



Wprowadzenie do Assemblera x86-64

Klaudia Fil
Damian Płociennik

Akademia Górniczo-Hutnicza im. Stanisława Staszica w Krakowie
Wydział Fizyki i Informatyki Stosowanej

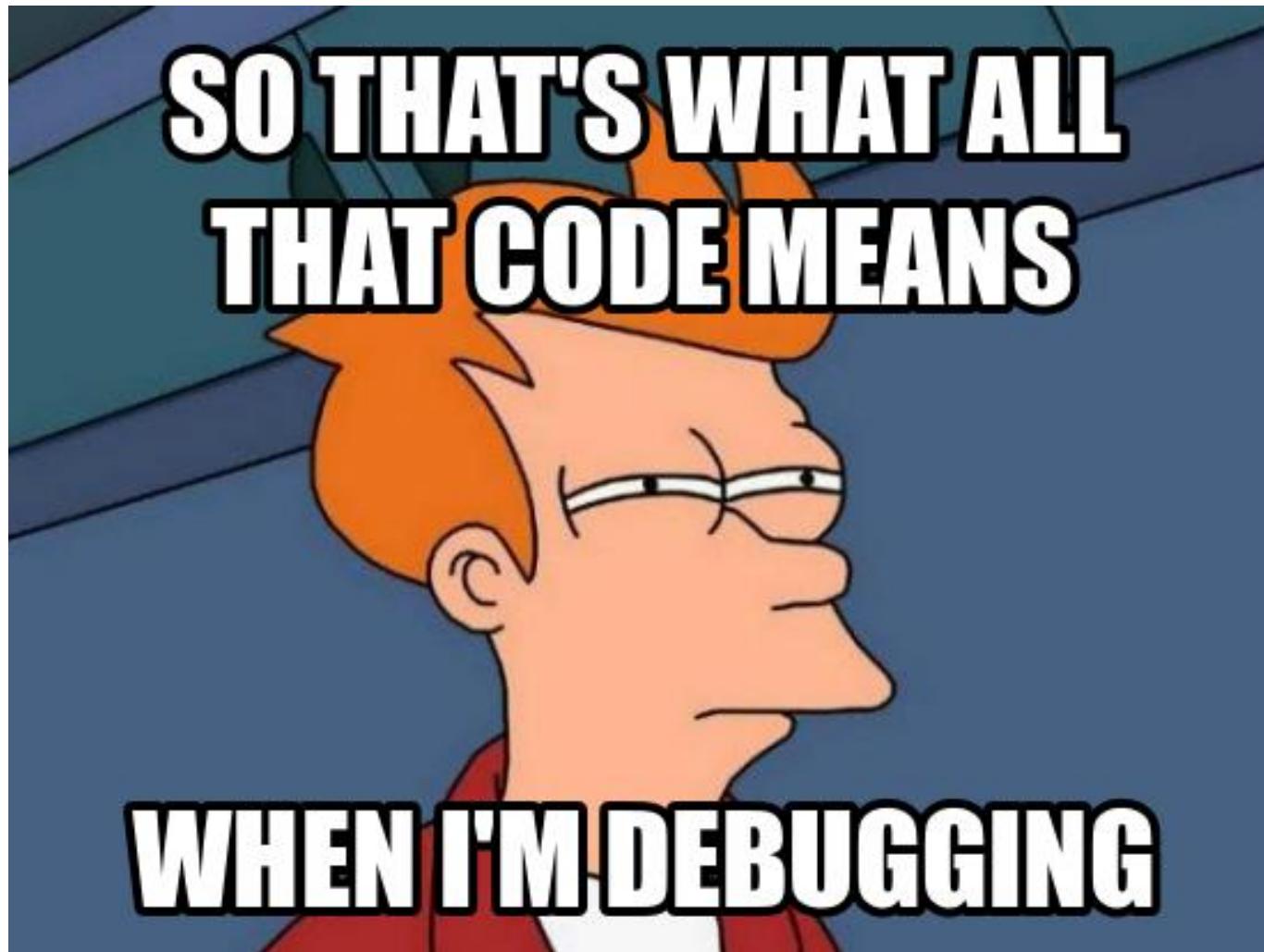
Agenda

1. Po co sięgać do języka wewnętrzniego? (Klaudia)
2. Assembly a Assembler (Damian)
3. „Hello world” w Assembly. Trzy główne sekcje. (Klaudia)
4. Segmente pamięci (Klaudia)
5. Rejestry (Damian)
6. Wywołania systemowe (Klaudia)
7. Tryby adresowania (Damian)
8. Zmienne i stałe (Klaudia)
9. Operatory arytmetyczne (Damian)
10. Instrukcje logiczne (Klaudia)
11. Skoki warunkowe i bezwarunkowe. Pętle. (Damian)
12. Łańcuchy znaków (Klaudia)
13. Tablice (Klaudia)
14. Procedury. Ramki stosu. (Klaudia)
15. Konwencje wywołań. (Damian)
16. Makra (Klaudia)
17. Godbolt.org (Damian)

Czego nie chcemy?



