenie ma następujący format:

```
[label]    mnemonic    [operands]    [;comment]
```

# PODSTAWY SKŁADNI - LABEL

---

Labele (etykiety) - nadają nazwy adresom pamięci, aby można było się do nich odnosić w łatwy sposób.

```
section    .text  
global _start
```

```
_start:  
    mov     rdx, len  
    mov     rsi, msg  
    mov     rdi, 1  
    mov     rax, 1  
    syscall
```

```
    mov     rax, 60  
    xor     rdi, rdi  
    syscall
```

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

# PODSTAWY SKŁADNI - MNEMONIC ORAZ OPERAND

---

Mnemonic - nazwa przypisana do instrukcji wykonywanej przez procesor. Operand - argument mnemonica.

Przykładowe mnemonic'i:

```
inc byte [count] ; Inkrementacja zmiennej pod adresem 'count'
```

```
mov byte [total], 48 ; Kopiuje liczbę pod adres 'total'
```

```
add ah, bh ; Dodaje zawartość rejestru bh do ah i zapisuje w ah
```

# SEGMENTY PAMIĘCI

---

Segmenty pamięci - **model pamięciowy** dzielący pamięć systemową na niezależne segmenty, do których możemy się dostać za pomocą wskaźników znajdujących się w specjalnych rejestrach.

Segment vs Sekcja:

"The SECTION directive (SEGMENT is an exactly equivalent **synonym**)..."

NASM documentation

# SEGMENTY PAMIĘCI - DANE

---

Segment danych - reprezentowany przez sekcje `.data` oraz `.bss`.

Sekcja `.data` - deklaruje statyczny region pamięci, w którym znajdują się zmienne globalne. Sekcja ta fizycznie występuje w pliku wykonywalnym - loader musi ją załadować do pamięci ram za nas.

Sekcja `.bss` - deklaruje statyczny obszar pamięci, w którym pamięć jest wypełniona zerami. Ładują tutaj globalne zmienne niezainicjalizowane, lub zainicjalizowane zerami. Sekcja ta nie występuje również w pliku wykonywalnym - loader wie, że przed wejściem do funkcji `main` należy zaalokować pamięć na tę sekcję i wypełnić ją zerami.

# SEGMENTY PAMIĘCI - KOD

---

Segment kodu - reprezentowany przez sekcję `.text`.

Sekcja `.text` - definiuje obszar w pamięci, w którym przechowywane są instrukcje programu. Ten obszar również jest statyczny.



# SEGMENTY PAMIĘCI - STOS

---

Segment stosu - zawiera tymczasowe wartości.

# REJESTRY

---

Rejestr – układ służący do przechowywania i odtwarzania informacji w postaci bitów. Na każdej pozycji rejestru przechowywany jest jeden bit informacji.

# REJESTRY - OGÓLNEGO PRZEZNACZENIA

---

Mamy 16 rejestrów ogólnego przeznaczenia. Możemy się odwoływać do wszystkich 64-bitów danego rejestru lub do ich części. Przechowują one następujące informacje:

- argumenty dla operacji logicznych i arytmetycznych,
- argumenty dla operacji dokonywanych na adresach,
- wskaźniki na miejsca w pamięci.

# REJESTRY - OGÓLNEGO PRZEZNACZENIA

---

- **RSP** - Rejestr w którym przechowywany jest aktualny wskaźnik na ostatni element dodany do stosu.
- **RBP** - Rejestr w którym przechowywany jest wskaźnik, wykorzystywany przy wywoływaniu funkcji.

# REJESTRY - RIP

---

**RIP** - Rejestr w którym przechowywany jest wskaźnik na następną instrukcję do wykonania.

# REJESTR - FLAG

---

Rejestry który jest używany, aby pokazać status oraz informacje kontrolne. Jest on aktualizowany przez procesor po wykonaniu każdej instrukcji. Przykładowe flagi:

- **ZF** - przetrzymuje informację czy wynikiem ostatniej operacji było 0,
- **CF** - służy do wskazania czy poprzednia operacja zakończyła się przeniesieniem.

# REJESTR - XMM

---

Rejestry o rozmiarze 128-bitów, przeznaczone do 64-bitowych oraz 32-bitowych operacji zmiennoprzecinkowych. Wspierają one również SIMD (Single Instruction Multiple Data). Nazwy rejestrów to xmm0 - xmm15.

# WYWOŁANIA SYSTEMOWE

---

Wywołania systemowe (syscalls) - podstawowy **interfejs** pomiędzy aplikacją a jądrem systemu.



