Wprowadzenie
do wysokowydajnych komputerów

Tadeusz Tomczak
tadeusz.tomczak@pwr.wroc.pl
pok. 201 bud. C3

Po co?

- Wydajność (ang.performance)
  - nie mylić z prędkością, ...
- Bezpieczeństwo (ang. security)
  - nie mylić z niezawodnością, ...
- Poprawność (and. correctness)
  - nie mylić z dokładnością, …
- Żeby zrozumieć artykuły:

  https://www.anandtech.com/show/17585/amd-zen-4-ryzen-9-7950x-and-ryzen-5-7600x-review-retaking-the-high-end/8

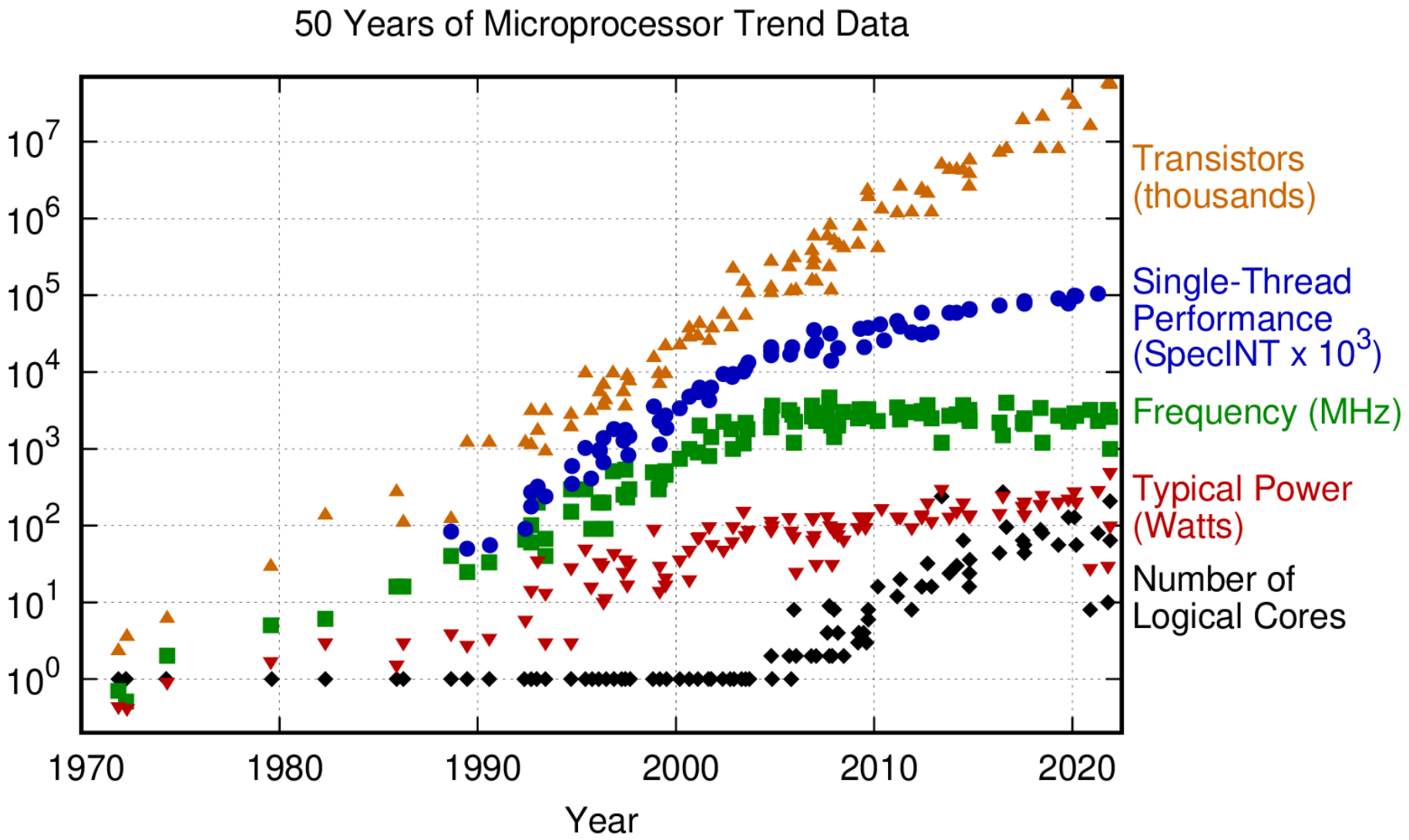
  https://chipsandcheese.com/2022/11/05/amds-zen-4-part-1-frontend-and-execution-engine/

  https://www.tomshardware.com/reviews/arm-cortex-a72-architecture,4424.html

  https://www.anandtech.com/show/14514/examining-intels-ice-lake-microarchitecture-and-sunny-cove/3

  ...

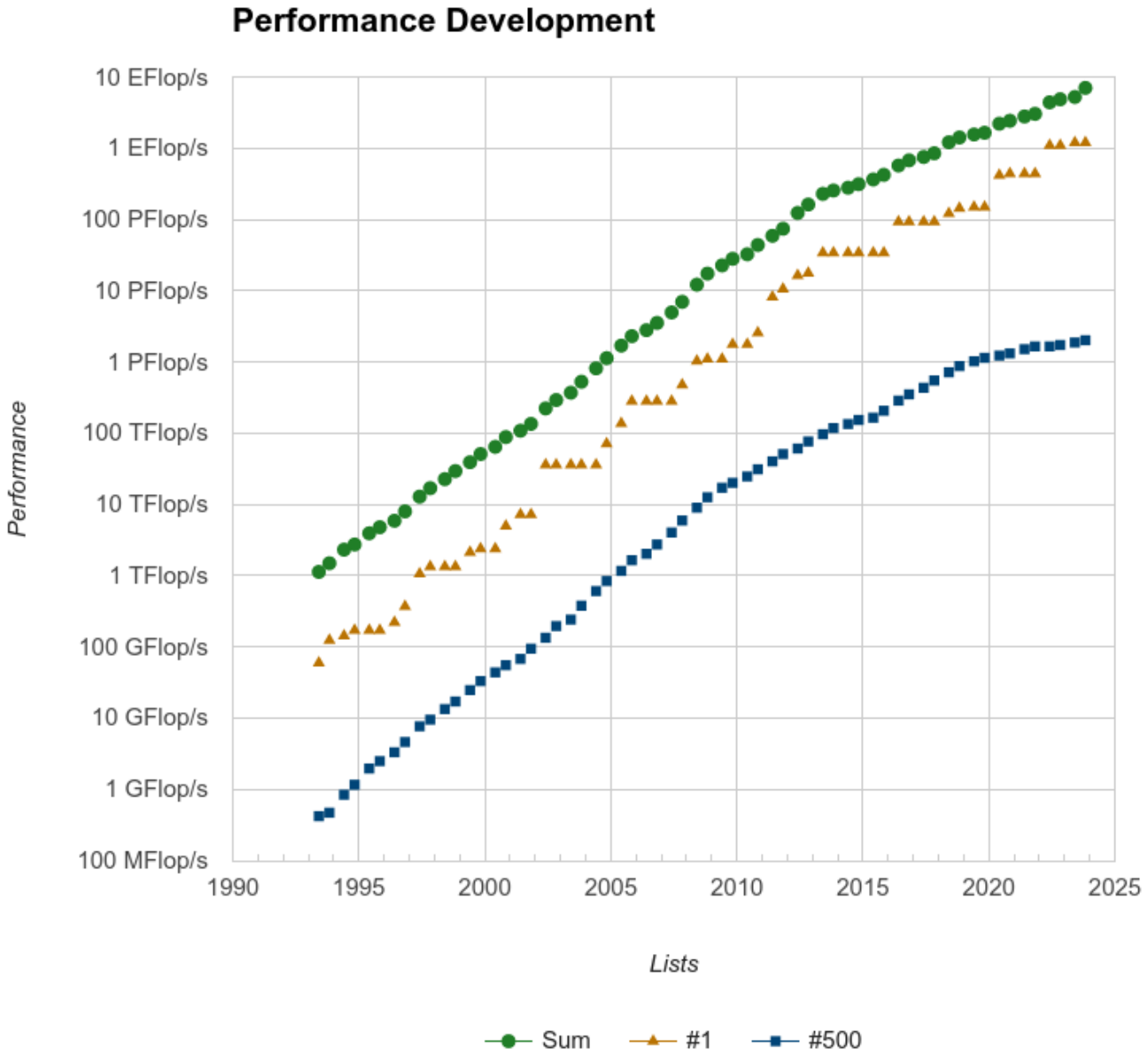
Po co?



Original data up to the year 2010 collected and plotted by M. Horowitz, F. Labonte, O. Shacham, K. Olukotun, L. Hammond, and C. Batten
New plot and data collected for 2010-2021 by K. Rupp

źródło: https://github.com/karlrupp/microprocessor-trend-data

Po co?



źródło: https://www.top500.org/statistics/perfdevel/

Po co?

```
$ lscpu
...
Vulnerability Itlb multihit:     KVM: Mitigation: Split huge pages
Vulnerability L1tf:              Mitigation; PTE Inversion; VMX
conditional cache flushes, SMT vulnerable
Vulnerability Mds:               Mitigation; Clear CPU buffers; SMT
vulnerable
Vulnerability Meltdown:          Mitigation; PTI
Vulnerability Spec store bypass: Mitigation; Speculative Store Bypass
disabled via prctl and seccomp
Vulnerability Spectre v1:        Mitigation; usercopy/swapgs barriers
and __user pointer sanitization
Vulnerability Spectre v2:        Mitigation; Full generic retpoline,
IBPB conditional, IBRS_FW, STIBP conditional, RSB filling
Vulnerability Tsx async abort:   Not affected
```

Komputer z programem przechowywanym
(*ang. program-stored computer*)

Komputer – uproszczone definicje

- Podstawowe zadanie: wykonywanie obliczeń
- Program: *zazwyczaj** sformalizowany opis sposobu wykonywania działań

  Programy = algorytmy + struktury danych
- Algorytm: „przepis” w postaci ciągu operacji na określonych reprezentacjach i strukturach danych określających uporządkowanie tych danych
- Dane: reprezentacja w postaci symboli/znaków

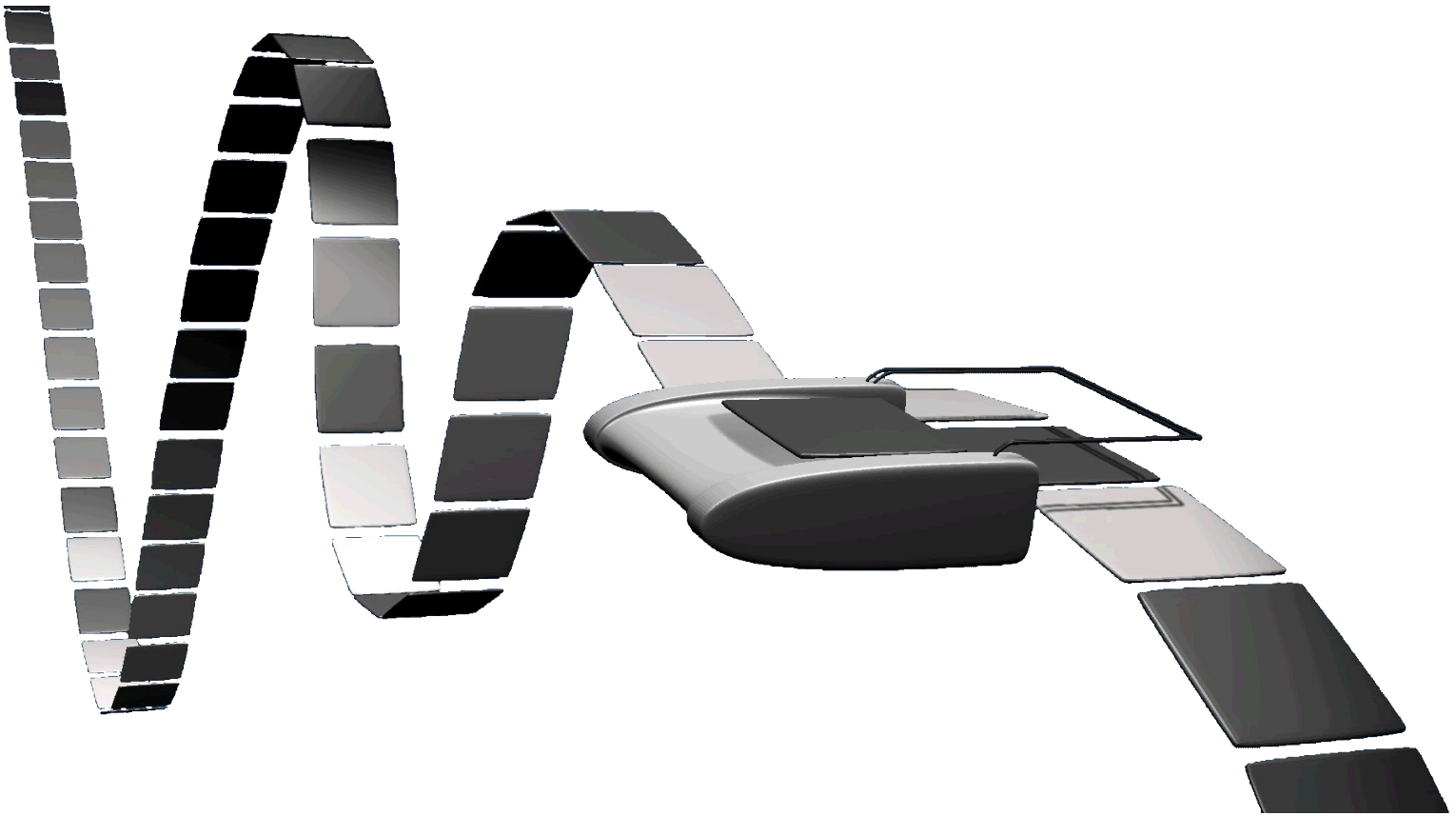
*Pomijając np. deklaratywne paradygmaty programowania

Programowanie imperatywne

Program - sekwencja rozkazów (instrukcji) zmieniających stan programu (zawartość rejestrów i pamięci).

Języki: kod maszynowy, język asemblera, Fortran, Algol, C, Pascal, ...