Co chcemy?



Po co sięgać do języka wewnętrznego?

Znajomość języka assemblera uświadamia:

- Jak programy łączą się z systemem operacyjnym, procesorem
- Jak dane są reprezentowane w pamięci i innych urządzeniach zewnętrznych
- Jak procesor uzyskuje dostęp do instrukcji i wykonuje je

Assembly a assembler

Assembly

Niskopoziomowy język programowania bazujący na podstawowych operacjach procesora

Assembler

Program dokonujący tłumaczenia języka asemblera na język maszynowy, czyli tzw. asemblacji.

Składnie assembly

» AT&T

```
mov $5, %rax
```

» Intel

```
mov rax, 5
```

Różne assembly

- » **NASM**
- » **FASM**
- » **MASM**
- » **TASM**
- » **YASM**
- » **GNU Assembler**
- » i więcej....

Asemblacja i konsolidacja

» Asemblacja

```
nasm -f elf64 nazwa.asm -o nazwa.o
```

» Konsolidacja (linkowanie)

```
ld nazwa.o -o nazwa
```

» Uruchomienie

```
./nazwa
```

Hello world w Assembly – trzy główne sekcje

```
section .text

    global _start

_start:
    mov    rax, 1          ; numer funkcji systemowej - sys_write
    mov    rdi, 1          ; numer pliku (1 == standardowe wyjscie)
    mov    rsi, message   ; adres tekstu
    mov    rdx, size       ; dlugosc tekstu
    syscall               ; wywolanie funkcji systemowej
    mov    rax, 60          ; numer funkcji systemowej - sys_exit
    syscall               ; wywolanie funkcji systemowej

section .data

message db "Hello, World", 0xa      ; napis wraz z znakiem nowej linii (10)
size    equ $ - message            ; dlugosc napisu
```

OUTPUT:

Hello, World

Hello world w Assembly – trzy główne sekcje

```
section .text
    global _start
_start:
    mov    rax, 1          ; numer funkcji systemowej - sys_write
    mov    rdi, 1          ; numer pliku (1 == standardowe wyjscie)
    mov    rsi, message   ; adres tekstu
    mov    rdx, size       ; dlugosc tekstu
    syscall
    mov    rax, 60          ; numer funkcji systemowej - sys_exit
    syscall

section .data
message db "Hello, World", 0xa
size    equ $ - message      ; napis wraz z znakiem nowej linii (10)
                                ; dlugosc napisu
```

```
section .bss
```

Hello world w Assembly – trzy główne sekcje

[label] mnemonic [operands] [;comment]

INC COUNT ; Inkrementacja zmiennej pod adresem COUNT

MOV TOTAL, 48 ; Kopiuje liczbę pod adresem TOTAL

ADD AH, BH ; Dodanie treści z rejestru AH do BH i zapisuje w AH

AND MASK1, 128 ; Operacja AND na zmiennej MASK1

Segmenty pamięci

```
→ segment .text

    global _start

_start:
    mov    rax, 1          ; numer funkcji systemowej - sys_write
    mov    rdi, 1          ; numer pliku (1 == standardowe wyjście)
    mov    rsi, message   ; adres tekstu
    mov    rdx, size       ; długość tekstu
    syscall
    mov    rax, 60         ; numer funkcji systemowej - sys_exit
    syscall

→ segment .data

message db "Hello, World", 0xa ; napis wraz z znakiem nowej linii (10)
size    equ $ - message      ; długość napisu
```

Segmenty pamięci

```
segment .text

    global _start

_start:
    mov    rax, 1          ; numer funkcji systemowej - sys_write
    mov    rdi, 1          ; numer pliku (1 == standardowe wyjście)
    mov    rsi, message   ; adres tekstu
    mov    rdx, size       ; długość tekstu
    syscall
    mov    rax, 60         ; wywołanie funkcji systemowej
    syscall
    mov    rax, 60         ; numer funkcji systemowej - sys_exit
    syscall

segment .data

message db    "Hello, World", 0xa
size    equ    $ - message           ; napis wraz z znakiem nowej linii (10)
                                         ; długość napisu
```