# WYWOŁANIA SYSTEMOWE - JAK TO ROBIĆ?

---

Aby skorzystać z wywołania systemowego na Linuxie należy:

- Umieścić numer wywołania systemowego w rejestrze **RAX**
- Umieścić argumenty wywołania systemowego w rejestrach **RDI**, **RSI**, ...
- Wywołać przerwanie **syscall**
- Rezultat zazwyczaj jest zwracany do rejestru **RAX**

# TRYBY ADRESOWANIA

---

Podstawowe tryby adresowania:

- adresowanie rejestrowe,
- adresowanie natychmiastowe,
- adresowania bezpośrednie.

# TRYBY ADRESOWANIA - ADRESOWANIE REJESTROWE

---

Adresowanie rejestrowe - to tryb adresowania, gdy rejestr jest pierwszym, drugim lub obydwojema parametrami mnemonica.

Przykłady:

```
mov rdx, var  
mov rax, rbx
```

# TRYBY ADRESOWANIA - ADRESOWANIE NATYCHMIASTOWE

---

Adresowanie natychmiastowe - następuje gdy, drugim parametrem jest stała wartość lub stałe wyrażenie.

Przykłady:

```
add    byte [byte_val], 65  
mov    ax, 45H
```

# TRYBY ADRESOWANIA - ADRESOWANIE BEZPOŚREDNIE

---

Adresowanie bezpośrednie - występuje gdy wymagany jest dostęp do pamięci np do segmentu .data. Tryb ten jest znacznie wolniejszy od poprzednich.

Przykłady:

```
add    [byte_val], dl
```

# DEKLAROWANIE ZMIENNYCH - SKŁADNIA

---

Składnia deklarowania zmiennych wygląda następująco:

```
[variable-name] define-directive initial-value [,initial-value]...
```

# DEKLAROWANIE ZMIENNYCH - DYREKTYWY

---

Dyrektywa	Ilość zaalokowanej pamięci (w bajtach)
DB	1
DW	2
DD	4
DQ	8
DT	10

# DEKLAROWANIE ZMIENNYCH NIEZAINICJALIZOWANYCH

---

Dyrektywa	Ilość zarezerwowanej pamięci (w bajtach)
RESB	1
RESW	2
RESD	4
RESQ	8
REST	10



# WIELOKROTNE INICJALIZACJE

---

Istnieją sytuacje, w których chcemy zainicjalizować wiele bajtów tą samą wartością (np. tablice). Można wykorzystać do tego dyrektywę **TIMES**.

```
arr TIMES 9 DW 0
```

# DEFINIOWANIE STAŁYCH

---

Stałe w assembly możemy definiować na 3 sposoby, wykorzystując trzy różne dyrektywy:

- EQU
- %assign
- %define

# DEFINIOWANIE STAŁYCH - DYREKTYWA EQU

---

Dyrektywa EQU jest wykorzystywana do definiowania stałych.

Składnia:

```
CONSTANT_NAME EQU expression
```

Stałe te można potem wykorzystywać w kodzie:

```
mov eax, CONSTANT_NAME
```

# DEFINIOWANIE STAŁYCH - DYREKTYWA %ASSIGN

---

Dyrektywa `%assign` jest również wykorzystywana do definiowania stałych, jednak pozwala na **redefinicję**.

Składnia:

```
%assign CONSTANT_NAME expression
```

# DEFINIOWANIE STAŁYCH - DYREKTYWA %DEFINE

---

Dyrektywa %define jest bardzo podobna do #define znanego z języka C.

Składnia:

```
%define CONSTANT_NAME expression
```

# ARYTMETYKA - INSTRUKCJA INC

---

INC - instrukcja używana do inkrementacji przekazanego parametru.  
Składnia:

```
inc argument
```

Przykład:

```
inc rbx  
inc bx  
inc byte [val]
```

# ARYTMETYKA - INSTRUKCJE ADD ORAZ SUB

---

add/sub - instrukcja używana do dodawania i odejmowania.

Składnia:

```
add/sub dest, src
```

Przykład:

```
add rbx, rax  
sub val, rbx  
add rcx, 5
```

# ARYTMETYKA - INSTRUKCJE MUL ORAZ IMUL

---

IMUL - instrukcja używana do mnożenia liczb biorąc pod uwagę znak.

MUL - instrukcja używana do mnożenia liczb bez znaku.