Maszyna Turinga



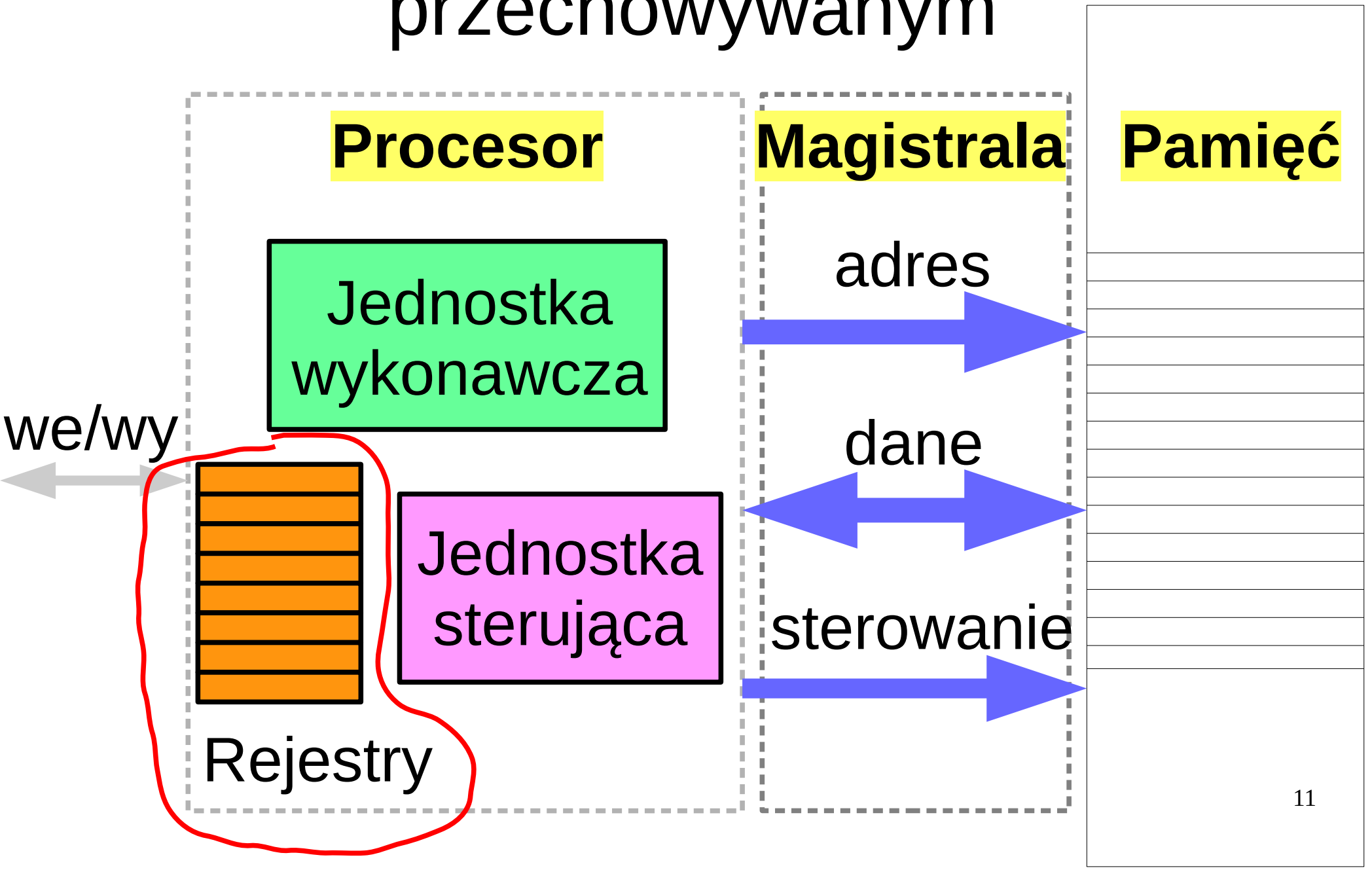
Prosty model abstrakcyjny

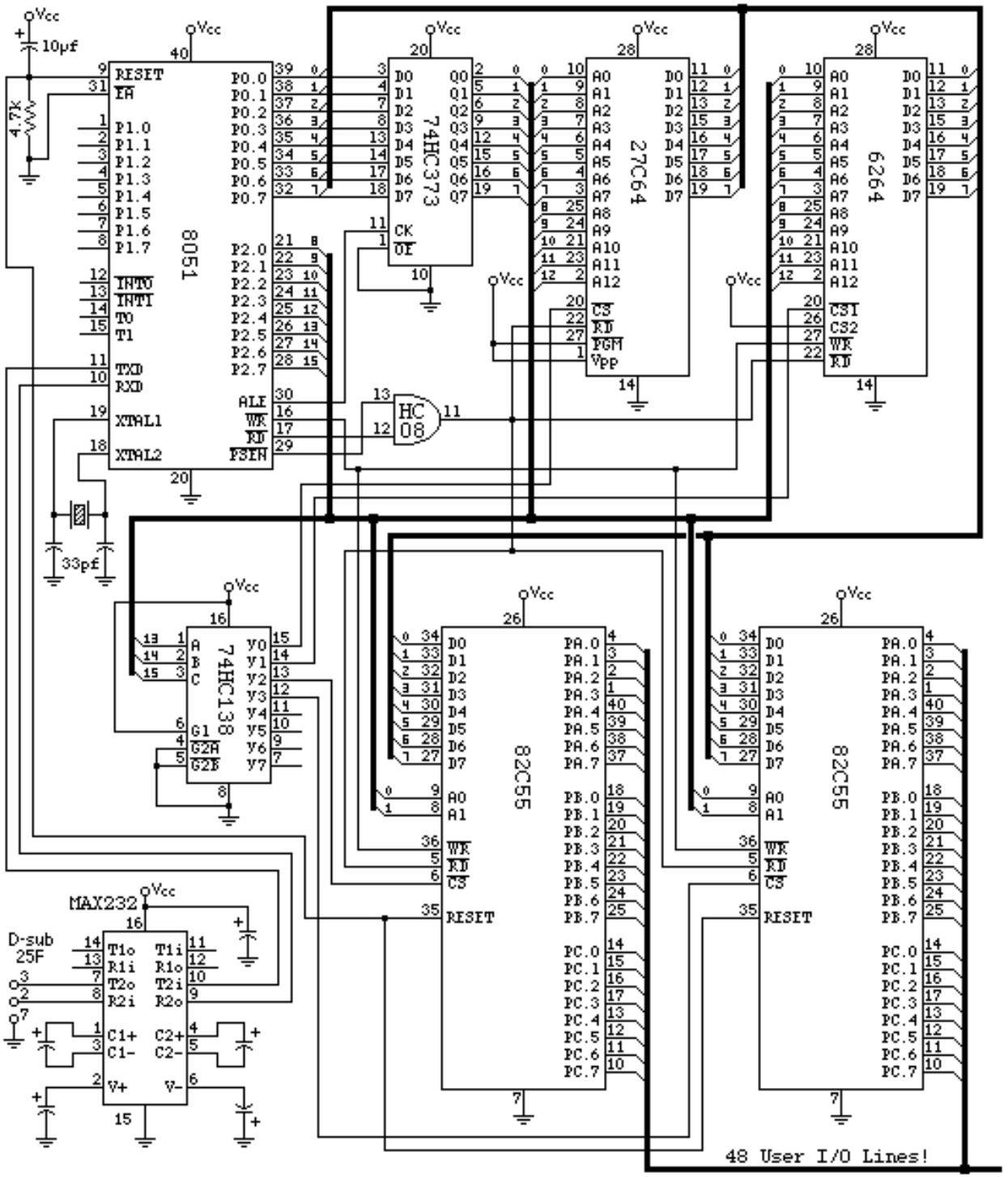
źródło: Wikimedia Commons

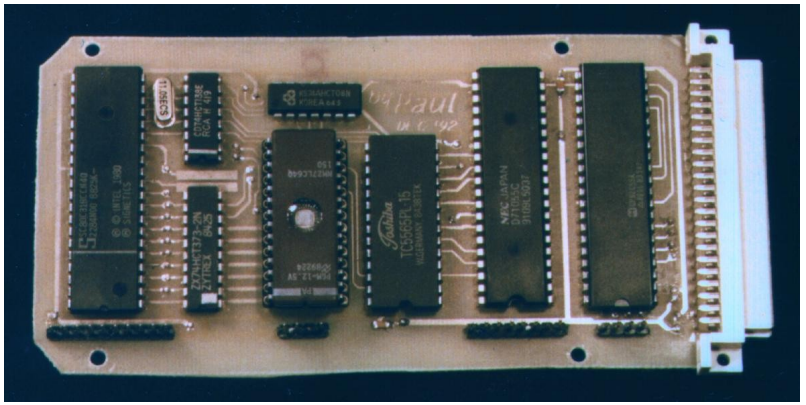
Maszyna Turinga

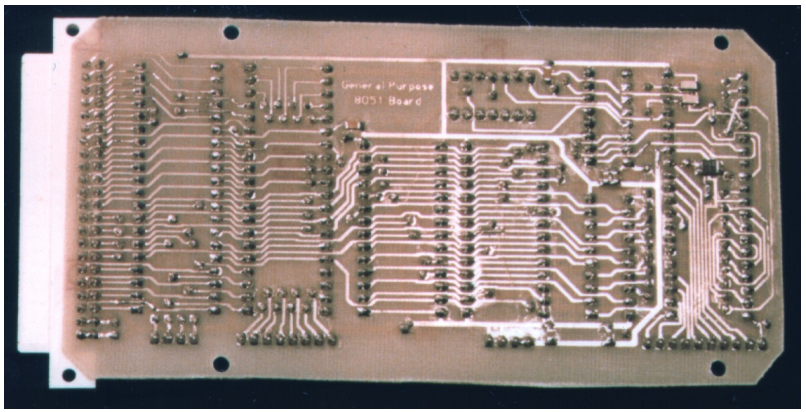
- taśma – pamięć
- głowica – procesor
- stany maszyny – zawartość rejestrów procesora
- rozkazy maszyny – instrukcje procesora

Komputer z programem przechowywanym







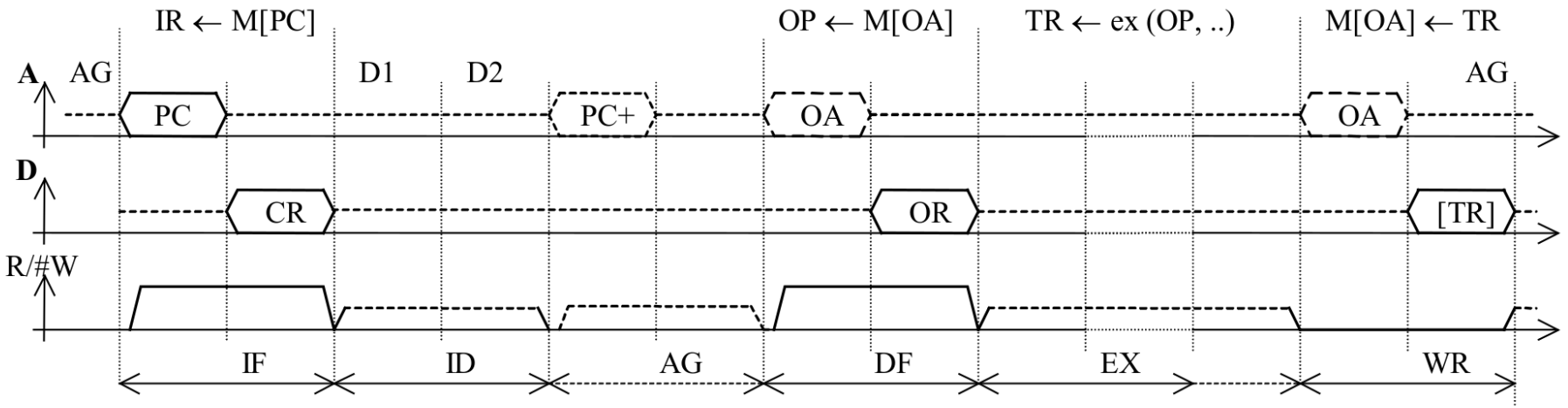


źródło: https://www.pjrc.com/tech/8051/dev-board-schematic.html

Komputer z programem przechowywanym

- procesor odczytuje/zapisuje dane z/do pamięci
- odczytane dane są interpretowane jako instrukcje lub argumenty instrukcji
  - architektura harwardzka – rozdzielone pamięci instrukcji i danych
- domyślnie instrukcje są odczytywane sekwencyjnie
- możliwa jawna zmiana adresu odczytywanych instrukcji (instrukcje rozgałęzień)

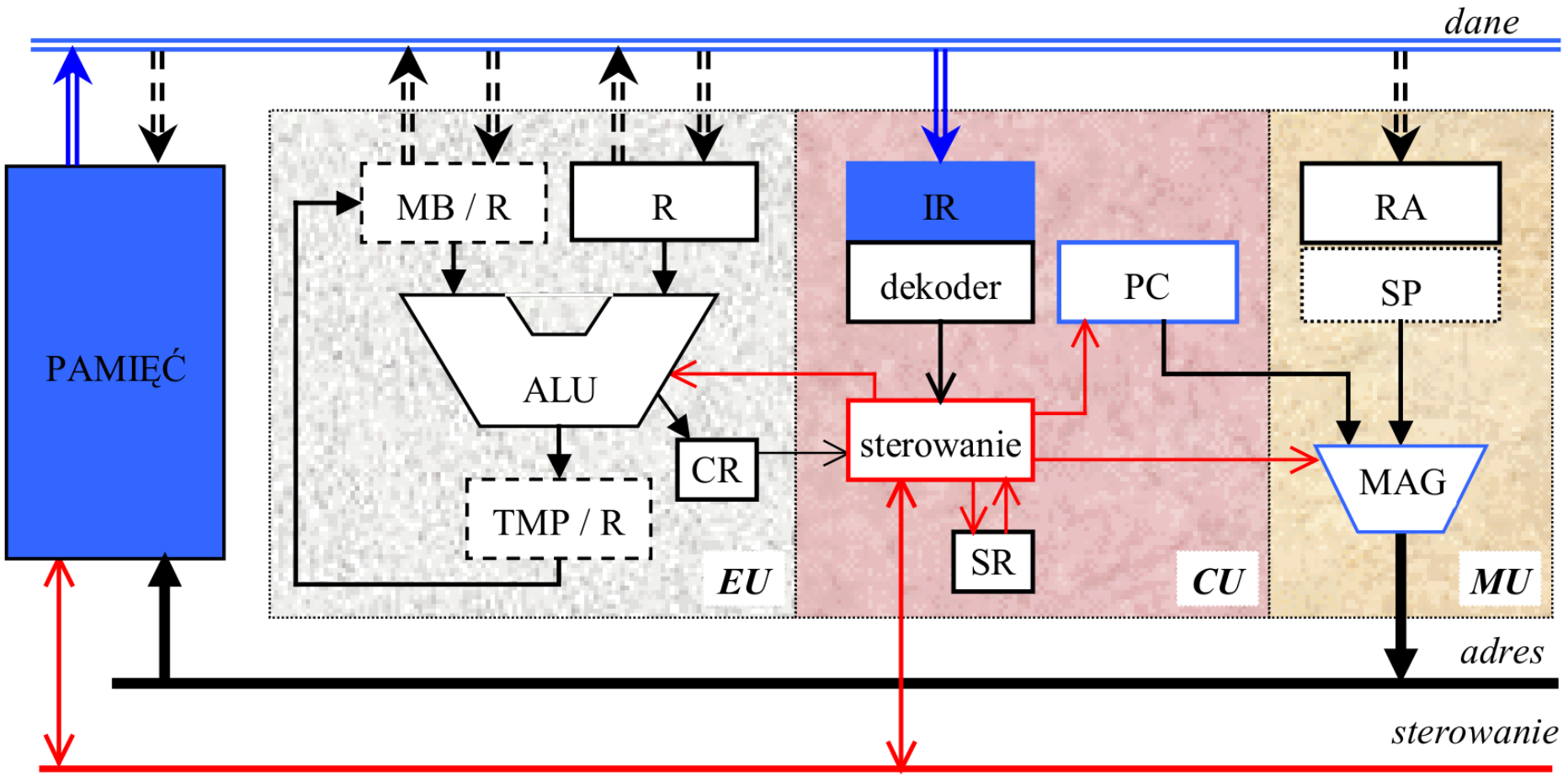
Cykl rozkazowy (CISC)



(architektura R/M)