OUTPUT:

Hello, World

Segmenty pamięci

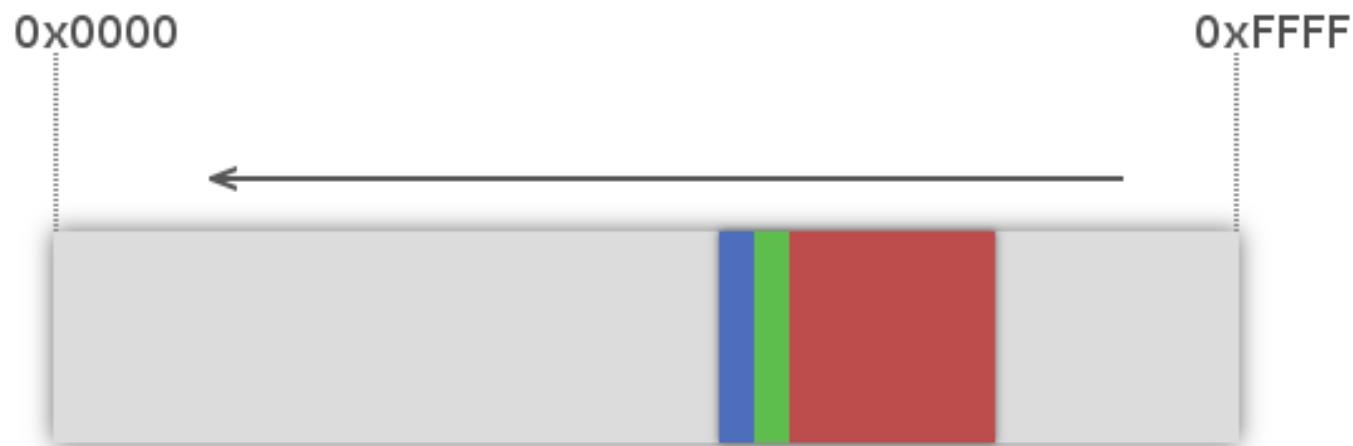
Segmenty pamięci – model podziału pamięci, który dzieli pamięć systemową na grupy niezależnych segmentów, do których możemy się dostać przez wskaźniki z specjalnych rejestrów

Segmenty pamięci:

- Segment kodu
- Segment danych
- Stos

Segmenty pamięci

Stos



sub dq rsp, 8
mov dq rsp, 3 → push dq 3

mov dq rax, rsp
add dq rsp, 8 → pop dq rax

Rejestry ogólnego przeznaczenia

Rejestr	Accumulator		Counter		Data		Base		
64-bit	RAX		RCX		RDX		RBX		
32-bit		EAX		ECX		EDX		EBS	
16-bit		AX		CX		DX		BS	
8-bit		A H	A L	C H	C L	D H	D L	B H	B L

Rejestr	Stack pointer		Stack base pointer		Source		Destination	
64-bit	RSP		RBP		RSI		RDI	
32-bit		ESP		EBP		ESI		EDI
16-bit		SP		BP		SI		DI
8-bit			S P L		B P L	S I L		D I L

Rejestry

Pozostałe rejesty:

- » **r8-15** – ogólnego przeznaczenia
- » **RIP** – wskaźnik na następną instrukcję do wykonania
- » **rejestry segmentowe** – aktualnie bardzo rzadko używane
- » **XMM** – rejesty o rozmiarze 128 bitów do operacji zmiennoprzecinkowych
- » **RFLAGS/EFLAGS** – rejestr flag

Rejestr FLAGS

Przykładowe flagi:

- » **CF** (carry flag - flaga przeniesienia),
- » **OF** (overflow flag - flaga przepelnienia),
- » **SF** (sign flag - flaga znaku),
- » **ZF** (zero flag - flaga zera),
- » **PF** (parity flag - flaga parzystosci).