Składnia:

```
mul/imul multiplier
```

Przykład:

```
mul dl; ax = al * dl(byte)
mul dx; dx:ax = ax * dx(word)
mul edx; edx:eax = eax * edx(double)
mul rdx; rdx:rax = rax * rdx(quad)
```



# ARYTMETYKA - INSTRUKCJE DIV ORAZ IDIV

---

IDIV - instrukcja używana do dzielenia liczb ze znakiem.

DIV - instrukcja używana do dzielenia liczb bez znaku.

Składnia:

```
div/idiv multiplier
```

Przykład:

```
div dl; ax = al * dl(byte)
div dx; dx:ax = ax * dx(word)
div edx; edx:eax = eax * edx(double)
div rdx; rdx:rax = rax * rdx(quad)
```

# OPERATORY LOGICZNE

---

Architektura IA-32 udostępnia nam również standardowe instrukcje logiczne:

- AND
- OR
- XOR
- TEST
- NOT

Składnia:

```
LOGICAL_INSTRUCTION REGISTER/MEMORY REGISTER/MEMORY/IMMEDIATE
```

# INSTRUKCJE WARUNKOWE - CMP

---

cmp - porównuje dwa parametry, nie edytując ich.

Składnia:

```
cmp arg1, arg2
```

# INSTRUKCJE WARUNKOWE - J...

---

J... - to polecenie wykonuje skok warunkowy na podstawie wyniku ostatniej instrukcji.

Składnia:

```
j... label
```

Przykład:

```
cmp rax, rbx
jz/je label ; wykonaj skok jeśli równe
...
cmp rax, 10
jnz/jne label ; wykonaj skok jeśli nie równe
```

# SKOK BEZWARUNKOWY - JMP

---

jmp - to polecenie wykonuje bezwarunkowy skok do etykiety.

Składnia:

```
jmp label
```

Przykład:

```
label:  
  add rax, rbx  
  jmp label ; wykonaj skok do etykiety label
```

# PĘTLE

---

Aby zaimplementować pętle w języku assembly trzeba wykorzystać "skoki" warunkowe, bądź zwykłe.

Przykład:

```
    mov rcx, 0x5
loop1:
    [body]
    dec rcx
    jnz 11
```

# PĘTLE - UPROSZCZENIE

---

Język assembly dostarcza nam specjalną instrukcję **loop**, która ułatwia pracę z pętlami.

Instrukcja ta *zakłada*, że w rejestrze ECX znajduje się licznik. W momencie wykonania tej instrukcji wartość w rejestrze ecx jest dekrementowana i wykonywany jest skok do podanej etykiety, dopóki wartość w rejestrze ECX jest większa od zera.

Przykład:

```
mov rcx, 0x5
loop1:
    [body]
    loop 11
```

# TABLICE

---

```
numbers    dw  34,  45,  56,  67,  75,  89
```

Deklaracja tablicy składającej się z 6 elementów po 2 bajty. "numbers" - będzie adresem pierwszego elementu, adresem kolejnego będzie numbers + 2.

```
tab dw  0,  0,  0,  0,  0,  0,  0,  0  
...  
tab times 8 dw 0
```



# PROCEDURY

---

Język assembly pozwala także na definiowanie procedur. Procedury składają się z **nazwy**, **ciała** i wyrażenia **ret**.

Składnia:

```
procedure_name:  
    [body]  
    ret
```

Do wywołania procedury używa się instrukcji **call**.

Składnia:

```
call procedure_name
```

# PROCEDURY - KONWENCJE WYWOŁYWAŃ

---

Konwencje wywołań są ważnym aspektem wywoływania procedur w językach assembly. Pozwalają one ujednolicić dostęp do argumentów funkcji oraz zdefiniować sposób zarządzania stosem.

- cdecl
- syscall
- optlink
- Microsoft fastcall
- ...

# PROCEDURY - CDECL

---

Zasady konwencji:

- Caller clean-up
- Argumenty przekazywane poprzez stos
- Liczby całkowite/adresy zwracane poprzez rejestr EAX
- Liczby zmiennoprzecinkowe zwracane poprzez rejestr ST0
- Rejestry "ulotne": EAX, ECX, EDX, ST0 - ST7, ES oraz GS
- Rejestry "nieulotne": EBX, EBP, ESP, EDI, ESI, CS oraz DS
- Procedura wychodzi z funkcji przy użyciu słowa *ret*

# OPTYMALIZACJA KODU C/C++ PRZEZ KOMPILATOR

---

- Inline funkcji
- Wektoryzacja pętli
- Rozwijanie pętli
- Optymalizacja pętli
- Optymalizacja pętli - ciąg arytmetyczny
- RVO/NRVO
- Inline if

**DZIĘKUJEMY ZA  
UWAGĘ!**