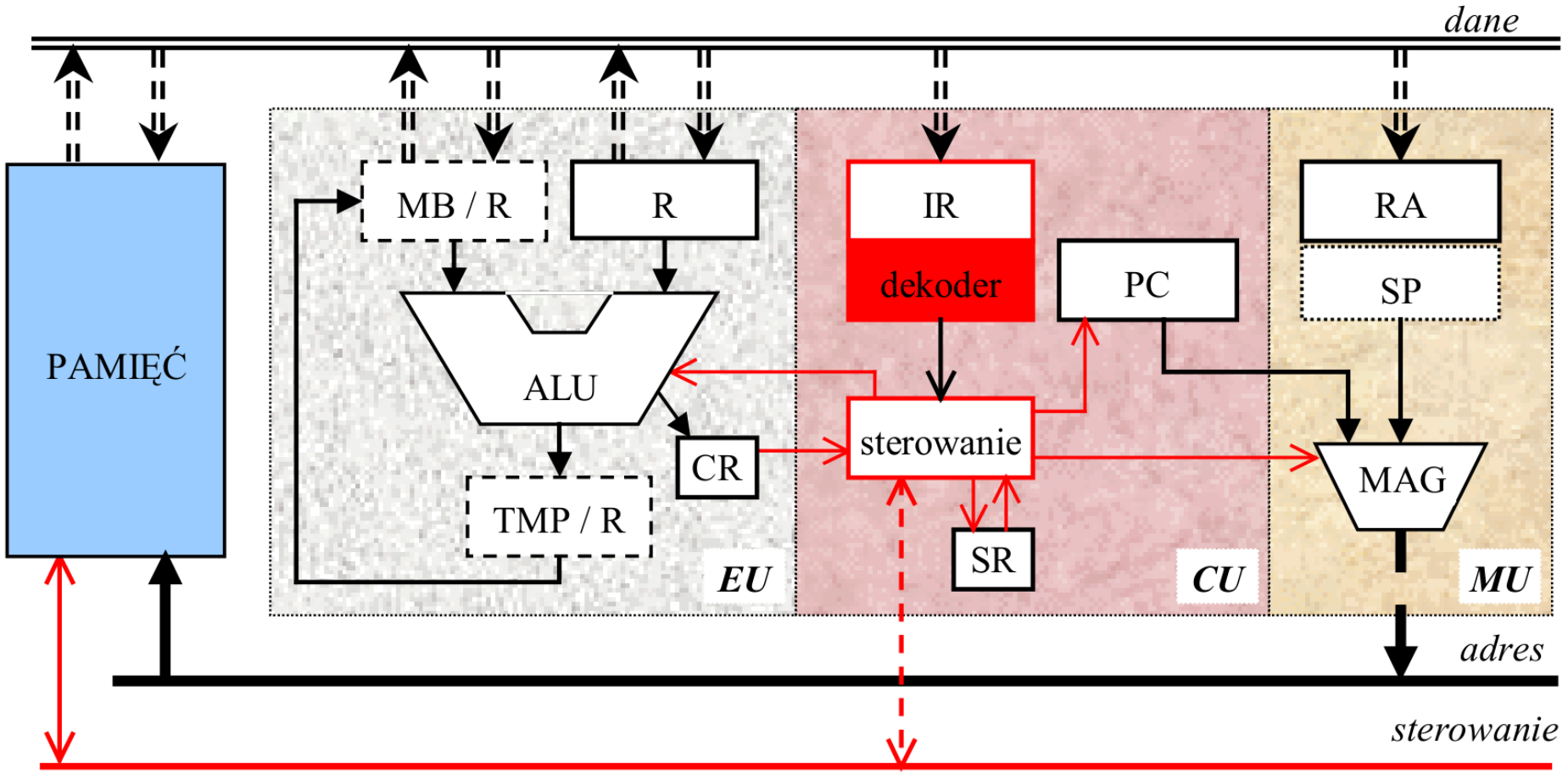
- pobranie kodu rozkazu z pamięci do rejestru rozkazów (ang. *instruction fetch*, IF)
- dekodowanie zawartości rejestru rozkazów (ang. *instruction decode*, ID)
- wytworzenie adresu operandu (ang. *address generation*, AG)
- pobranie operandów z pamięci (ang. *data fetch*, DF)
  - pobranie adresu skoku (ang. *target instruction address*, TA)
- wykonanie (ang. *execute*, EX)
- zapis wyniku (ang. *write*, WR) do pamięci (M[OA]) lub do rejestru (ang. *put away*)

Organizacja procesora sekwencyjnego (F)



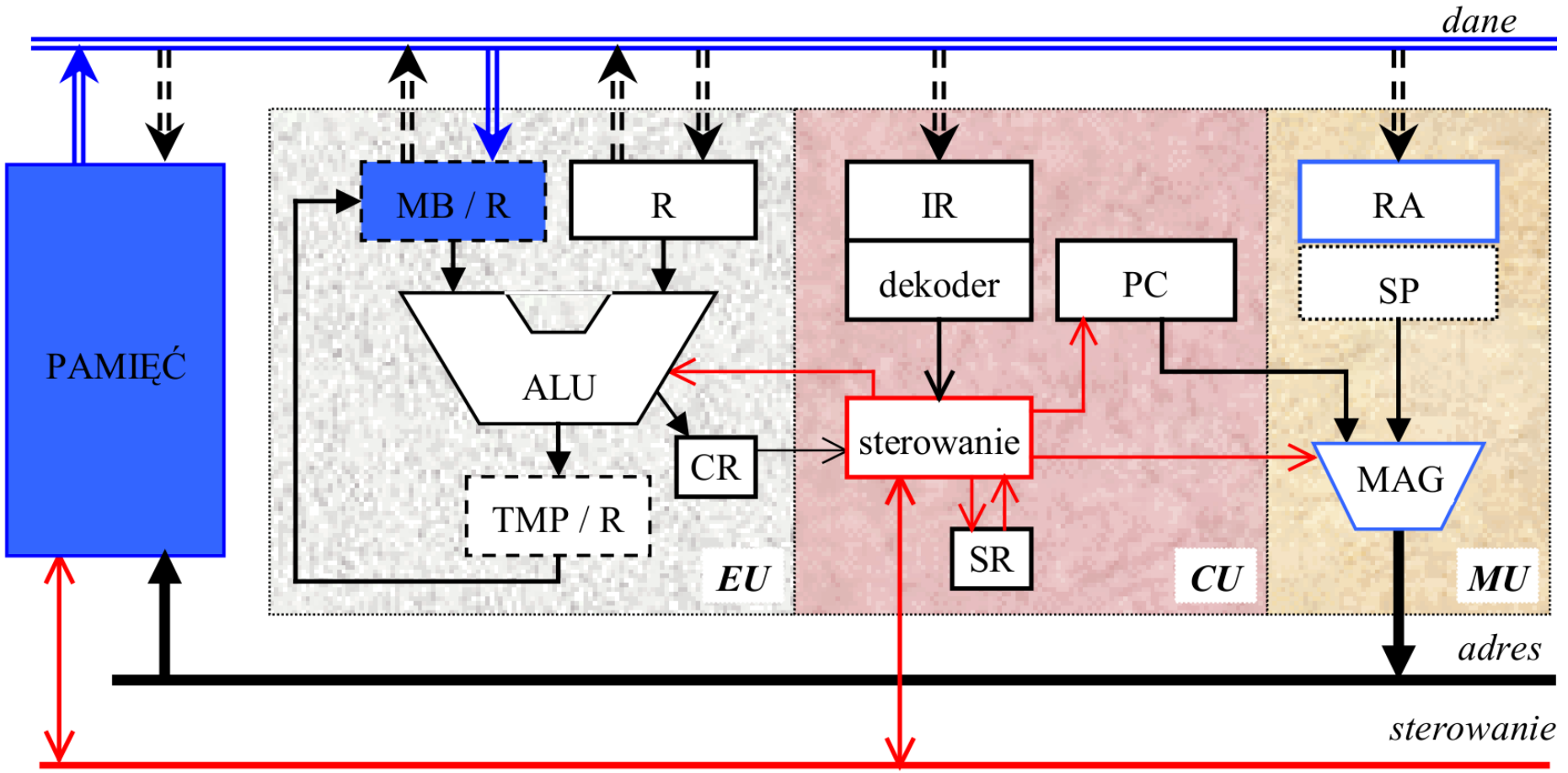
- *F* (pobranie kodu): adres (PC) → pamięć → rejestr rozkazów IR

Organizacja procesora sekwencyjnego (D)



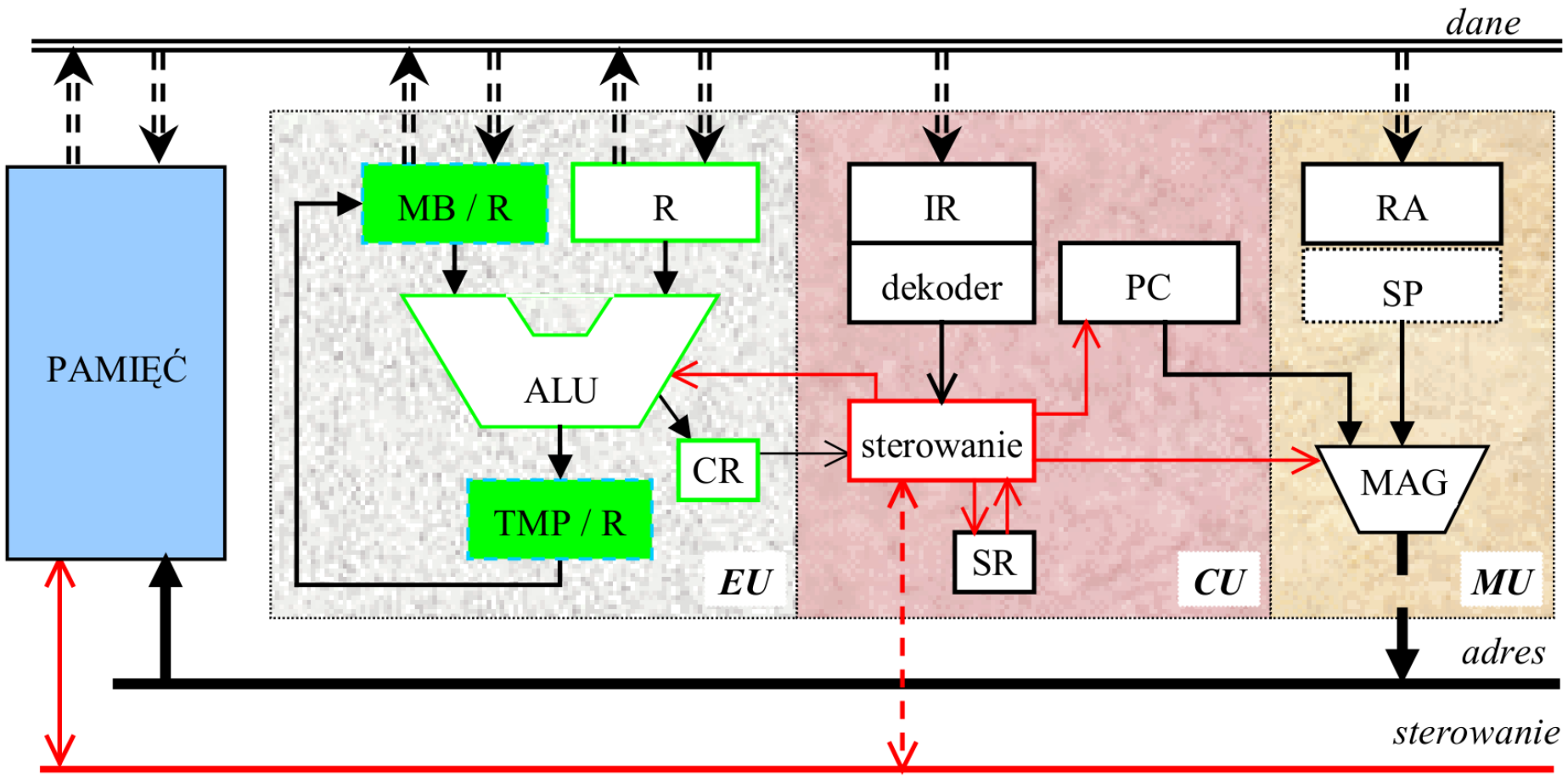
- *D* (dekodowanie) : rejestr rozkazów IR → sterowanie (CR,SR)

Organizacja procesora sekwencyjnego (R)



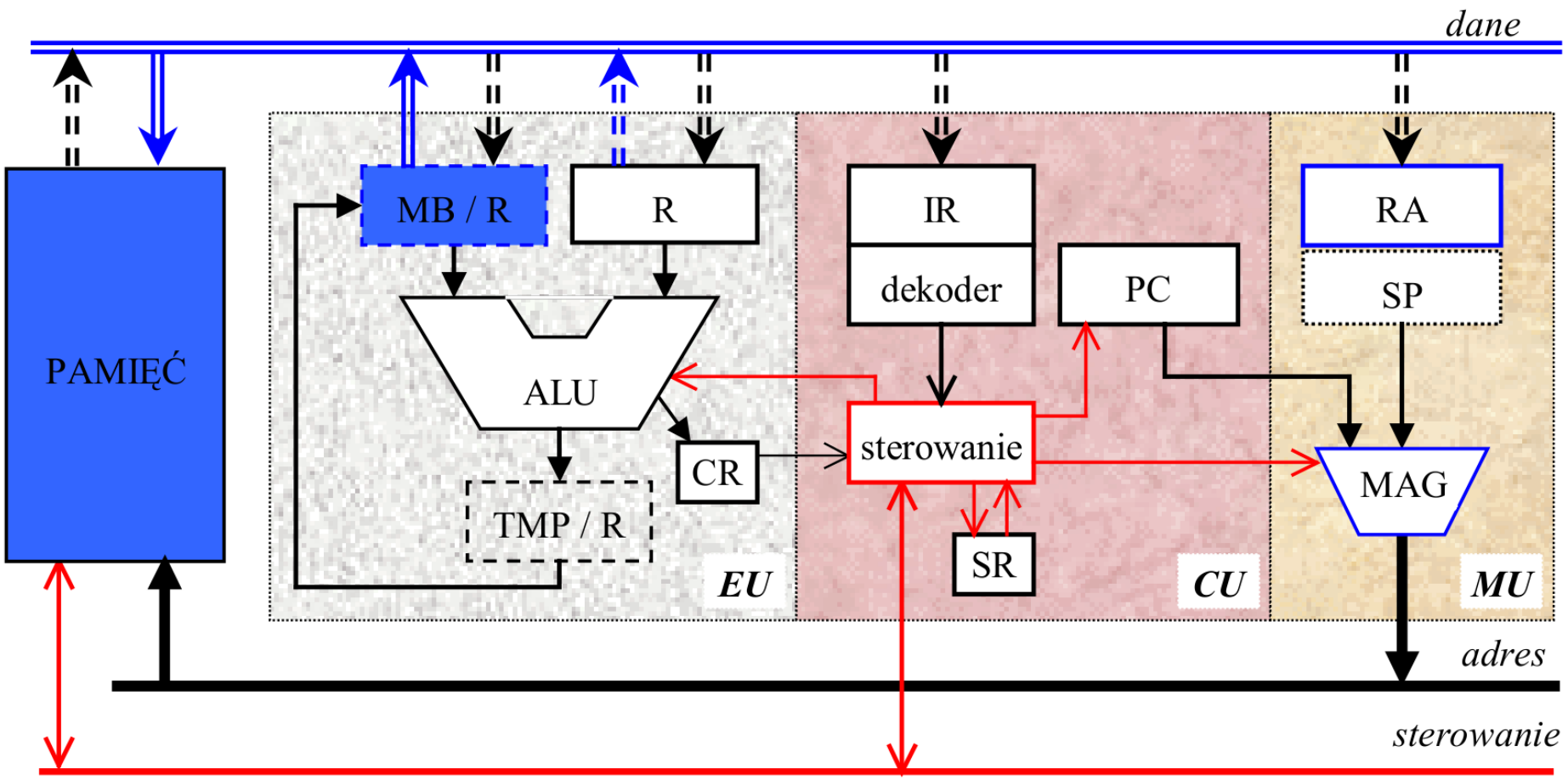
- *R* (odczyt danej): adres (RA) → pamięć → bufor (rejestr) MB