Wywołania systemowe

Krok, po kroku:

- 1) Umieść numer wywołania systemowego w rejestrze *RAX*
- 2) Umieść argumenty wywołania systemowego w rejestrach:
RDI, RSI, RDX, R10, R8, R9
- 3) Wywołaj przerwanie *syscall*
- 4) Rezultat wywołania zwracany jest zazwyczaj do rejestrów *RAX*

Wywołania systemowe

```
segment .text

    global _start

_start:
    mov    rax, 1          ; numer funkcji systemowej - sys_write
    mov    rdi, 1          ; numer pliku (1 == standardowe wyjscie)
    mov    rsi, message   ; adres tekstu
    mov    rdx, size       ; dlugosc tekstu
    syscall
    mov    rax, 60          ; wywolanie funkcji systemowej
    syscall
    mov    rax, 60          ; numer funkcji systemowej - sys_exit
    syscall

segment .data

message db "Hello, World", 0xa      ; napis wraz z znakiem nowej linii (10)
size    equ $ - message             ; dlugosc napisu
```

Wywołania systemowe

%rax	Name	Entry point	Implementation
0	read	sys_read	fs/read_write.c
1	write	sys_write	fs/read_write.c
2	open	sys_open	fs/open.c
3	close	sys_close	fs/open.c
4	stat	sys_newstat	fs/stat.c

Tryby adresowania

- » Adresowanie natychmiastowe
- » Adresowanie bezpośrednie pamięci
- » Adresowanie pośrednie pamięci
- » Adresowanie bezpośredni rejestrów
- » Adresowanie pośrednie rejestrów

Adresowanie natychmiastowe^{AGH}

- » Operand bezpośredni to stałe wyrażenie, takie jak liczba, stała znakowa, wyrażenie arytmetyczne lub stała symboliczna

```
    mov ax, 1  
    mov ax, 010Ch  
    mov al, 'x'  
    mov al, (40 * 50)
```

Adresowanie bezpośrednie pamięci

- » Operand to etykieta określająca położenie danej w pamięci

mov rcx, zmienna

mov ax, liczba

Adresowanie pośrednie pamięci

- » Operand to etykieta określająca położenie danej w pamięci

```
mov rcx, [zmienna]
```

Adresowanie bezpośrednie rejestrowe

- » Operand to register of the processor containing the argument

```
mov rax, rbx
```

```
mov al, bl
```

Adresowanie pośrednie rejestrowe

- » Operand to rejestr procesora zawierający adres argumentu

```
mov rax, [rbx]
```

```
mov al, [bx]
```

Adresowanie z przesunięciem

Jak zapisać wartość danej znajdującej się pod adresem przesuniętym o 8 bajtów od adresu w rejestrze rbx?

mov rax, [rbx + 8]

Jak zapisać adres przesunięty o 8 bajtów od adresu rejestrów rbx?

~~mov rax, rbx + 8~~ **ŽLE!**

Pobieranie adresu efektywnego

```
mov rax, rbx  
add rax, 8
```

```
lea rax, [rbx + 8]
```

Zmienne i stałe

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

Dane zainicjalizowane:

Dyrektyna	Definicja	Przestrzeń
DB	bajt	1 bajt
DW	słowo	2 bajty
DD	podwójne słowo	4 bajty
DQ	poczwórne słowo	8 bajtów
DT	dziesięć bajtów	10 bajtów

Zmienne i stałe

~~-duza_liczba~~ dt ~~6af4aD8b4a43ac4d33h~~

duza_liczba dd ~~43ac4d33h, 0f4aD8b4ah~~

~~6gamma~~ db 9

gamma db 4

Zmienne i stałe

Dane niezainicjalizowane:

Dyrektywa	Cel
RESB	Zarezerwuj bajt
RESW	Zarezerwuj słowo
RESD	Zarezerwuj podwójne słowo
RESQ	Zarezerwuj poczwórne słowo
REST	Zarezerwuj dziesięć bajtów

Zmienne i stałe

TIMES - dyrektywa umożliwiająca wielokrotne zainicjalizowanie wielu bajtów, tą samą wartością

```
section .data
stars    times 9 db '*'
array    times 5 dw 0
```

Zmienne i stałe

Istnieje kilka dyrektyw dostarczonych przez NASM, deklarujących stałe.