Organizacja procesora sekwencyjnego (E)



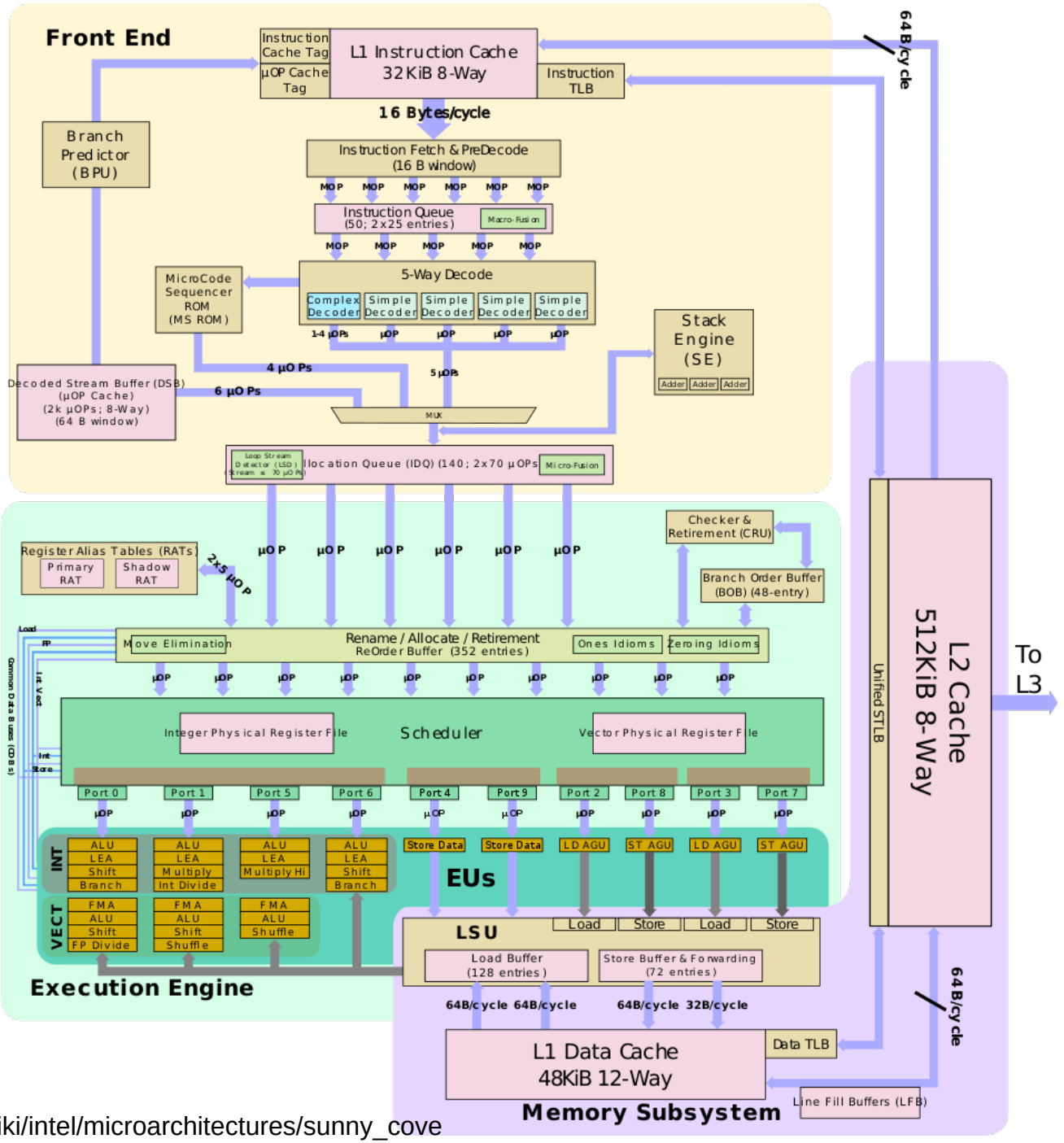
- *E* (wytworzenie wyniku): ALU (MB/R, R) → bufor TMP/ rejestr R

Organizacja procesora sekwencyjnego (W)



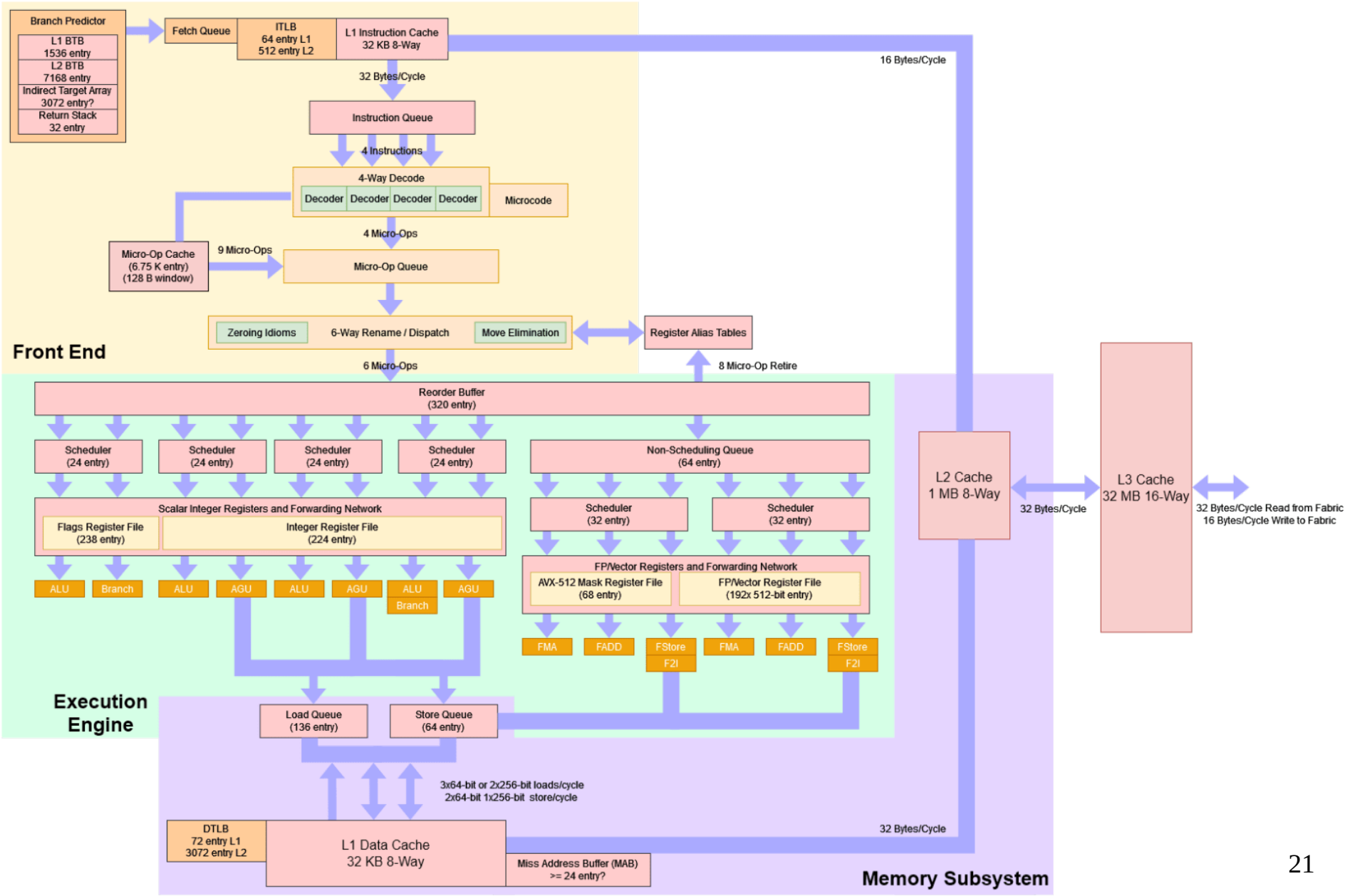
- *W* (zapis wyniku): bufor MB → pamięć (adres (RA))

Sunny Cove (2019)



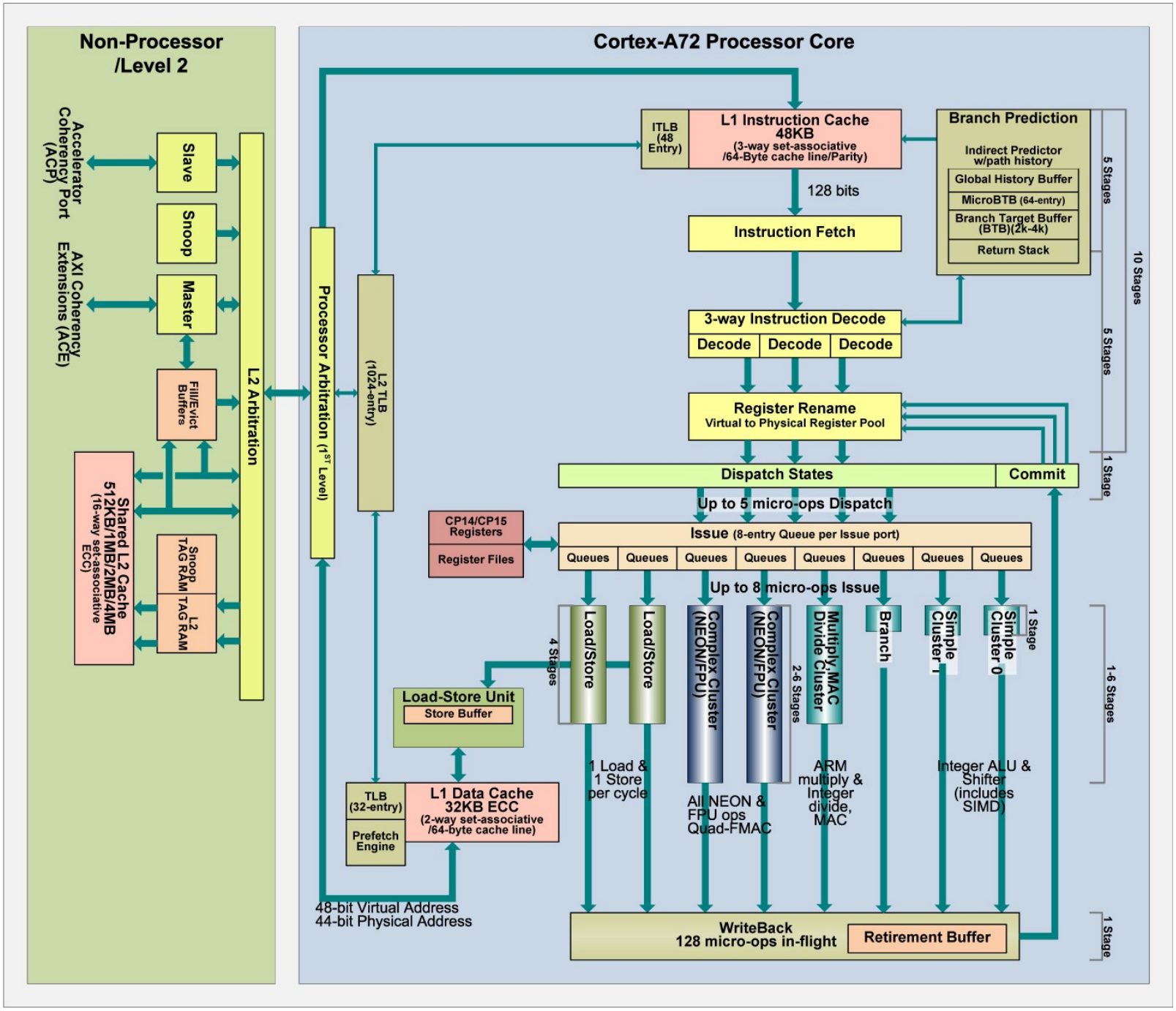
źródło: https://en.wikichip.org/wiki/intel/microarchitectures/sunny_cove

Zen 4 (2022)



źródło: https://chipsandcheese.com/2022/11/05/amds-zen-4-part-1-frontend-and-execution-engine/

ARM Cortex-A72 Block Diagram



źródło: https://pc.watch.impress.co.jp/docs/column/kaigai/699491.html

Jak używać

Program (komputerowy)/proces ?

Kod źródłowy

```
#include "ProgramParameters.hpp"
#include "Simulation.hpp"
#include "initialize.hpp"

#include <signal.h>


microflow::Simulation * simulation = NULL



void handleSigInt(int s)
{
  if (NULL == ::simulation)
  {
    exit(1) ;
  } else {
    ::simulation->stop() ;
  }
}



void registerSigIntHandler()
{
  struct sigaction sigIntHandler;

  sigIntHandler.sa_handler = handleSigIn
  sigemptyset(&sigIntHandler.sa_mask) ;
  sigIntHandler.sa_flags = 0 ;

  sigaction(SIGINT, &sigIntHandler, NULL
} ;



int main(int argc, char ** argv)
{
  microflow::ProgramParameters programPa
  bool preprocessGeometryImageOnly = pro

  microflow::initialize( programParamete

  microflow::Simulation simulation( "./"
```

preprocesor +
kompilator +
asembler +
linker

- definicja języka (standard)
- ABI
- biblioteki
- inne elementy...

```
g++ -Wall -O3 -ggdb
-Wnon-virtual-dtor -std=gnu++0x
-Wno-deprecated-declarations -DNDEBUG
-march=native -fopenmp -pg -c -o ...
```

Plik wykonywalny

- kod maszynowy
- dane
- inne elementy...

```
$ readelf -a program
ELF Header:
  Magic:   7f 45 4c 46 02 01 01 03 00 00 0
  Class:                             ELF64
  Data:                              2's c
  Version:                           1 (cu
  OS/ABI:                            UNIX
  ABI Version:                       0
  Type:                              EXEC
  Machine:                           Advan
  Version:                           0x1
  Entry point address:               0x407
  Start of program headers:          64 (b
  Start of section headers:          13601
  Flags:                             0x0
  Size of this header:               64 (b
  Size of program headers:           56 (b
  Number of program headers:         9
  Size of section headers:           64 (b
  Number of section headers:         41
  Section header string table index: 38
...
```

Program – kod binarny – proces

Program – *opis danych i działań*

- **struktura danych** – definicja danych i opis ich powiązań w **logicznej przestrzeni adresowej** (dostępnej w opisie algorytmu)
- **algorytm** – opis wykonania funkcji za pomocą **działań elementarnych** **(instrukcji/rozkazów)** **zdefiniowanych** ***w architekturze komputera*** **(lista rozkazów, ISA)**

Kod pośredni programu – ***obraz programu w logicznej przestrzeni*** adresowej

- kody operacyjne działań
- kody danych i obszary robocze
- adresy danych w logicznej przestrzeni adresowej
- powiązanie z funkcjami środowiska

Proces – program w czasie wykonania (ang. *run-time*)

- przydział (alokacja) pamięci operacyjnej
- **odwzorowanie kodu** w przydzielonej pamięci operacyjnej (fizycznej)
- powiązanie z innymi procesami
- kontrola stanu i reakcja na błędy (wyjątki)

Odwzorowanie programu w przestrzeni logicznej

Edycja – plik źródłowy (ang. *source*) – tekstowy (*.asm, *.txt, *.c, itp.)