Omówione zostaną:

- EQU
- %assign
- %define

Zmienne i stałe

EQU – wykorzystywana do definiowania stałych

CONSTANT_NAME EQU expression

Przykład:

```
TOTAL_STUDENTS equ 50
```

```
section .text
    global _start

_start:
    mov  ecx, TOTAL_STUDENTS
    cmp  eax, TOTAL_STUDENTS
```

Zmienne i stałe

%assign – wykorzystywana do definiowania stałych, pozwala na redefinicję

```
%assign CONSTANT_NAME expression
```

Przykład:

```
%assign TOTAL 25
```

```
...
```

```
%assign TOTAL 44
```

Zmienne i stałe

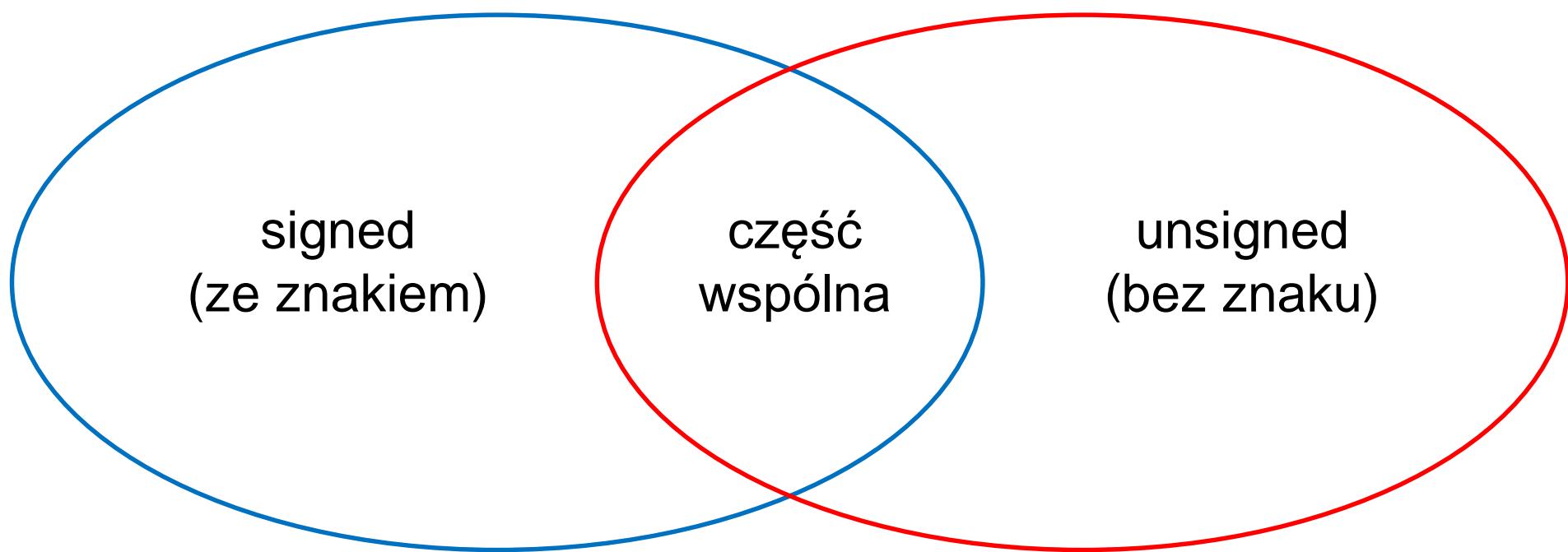
%define – wykorzystywana do definiowania numerycznych, łańcuchowych,
podobna do #define z C

%define CONSTANT_NAME expression

Przykład:

%define PTR [EBR + 4]

Operacje arytmetyczne



Operacje arytmetyczne

» **inc** – inkrementacja

inc *rax*

inc *[zmienna]*

» **dec** – dekrementacja

dec *rax*

dec *[zmienna]*

Operacje arytmetyczne

» **add/sub** – dodawanie/odejmowanie

add/sub dest, src

add rax, rcx

add rax, [zmienna]

sub rsp, 8

Operacje arytmetyczne

» **mul** – mnożenie (liczb bez znaku)

mul mnoznik

mul dl ; ax = al * dl

mul dx ; dx:ax = ax * dx

mul edx ; edx:eax = eax * edx

mul rdx ; rdx:rax = rax * rdx

Operacje arytmetyczne

» **imul** – mnożenie (liczb ze znakiem)

```
imul rdx
imul rdx, rcx
imul rdx, 2
imul rdx, rcx, 2
```

Operacje arytmetyczne

- » **div** – dzielenie (liczb bez znaku)
- » **idiv** – dzielenie (liczb ze znakiem)

div/idiv dzielnik

div dl

div dx

div edx

div rdx

Operacje arytmetyczne

» **shr** – przesunięcie bitowe w prawo

shr dl, 2

10011101

00100111

» **shl** – przesunięcie bitowe w lewo

shl dl, 2

00001010

00101000

Operacje arytmetyczne

» **sar** – przesunięcie arytmetyczne w prawo

```
sar dl, 2
```

```
10011101  
11100111
```