- opis danych (zmienne) i ich struktury
- zapis algorytmu w języku programowania
- bezpieczne zakończenie programu – zwrot sterowania (gorący restart)

Kompilacja – plik wynikowy (ang. *object*) (*.obj) – częściowo binarny

- przekodowanie opisu na postać półskompilowaną (kod + powiązania)
- specyfikacja powiązań statycznych (ang. *early binding*)

Konsolidacja – tworzenie pliku wykonalnego (ang. *executable*) (*.exe, *.dll):

- łączenie modułów (ang. *linking*)
- realizacja powiązań statycznych (ang. *early binding*) – zmienne deklarowane
- tworzenie nagłówka pliku wykonalnego (ang. *header*)
  - nazwa, struktura i atrybuty pliku
  - zapotrzebowanie na pamięć operacyjną

Odwzorowanie programu w przestrzeni fizycznej

Przydział pamięci operacyjnej (RAM) (ang. *memory allocation*)

- dane wejściowe i zmienne robocze programu
- kod programu
- stos obliczeniowy
- bufor dynamiczny

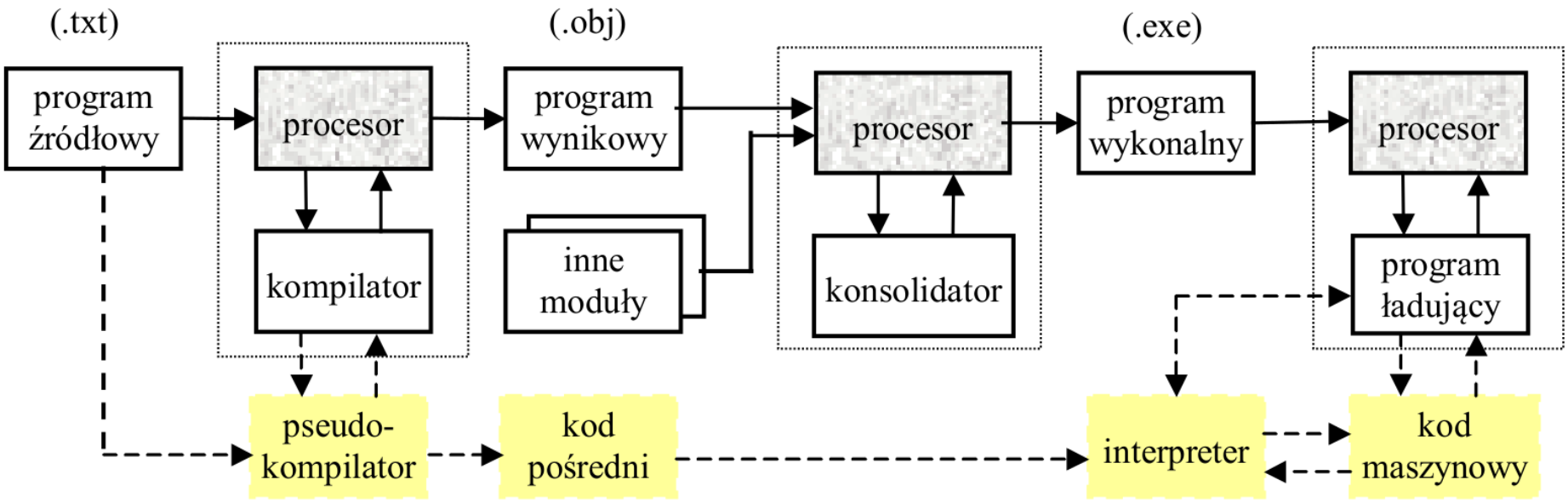
Załadowanie kodu do pamięci alokowanej (ang. *loading*)

- przydział bloku (partycji)
- odwzorowanie logicznej/wirtualnej przestrzeni programu w bloku
- kopiowanie kodu

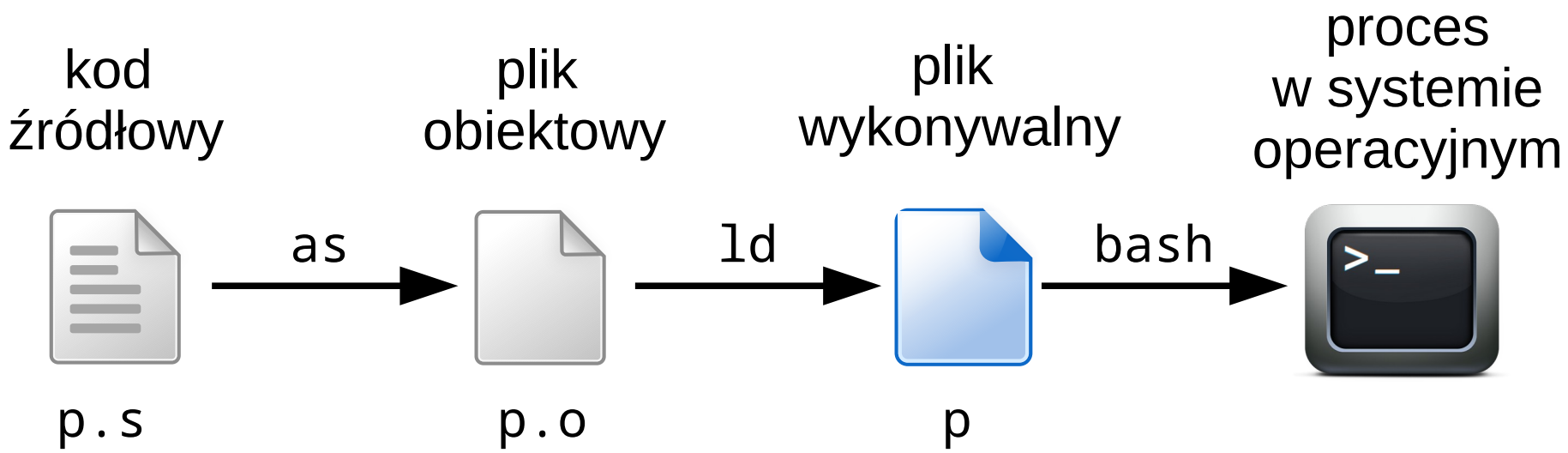
Wykonanie programu

- przekazanie sterowania do pierwszej instrukcji programu
- realizacja powiązań dynamicznych (ang. *late binding*) – (stos)
- zwrot sterowania do systemu operacyjnego
- zwolnienie alokowanego bloku (partycji) albo
  - pozostawienie kodu w pamięci (ang. *Terminate and Stay Resident*)

Konwersja międzypoziomowa - kompilacja i interpretacja



Laboratorium przykład




```
$ vim p.s
$ as p.s -o p.o && ld p.o -o p && ./p > wy && hexdump -C wy
```

Więcej przykładów w pliku sesja.txt i w materiałach do laboratorium

```
$ as -o rw.o rw.s
$ ls -a
.  ..  rw.o  rw.s
$ file rw.o
rw.o: ELF 64-bit LSB relocatable, x86-64, version 1 (SYSV), not
stripped
$ ld -o rw rw.o
$ ls -a
.  ..  rw  rw.o  rw.s
$ file rw
rw: ELF 64-bit LSB executable, x86-64, version 1 (SYSV), statically
linked, not stripped
$ ./rw
Hello, world!
$ objdump -f rw
rw:     file format elf64-x86-64
architecture: i386:x86-64, flags 0x00000112:
EXEC_P, HAS_SYMS, D_PAGED
start address 0x0000000000400086
```

```
$ as -alh rw.s
GAS LISTING rw.s  page 1
   1              # Numbers of kernel functions.
   2              EXIT_NR  = 1
   3              READ_NR  = 3
   4              WRITE_NR = 4
   5
   6              STDOUT = 1
   7              EXIT_CODE_SUCCESS = 0
   8
   9
  10              .text
  11 0000 48656C6C  msg: .ascii "Hello, world!\n"
  11      6F2C2077
  11      6F726C64
  11      210A
  12              msgLen = . - msg
  13
  14
  15              .global _start
  16
  17              _start:
  18
  19 000e B8040000  mov $WRITE_NR, %eax
  19      00
  20 0013 BB010000  mov $STDOUT  , %ebx
  20      00
  21 0018 B9000000  mov $msg     , %ecx
  21      00
  22 001d BA0E0000  mov $msgLen  , %edx
  22      00
  23 0022 CD80      int $0x80
  24
  25
  26 0024 B8010000  mov $EXIT_NR          , %eax
  26      00
  27 0029 BB000000  mov $EXIT_CODE_SUCCESS, %ebx
  27      00
  28 002e CD80      int $0x80
```

Asembler

- **asembler** – program tłumaczący kod w języku asemblera na kod maszynowy (przykłady dla architektury x86: GNU Assembler *gas*, The Netwide Assembler *nasm*, Turbo Assembler *tasm*, Microsoft Macro Assembler *masm)*
- **język asemblera** – potocznie *asembler*, w dużym uproszczeniu mnemoniczny zapis kodu maszynowego

Język asemblera

- mnemoniki instrukcji + argumenty
- dyrektywy
- symbole, etykiety
- wyrażenia
- komentarze

```
        .file "program.c"
        .section    .rodata
.LC1:
        .string     "Size of mystruct is %d\n"
        .align 4
        .text
        .globl      main
        .type main, @function
main:
        .cfi_startproc
        pushl %ebp
        jmp   .L2
.L3:
        movl  %ecx, (%eax)
        addl  $1, 20(%esp)
.L2:
        cmpl  $9, 20(%esp)
        jbe   .L3
        movl  20(%esp), %edx
        movl  %edx, %eax
        movl  (%eax), %eax
        movl  %eax, 4(%esp)
        movl  $.LC0, (%esp)
        call  printf
        leave
        ret
        .cfi_endproc
.LFE0:
        .size main, .-main
        .ident      "GCC: (Ubuntu/Linaro 4.7.3-1ubuntu1)
        .section    .note.GNU-stack,"",@progbits
```

dyrektywa

etykieta

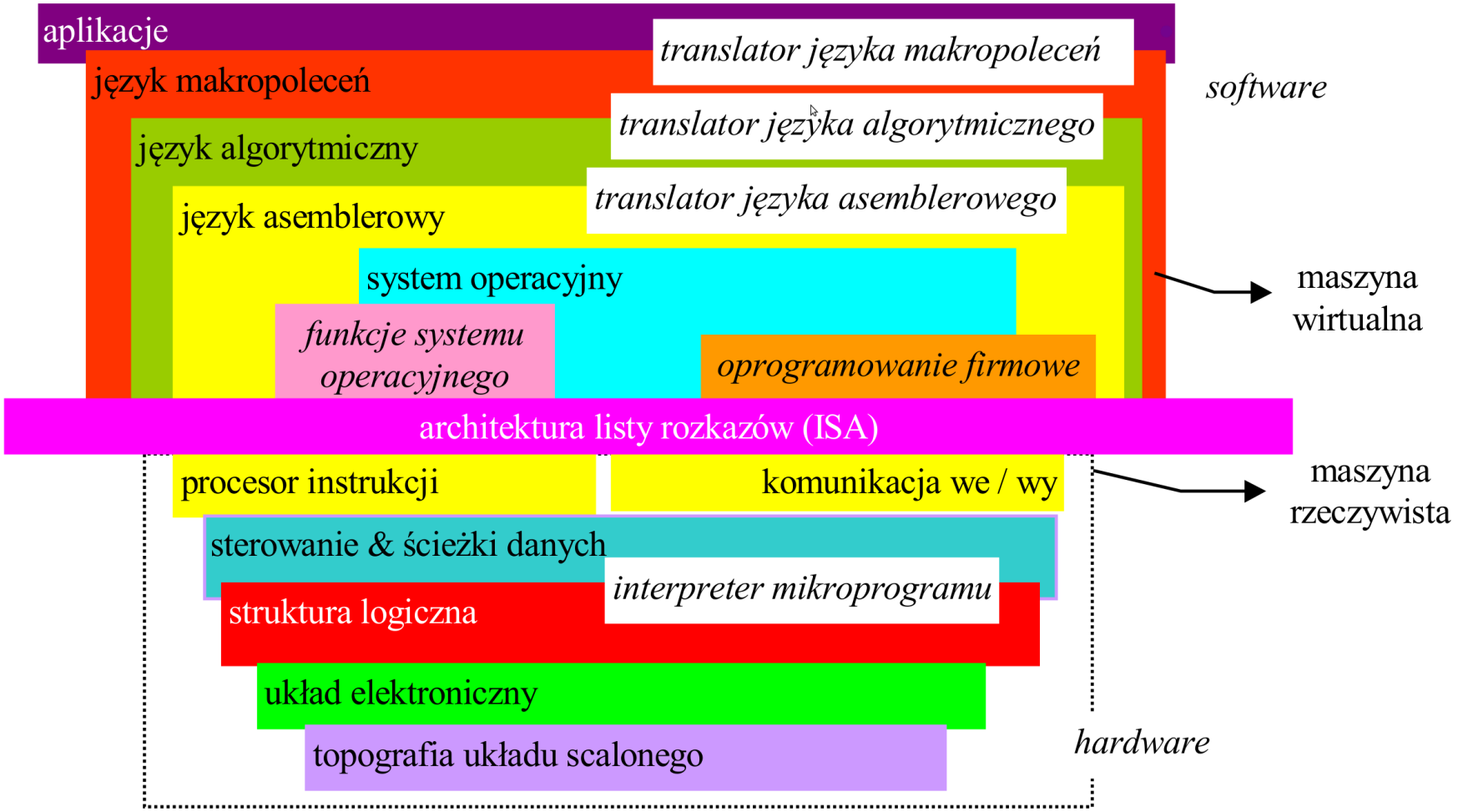
mnemonik

argumenty

symbol

wyrażenie

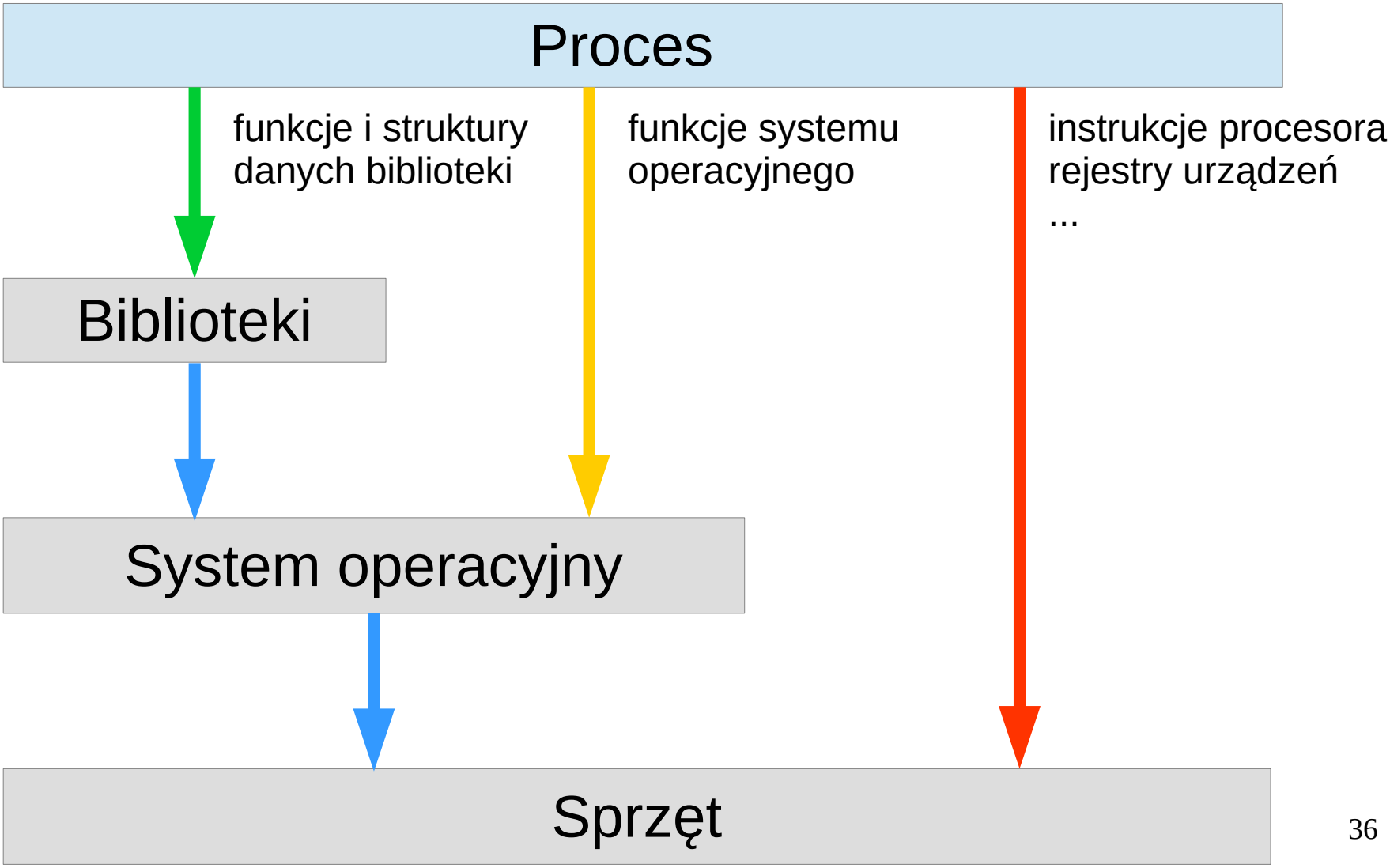
Poziomy maszynowe*)



Poziomy maszynowe i sprzężenia międzypoziomowe

[sprzężenia – realizacja powiązań (przypisania) nazw poziomu wyższego z identyfikatorami (nazwy, adresy, lokacje) na poziomie niższym]

Interfejsy wysokopoziomowe



Interfejsy wysokopoziomowe

```
$ man 2 read
```

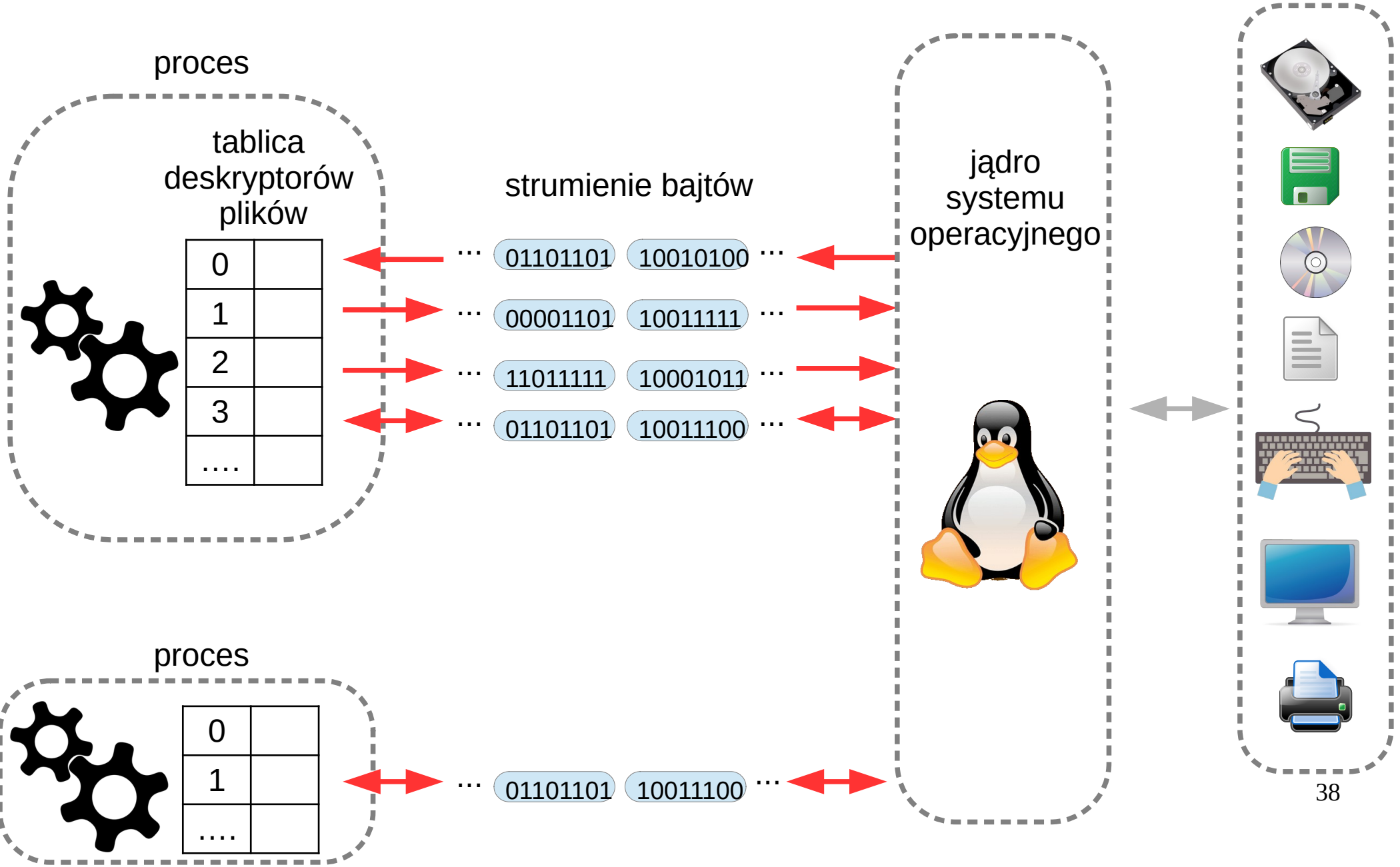
```
WRITE(2)                          Linux Programmer's Manual                          WRITE(2)

NAME
       write - write to a file descriptor

SYNOPSIS
       #include <unistd.h>

       ssize_t write(int fd, const void *buf, size_t count);

DESCRIPTION
       write()  writes  up  to  count bytes from the buffer pointed buf to the file referred to by the file
       descriptor fd.

       The number of bytes written may be less than count if, for example, there is insufficient  space  on
       the  underlying  physical  medium,  or  the  RLIMIT_FSIZE  resource  limit is encountered (see setr-
```

/usr/include/asm/unistd_32.h

```
 1 #ifndef _ASM_X86_UNISTD_32_H
 2 #define _ASM_X86_UNISTD_32_H 1
 3
 4 #define __NR_restart_syscall 0
 5 #define __NR_exit 1
 6 #define __NR_fork 2
 7 #define __NR_read 3
 8 #define __NR_write 4
 9 #define __NR_open 5
10 #define __NR_close 6
11 #define __NR_waitpid 7
12 #define __NR_creat 8
13 #define __NR_link 9
14 #define __NR_unlink 10
15 #define __NR_execve 11
```

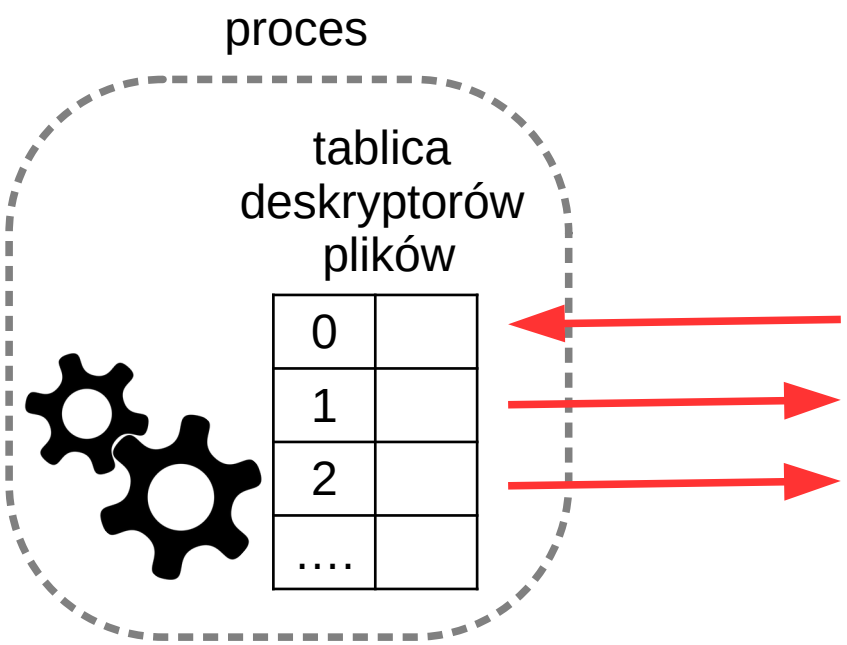
```
24 mov $WRITE_NR, %eax
25 mov $STDOUT  , %ebx
26 mov $msg     , %ecx
27 mov $msgLen  , %edx
28 int $0x80
```

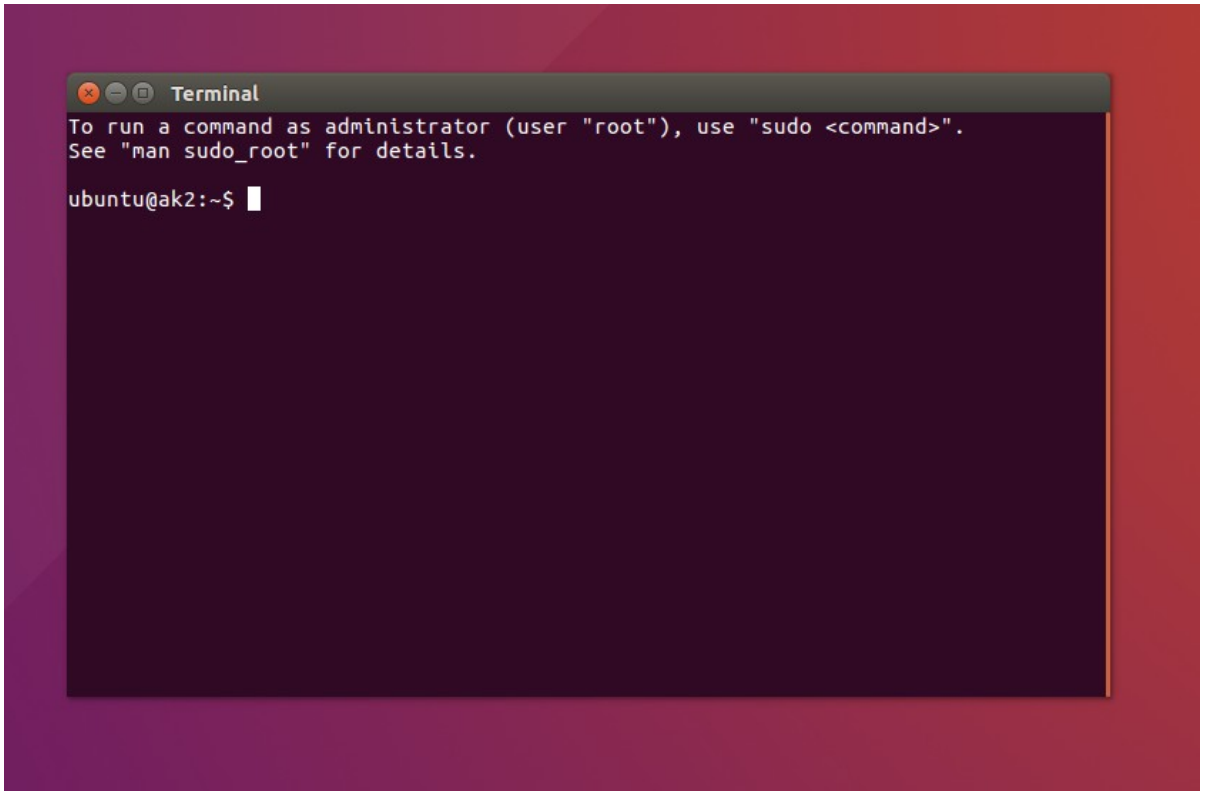
Interfejsy wysokopoziomowe



Interfejsy wysokopoziomowe

Interfejsy wysokopoziomowe





Podczas tworzenia procesu tworzone są trzy strumienie:
0 – standardowy strumień wejściowy
1 – standardowy strumień wyjściowy
2 – standardowy strumień błędów

Proces emulujący terminal.
Wyświetla glify (potocznie obrazki czcionek) na podstawie odbieranych bajtów traktowanych jak kody ASCII lub kody sterujące. Pozwala na edycję wprowadzanych danych dzięki ich buforowaniu.

Uwaga – glify nie są wyświetlane dla części odbieranych bajtów!

Interfejsy wysokopoziomowe

Powłoka systemowa (*ang. shell*) pozwala na przekierowanie strumieni:

https://www.gnu.org/software/bash/manual/html_node/Redirections.html

```
$ ./p < input_data > output_data
```

```
$ ./p < input_data 2>&1 | hexdump -C
```

Kod ASCII (American Standard Code for Information Interchange)

Kod ASCII (część międzynarodowa) = 0 || ISO-7 (CCITT No 5)

H L	0000	0001	0010	0011	0100	0101	0110	0111	1000	1001	1010	1011	1100	1101	1110	1111
0000	NUL	SOH	STX	ETX	EOT	ENQ	ACK	BEL	BS	HT	LF	VT	FF	CR	SO	SI
0001	DLE	DC1	DC2	DC3	DC4	NAK	SYN	ETB	CAN	EM	SUB	ESC	FS	GS	RS	US
0010	SP	!	”	#	$	%	&	‘	(	)	*	+	,	–	.	/
0011	**0**	1	2	3	4	5	6	7	8	9	:	;	<	=	>	?
0100	@	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O
0101	P	Q	R	S	T	U	V	W	X	Y	Z	[	\	]	^	—
0110	`	a	b	c	d	e	f	g	h	i	j	k	l	m	n	o
0111	p	q	r	s	t	u	v	w	x	y	z	{	|	}	~	DEL

NUL *nullify*	SOH – *start of header*	STX – *start of text*	ETX – *end of text*
EOT – *end of transfer*	ENQ – *enquire*	ACK – *acknowledge*	BEL – *bell*
BS – *backspace*	HT – *horizontal tab*	LF – *line feed*	VT – *vertical tab*
FF – *form feed*	CR – *carriage return*	SO / SI – *shift out i in*	DLE – *data link ESC*
DC1,...4 – *data control*	NAK – *negative ACK*	SYN – *synchronize*	ETB – *end of text block*
CAN – *cancel*	EM – *end of medium*	SUB – *substitute*	ESC – *escape*
FS – *file separator*	GS – *group separator*	RS – *record separator*	US – *unit separator*

UNICODE – kod 16/32-bitowy, obejmujący znaki diaktryczne większości języków

Kod ASCII – regularności

Konwencje asemblerowe:

'*znak*' – kod ASCII (1bajt) znaku alfanumerycznego (litery, cyfry, +, -, =,...)