» **sal** – przesunięcie arytmetyczne w lewo

```
sal dl, 2
```

```
00001010  
00101000
```

Instrukcje logiczne

Operatory logiczne:

- AND - AND operand1, operand2
- OR - OR operand1, operand2
- XOR - XOR operand1, operand2
- TEST - TEST operand1, operand2
- NOT - NOT operand1

Instrukcje logiczne

Flaga	Opis
CF	Flaga przeniesienia
OF	Flaga przepelnienia
PF	Flaga parzystosci
SF	Flaga znaku
ZF	Flaga zera

Skok bezwarunkowy

jmp etykieta

...

etykieta:

sub rax, rcx

Instrukcja porównania

- » Oblicza różnicę i ustawia odpowiednie flagi

```
cmp rax, rcx
```

- » Wykonuje iloczyn logiczny i ustawia odpowiednie flagi

```
test rax, rax
```

Skoki warunkowe

Instrukcja	Rozwinięcie nazwy (ang)	Sprawdzane flagi
JE/JZ	Jump Equal or Jump Zero	ZF
JNE/JNZ	Jump not Equal or Jump Not Zero	ZF
JG/JNLE	Jump Greater or Jump Not Less/Equal	OF, SF, ZF
JGE/JNL	Jump Greater/Equal or Jump Not Less	OF, SF
JL/JNGE	Jump Less or Jump Not Greater/Equal	OF, SF
JLE/JNG	Jump Less/Equal or Jump Not Greater	OF, SF, ZF

Skoki warunkowe

Instrukcja	Rozwinięcie nazwy (ang)	Sprawdzane flagi
JE/JZ	Jump Equal or Jump Zero	ZF
JNE/JNZ	Jump not Equal or Jump Not Zero	ZF
JA/JNBE	Jump Above or Jump Not Below/Equal	CF, ZF
JAE/JNB	Jump Above/Equal or Jump Not Below	CF
JB/JNAE	Jump Below or Jump Not Above/Equal	CF
JBE/JNA	Jump Below/Equal or Jump Not Above	AF, CF

Pętle

```
mov rcx, 10
petla:
...
dec rcx
jnz petla
```

```
mov rcx, 10
petla:
...
loop petla
```

Łańcuchy znaków, tablice

Długość łańcucha znaków:

- Jawne przechowywanie długości

```
msg  db  'Hello, world!',0xa ;string
len  equ  $ - msg           ;rozmiar stringu
;lub
len  equ  13                ;rozmiar stringu
```

- Używanie znaku wartownika

Łańcuchy znaków, tablice

Instrukcje do operacji na łańcuchach znaków:

- MOVS – przenosi element z zadanej lokalizacji do innej
- CMPS – porównuje dwa elementy danych w pamięci
- SCAS – porównuje zawartość rejestru RAX z elementem w pamięci
- LODS – instrukcja ładująca z pamięci w zależności od operandu
- STOS – przechowuje dane z rejestru RAX w pamięci

Łańcuchy znaków, tablice

Instrukcje	Operacje na	Operacja bajtowa	Operacja na słowie	Operacja na dwóch słowach	Operacja na danych 64-bitowych
MOVS	RDI RSI	MOVSB	MOVSW	MOVSD	MOVSQ
LODS	RAX RSI	LODSB	LODSW	LODSD	LODSQ
STOS	RDI RAX	STOSB	STOSW	STOSD	STOSQ
CMPS	RSI RDI	CMPSB	CMPSW	CMPSD	CMPSQ
SCAS	RDI RAX	SCASB	SCASW	SCASD	SCASQ

Łańcuchy znaków, tablice

Prefix **REP** – powoduje powtarzanie instrukcji

```
mov rsi, zrodlo  
mov rdi, cel  
cld          ; idz do przodu  
mov rcx, 128  
rep movsb
```

Łańcuchy znaków, tablice

REP – odmiany:

- REP - bezwarunkowe powtórzenie (aż CX == 0)
- REPE / REPZ - warunkowe powtórzenie (aż CX == 0 lub ZF nie wskazuje 0)
- REPNE / REPNZ - warunkowe powtórzenie (aż CX == 0 lub ZF wskazuje 0)