"*tekst*" – ciąg kodów ASCII kolejnych znaków tekstu (konwencja BE)

```
Zapis znaków specjalnych w tekście ciągłym – \♠   # znak specjalny ♠ (np. \\, \", \?), w szczególności
\ddd  # kod ósemkowy ddd              \xDD # kod szesnastkowy DD
\n    # (\x0A=\012) LF,NL, new line   \0   # (\x00=\000) NUL, koniec rekordu
\b    # (\x08=\010) BS, backspace     \t   # (\x09=\011) HT,TAB, tabulation
\f    # (\x0C=\014) FF, form feed     \r   # (\x0D=\015) CR, carriage return
```

kody cyfr dziesiętnych ($Z_{16} = bbbb_2$ – wartość cyfry):

$3\#_{16}$ – w notacji szesnastkowej ('7' = 0x37)

0011 bbbb – w notacji dwójkowej ('3' = 0b0011 0011)

wartość cyfry X: 'X'–'0' lub 'X'– $0x30_{16}$ lub 'X' AND $0x0F_{16}$

kody liter (bbbbb – 5-bitowy nr litery w porządku alfabetu łacińskiego):

dużych: 010 bbbbb ('A' = 0b010 00001 = 0x41= 41_{16}),

małych: 011 bbbbb ('z' = 0b011 11010 = 0x7A= $7A_{16}$)

nr litery w porządku alfabetu: – 'X' AND $0x1F_{16}$

zamiana „duża" ↔ „mała": – 'X' XOR $0x20_{16}$

3.6.1.2 Characters

A single character may be written as a single quote immediately followed by that character.

Some backslash escapes apply to characters, \b, \f, \n, \r, \t, and \" with the same meaning as for strings, plus \' for a single quote. So if you want to write the character backslash, you must write '\\ where the first \ escapes the second \. As you can see, the quote is an acute accent, not a grave accent. A newline immediately following an acute accent is taken as a literal character and does not count as the end of a statement.

The value of a character constant in a numeric expression is the machine's byte-wide code for that character. as assumes your character code is ASCII: 'A means 65, 'B means 66, and so on.

Architektura komputera

Architektura komputera

Architektura komputera

(*ang. instruction set architecture ISA*):

specyfikacja cech funkcjonalnych

(typy danych, lista rozkazów)

czyli potocznie komputer z punktu widzenia programisty

Model programowy procesora

ang. Instruction Set Architecture, **ISA**

- lista rozkazów (instrukcje procesora)
- **tryby adresowania** (określenie operandów*)
- rejestry (liczba, rozmiary, przeznaczenie)
- typy danych
- obsługa wyjątków i przerwań

*Niejednoznaczność - o trybach adresowania można też mówić wyłącznie w kontekście dostępu do pamięci

Instrukcje procesora

- w prostych procesorach interfejs pozwalający na „bezpośredni” dostęp do sprzętu
- w złożonych procesorach język pośredni tłumaczony w późniejszych etapach (dodatkowa kompilacja lub sprzętowo - mikrokod)
  - niezależność języka od implementacji
  - łatwiejsze projektowanie sprzętu
  - bardziej złożone operacje – mniejsze instrukcje
  - ukrycie szczegółów implementacji
  - niższa wydajność – rozwiązania hybrydowe

Klasyfikacja instrukcji

- przesłania danych
- operacje logiczne
- zmiany formatu (np. rozszerzanie)
- konwersje (np. stałoprzecinkowe ↔ zmiennoprzecinkowe)
- arytmetyka (binarna, dziesiętna, …)
- rozgałęzienia (zmiana przepływu sterowania)
- specjalne (NOP, LEA, CPUID, ...)
- instrukcje koprocesorów (zmiennoprzecinkowego, wektorowych)
- systemowe (niedostępne w programach użytkownika)

Tryby adresowania

Adres – informacja, **gdzie** znajduje się **wartość** argumentu

gdzie: w kodzie rozkazu, w rejestrze, w komórce(kach) pamięci, w urządzeniu we/wy, na dysku, w internecie, w książce na stronie…

Rejestr (komórka pamięci itd.) **NIE** jest argumentem.

Tryby adresowania – ogólny podział

- **Bezpośrednie** – wartość lub adres (numer rejestru lub komórki pamięci) argumentu zawarty jest w kodzie rozkazu:
  - **zwarte** (argument domniemany)
    `cpuid`
  - **natychmiastowe**: wartość operandu zawarta w kodzie rozkazu
    `push $100`
  - **bezwzględne**: adres komórki pamięci w kodzie rozkazu
    `inc 100`
  - **bezpośrednie rejestrowe**: adres rejestru w kodzie rozkazu
    `inc %eax`
- **Pośrednie -** w kodzie rozkazu zawarta jest informacja pozwalająca na obliczenie adresu operandu (lokacje składowych i metoda obliczania)
  `inc (%eax)`

Model pamięci – architektura IA-32, IA-32e

model pamięci

sposób tworzenia adresu w logicznej przestrzeni adresowej;
odwzorowanie *logicznej przestrzeni adresowej* ***w pamięci operacyjnej***

adresowanie pamięci

tryb adresowania –sposób obliczenia adresu w logicznej przestrzeni adresowej;
adres fizyczny – wynik przekształcenia adresu logicznego,
zgodnie z systemowym mechanizmem odwzorowania

modele pamięci 80x86/IA-32

pamięć segmentowana

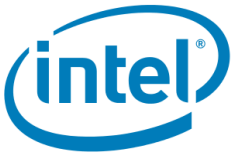
adres: – segment:[*tryb adr.*] = seg:[baza+indeks*skala+przemieszczenie]
seg = cs, ds, es, fs, gs, ss
baza|indeks = eax, ebx, ecx, edx, esi, edi, ebp, esp, (baza ≠ esp)

pamięć liniowa (seg = 00)

adres: – [baza+indeks*skala+przemieszczenie]
baza|indeks = dowolny rejestr 32- lub 64-bitowy (baza ≠ esp/rsp)

Model programowy x86-64

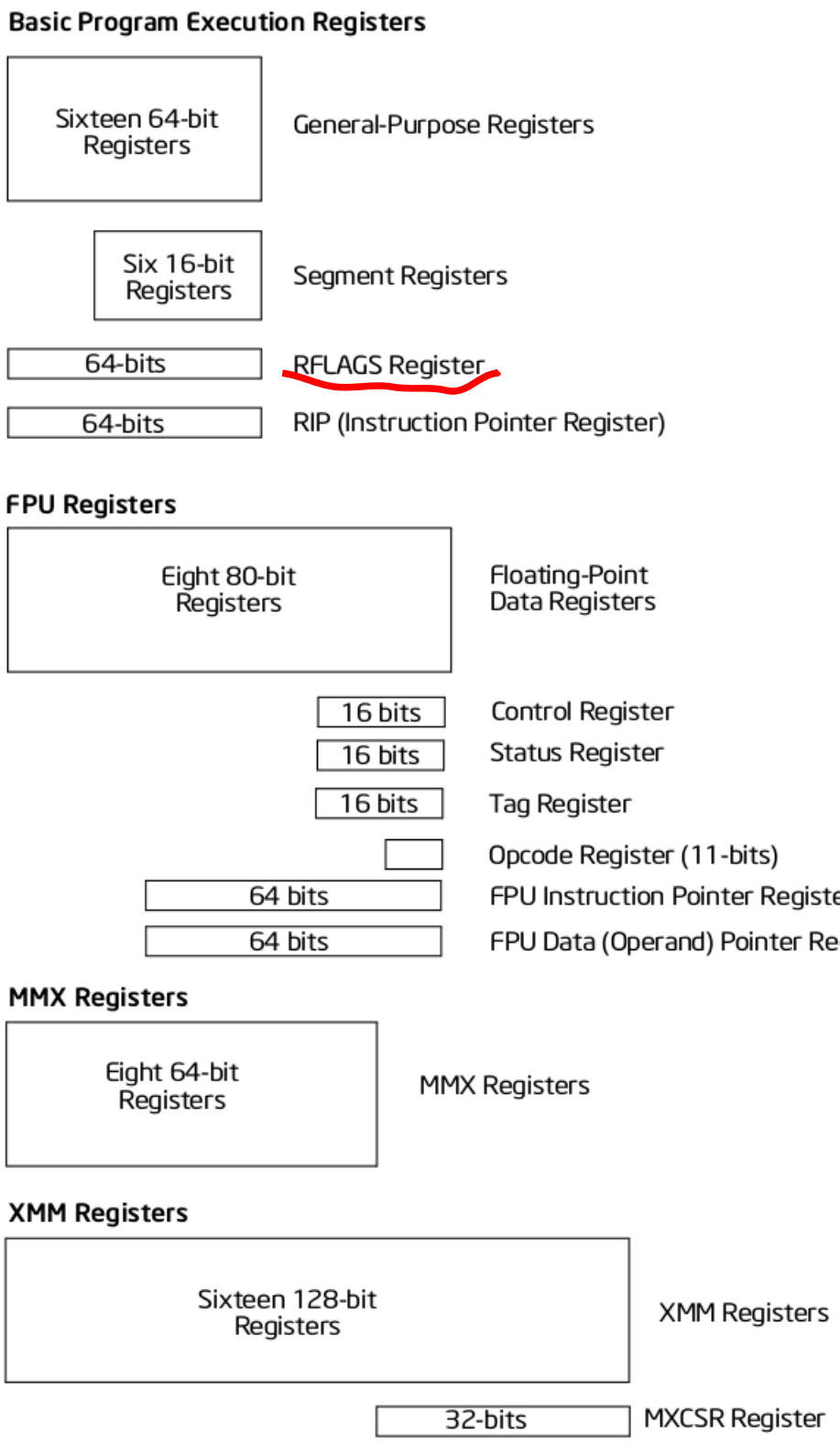
- http://www.intel.com/content/www/us/en/processors/architectures-software-developer-manuals.html

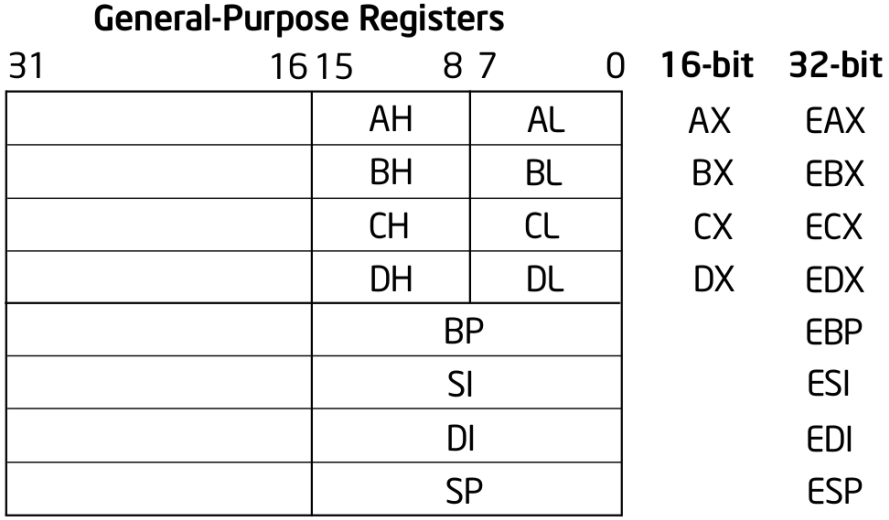


Intel® 64 and IA-32 Architectures Software Developer's Manual

Combined Volumes:
1, 2A, 2B, 2C, 3A, 3B and 3C

NOTE: This document contains all seven volumes of the Intel 64 and IA-32 Architectures Software Developer's Manual: *Basic Architecture*, *Instruction Set Reference A-M*, *Instruction Set Reference N-Z*, I*nstruction Set Reference, and the System Programming Guide, Parts 1*, 2 and 3. Refer to all seven volumes when evaluating your design needs.



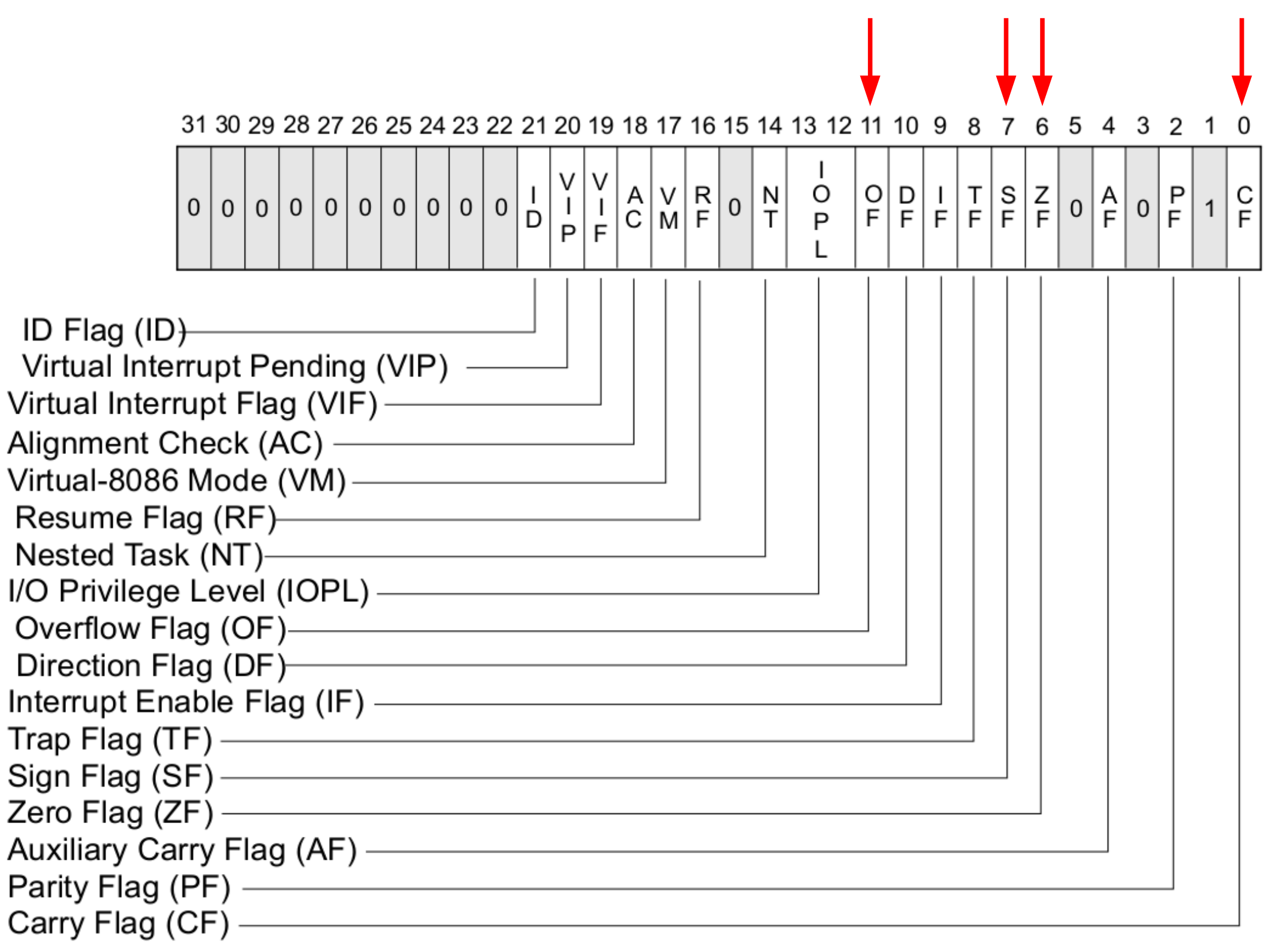


x86-64 (1999)

- rax, rbx, ...
- dodatkowe r8-r15
- sil, dil, bpl, spl
- r8b, r8w, r8d, …
- ograniczenia dla AH, BH, …

APX (2023)

- r16-r31
- ...