Łańcuchy znaków, tablice

Różne deklaracje tablicy:

```
INVENTORY      DW  0  
                DW  0  
                DW  0
```

```
INVENTORY      DW  0,  0,  0
```

```
INVENTORY      TIMES 3 DW  0
```

Adres symboliczny poniższej tablicy to *NUMBERS*, natomiast adres n-tego elementu to *NUMBERS + 2*n*

```
NUMBERS DW  34,  45,  56,  67,  75,  89
```

Procedury

Język Assembly pozwala na definiowanie procedur lub podprogramów

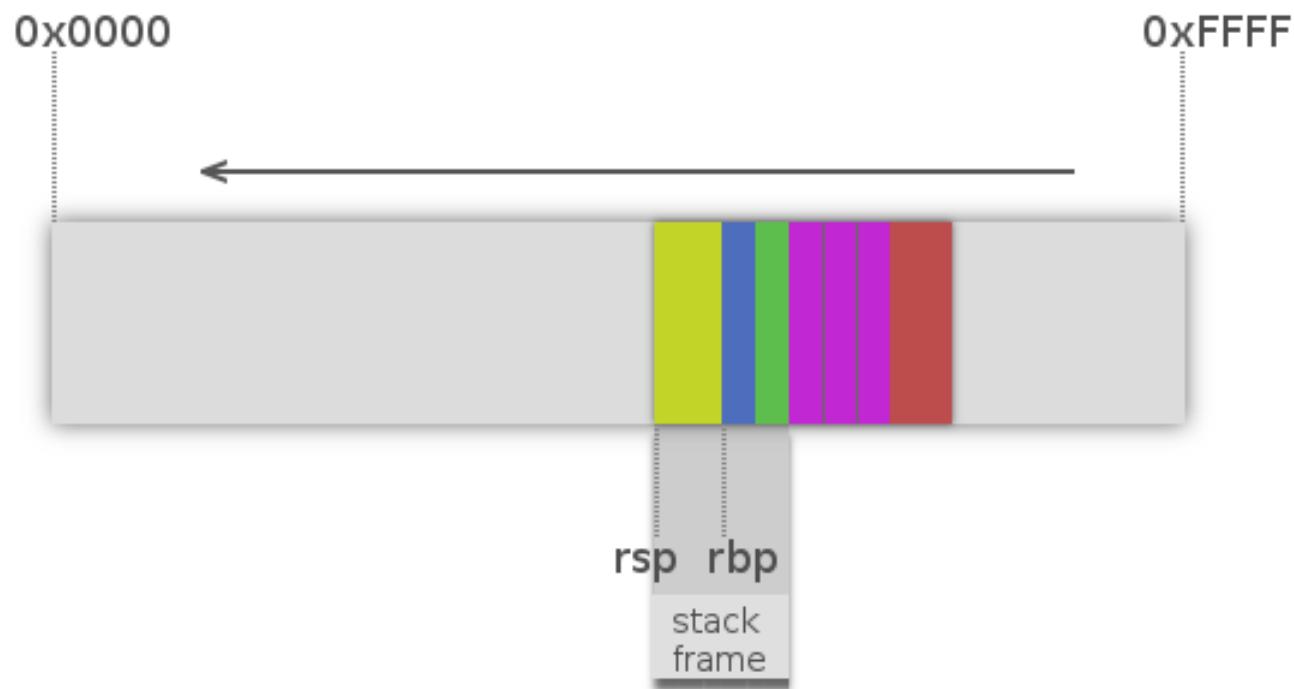
- Składnia

```
proc_name:  
    procedure body  
    ...  
    ret
```

- Wywołanie z innej funkcji

```
CALL proc_name
```

Procedury – ramka stosu



Procedury

Assembler:

```
myFun(int, int, int, int, int, int, int, int):
```

push	rbp
mov	rbp, rsp
mov	DWORD PTR [rbp-4], edi
mov	DWORD PTR [rbp-8], esi
mov	DWORD PTR [rbp-12], edx
mov	DWORD PTR [rbp-16], ecx
mov	DWORD PTR [rbp-20], r8d
mov	DWORD PTR [rbp-24], r9d
mov	DWORD PTR [rbp+32], 1234
mov	eax, 33
pop	rbp
ret	

C++:

int myFun (int _1, int _2, int _3, int _4, int _5, int _6, int _7, int _8, int _9) {
_9 = 1234;
return 33;
}
void funkcjaA () {
myFun(1, 2, 3, 4, 5, 6, 7, 8, 9);
}

funkcjaA():

push	rbp
mov	rbp, rsp
push	9
push	8
push	7
mov	r9d, 6
mov	r8d, 5
mov	ecx, 4
mov	edx, 3
mov	esi, 2
mov	edi, 1
call	myFun(int, int, int, int, int, int, int, int)
add	rsp, 24
nop	
leave	
ret	

Konwencje wywołania

- » W jaki sposób będą przekazywane parametry do procedury?
- » Które rejesty mają być zachowane a które można zmieniać?
- » Kto jest odpowiedzialny za sprzątanie stosu?
- » https://www.agner.org/optimize/calling_conventions.pdf

Konwencje wywołania

- » cdecl
- » stdcall
- » fastcall
- » System V AMD64
- » i więcej...

cdecl

- » Parametry na stosie (RLO),
- » Wynik w EAX,
- » Sprząta „caller” (wywołujący)
- » ret – bez argumentów

cdecl - przykład

```
; b = f(a, 5)
mov rax, 5
push rax
mov rax, [a]
push rax
call f

add rsp, 16
mov [b], eax
```

```
int f(int a, int b) {
    int c = a+b;
    return c;
}
```

```
f:
push rbp
mov rbp, rsp
sub rsp, 8

mov eax, [rbp+16]
add eax, [rbp+24]
mov [rbp-8], eax

mov rsp, rbp
pop rbp
ret
```

stdcall

- » Parametry na stosie (RLO),
- » Wynik w EAX,
- » Sprząta „callee” (wywoływany)
- » retn X/ret X – przesuwa wskaźnik stosu

fastcall

- » Brak standardu
- » Parametry w rejestrach
- » Gdy więcej parametrów – wrzucane na stos
- » Sprząta „callee” (wywoływany)

System V AMD64

- » Parametry w rejestrach RDI, RSI, RDX, RCX, R8, R9, [XYZ]MM0-7
- » Gdy więcej parametrów – wrzucane na stos (RLO)
- » Sprząta „caller” (wywołujący)

Makra

Makro – sekwencja instrukcji przypisana przez nazwę, może być użyte w dowolnym miejscu w programie

- Składnia

```
%macro macro_name  number_of_params  
<macro body>  
%endmacro
```

Makra

```
; A macro with two parameters
%macro write_string 2
    mov    rax, 1
    mov    rdi, 1
    mov    rsi, %1
    mov    rdx, %2
    syscall
%endmacro

section .text
global _start

_start:
    write_string msg1, len1
    write_string msg2, len2
    write_string msg3, len3

    mov    rax, 60
    syscall

section .data
msg1 db 'Hello, programmers!', 0xA, 0xD
len1 equ $- msg1

msg2 db 'Welcome to the world of, ', 0xA, 0xD
len2 equ $- msg2

msg3 db 'Linux assembly programming! '
len3 equ $- msg3
```

Godbolt.org – Compiler Explorer

- » Wstęp
- » <https://godbolt.org/z/Xt55Mq>
- » Dziedziczenie vs agregacja vs płaska struktura
- » <https://godbolt.org/z/t8KpF6>
- » <https://godbolt.org/z/tS3oZc>
- » <https://godbolt.org/z/UnWTDa>
- » Wyrażenia lambda
- » <https://godbolt.org/z/amWqok>
- » <https://godbolt.org/z/FGifch>
- » <https://godbolt.org/z/MpAqSf>

<https://cpp-polska.pl/post/czy-c-jest-wolniejszy-od-cij-kilka-slow-o-zero-cost-abstraction>

Manuale Intela

<https://software.intel.com/en-us/articles/intel-sdm>

Bibliografia

- » Asembler – Wikipedia [online]
<https://pl.wikipedia.org/wiki/Asembler>
[dostęp: 10 marca 2020]
- » x86 assembly language – Wikipedia [online]
https://en.wikipedia.org/wiki/X86_assembly_language
[dostęp: 10 marca 2020]
- » x86 Assembly – Wikibooks [online]
https://en.wikibooks.org/wiki/X86_Assembly
[dostęp: 10 marca 2020]
- » x64 Architecture – Windows drivers [online]
<https://docs.microsoft.com/en-us/windows-hardware/drivers/debugger/x64-architecture?redirectedfrom=MSDN>
[dostęp: 10 marca 2020]
- » Programowanie niskopoziomowe [online]
<http://ww2.ii.uj.edu.pl/~kapela/pn/print-lecture-and-sources.php>
[dostęp: 10 marca 2020]

Dziękujemy za uwagę!