31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
0 0 0 0 0 0 0 0 0 0 ID VIP VIF AC VM RF 0 NT IOPL OF DF IF TF SF ZF 0 AF 0 PF 1 CF
ID Flag (ID)
Virtual Interrupt Pending (VIP)
Virtual Interrupt Flag (VIF)
Alignment Check (AC)
Virtual-8086 Mode (VM)
Resume Flag (RF)
Nested Task (NT)
I/O Privilege Level (IOPL)
Overflow Flag (OF)
Direction Flag (DF)
Interrupt Enable Flag (IF)
Trap Flag (TF)
Sign Flag (SF)
Zero Flag (ZF)
Auxiliary Carry Flag (AF)
Parity Flag (PF)
Carry Flag (CF)

Argumenty instrukcji – typy danych

- Podstawowe:
  - bajt, słowo, podwójne/poczwórne słowo (*ang. doubleword/quadword*)
- Liczbowe:
  - stałoprzecinkowe (*ang. integer*)
    - ze znakiem (*ang. signed*)
    - bez znaku (*ang. unsigned*)
    - BCD i spakowane BCD (*ang. binary-coded decimal*)
  - zmiennoprzecinkowe IEEE 754 (*ang. floating-point)*
  - wektorowe stało- i zmiennoprzecinkowe *(ang. packed)*
- Wskaźnikowe (*ang. pointer*)
- Pola bitowe (*ang. bit field*)
- Ciągi (*ang. string*)

Argumenty instrukcji – typy danych

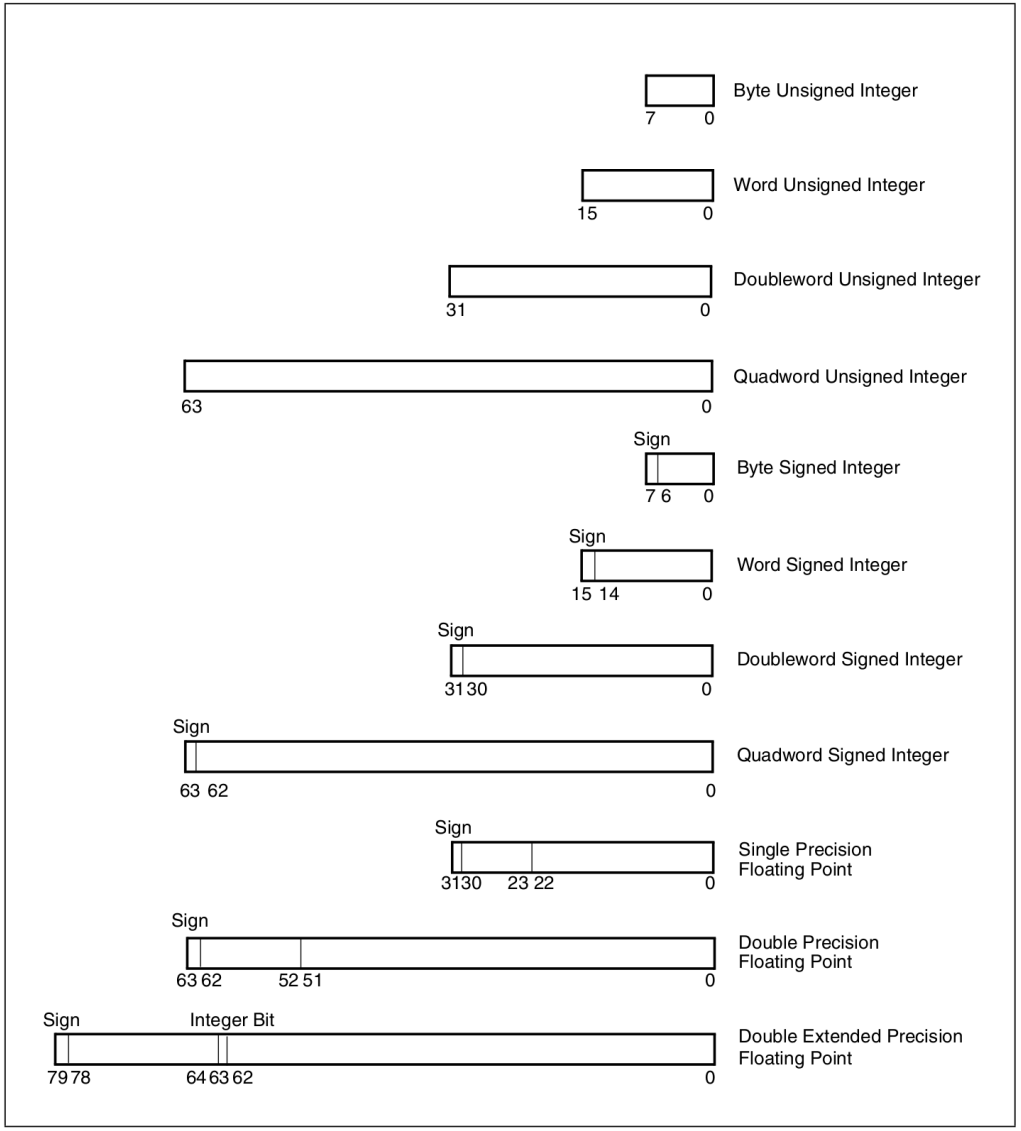


Figure 4-3. Numeric Data Types

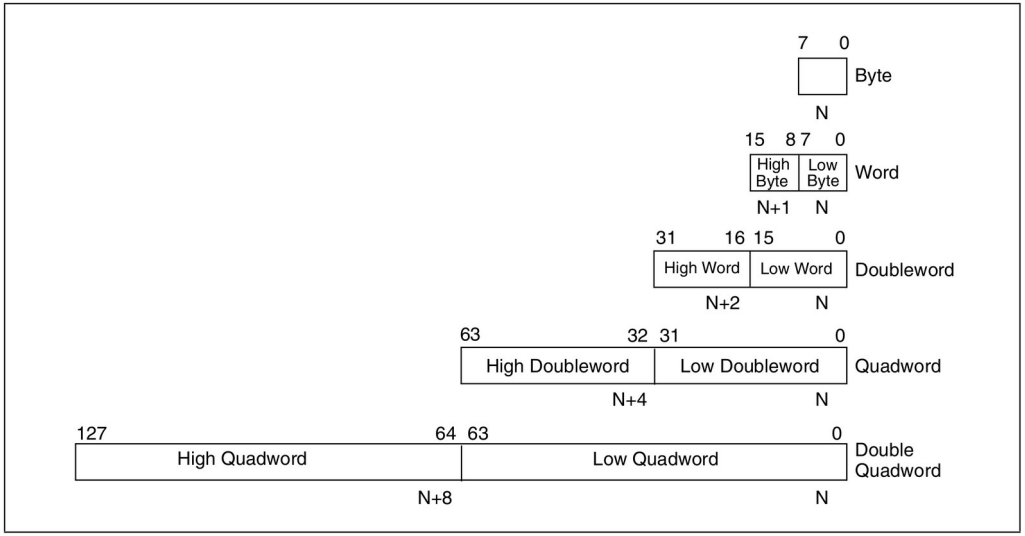


Figure 4-1. Fundamental Data Types

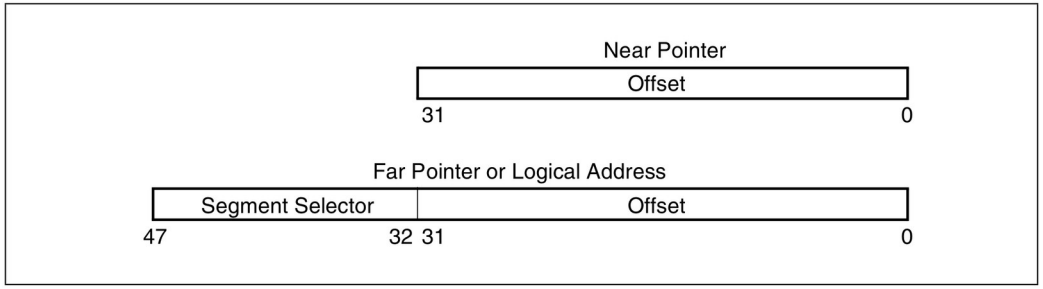
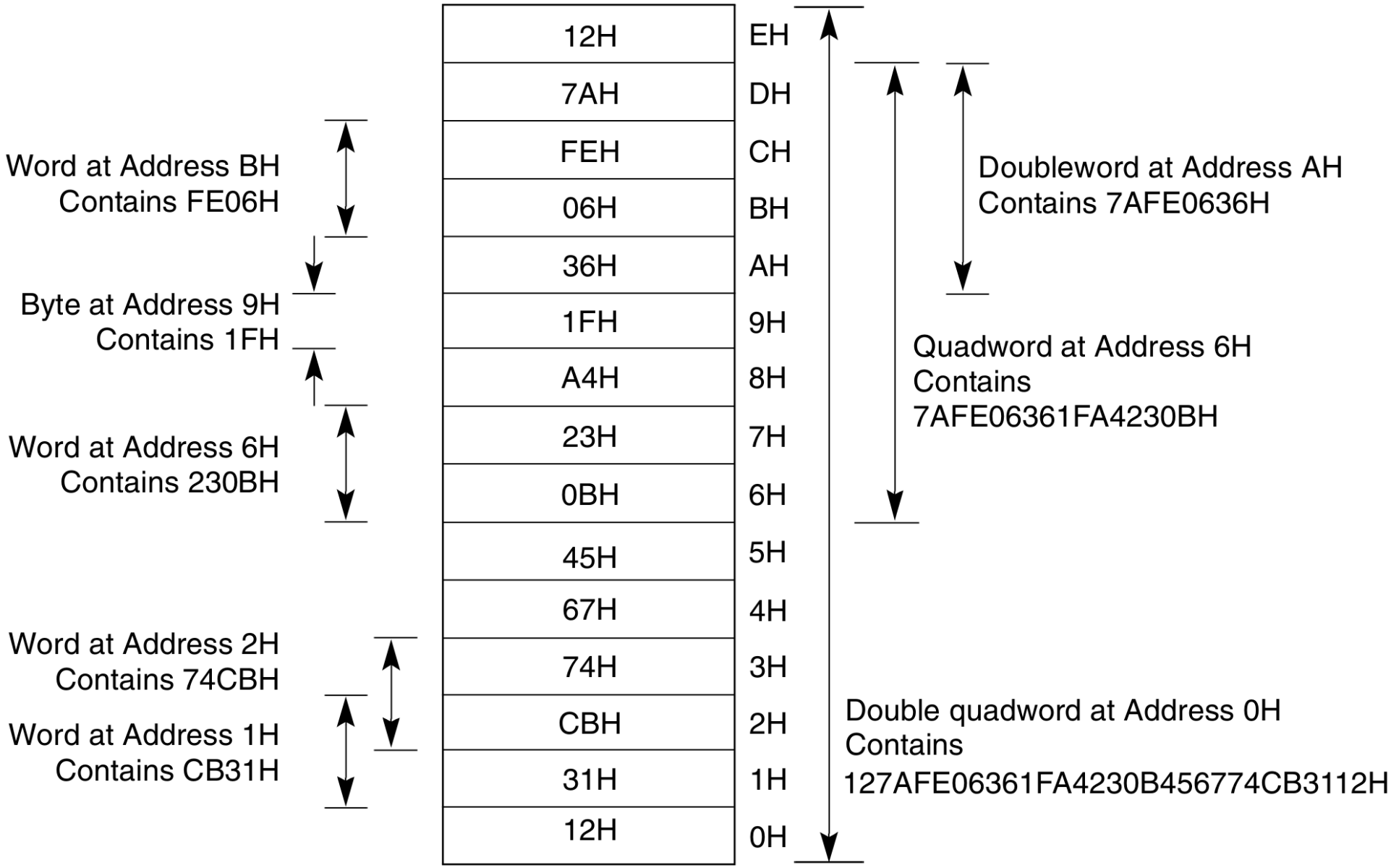


Figure 4-4. Pointer Data Types

Kolejność bajtów (*ang. endianness*)



Argumenty instrukcji – tryby adresowania

- natychmiastowe (*ang. immediate operands*)
- rejestry (*ang. register operands*)
- porty we/wy (*ang. IO ports*)
- lokacje w pamięci (*ang. memory operands*)

Pamięć

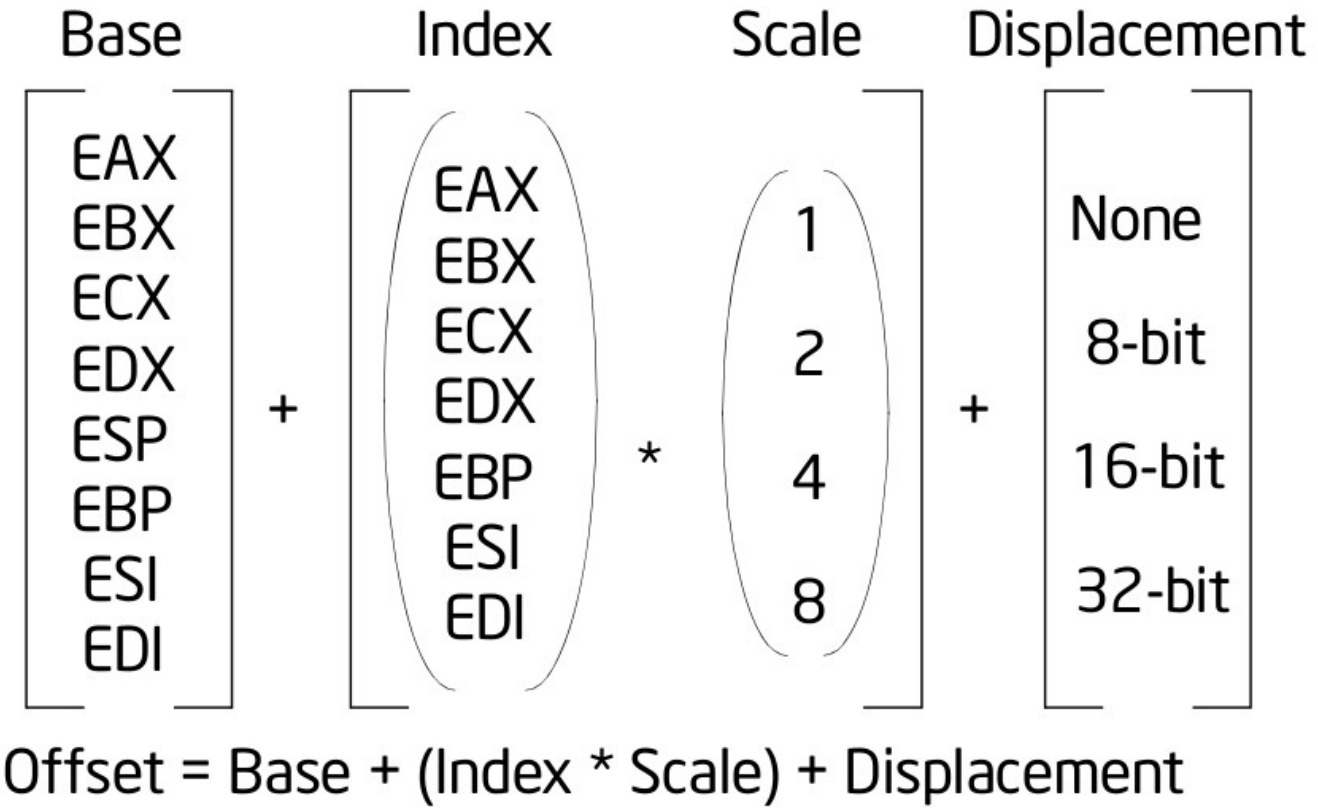
Tablica (wektor) ponumerowanych komórek

Ilu?

10^5 (mikrokontroler 8051), 10^9 do 10^{11} (desktop), 10^{13} do 10^{16} (serwer)

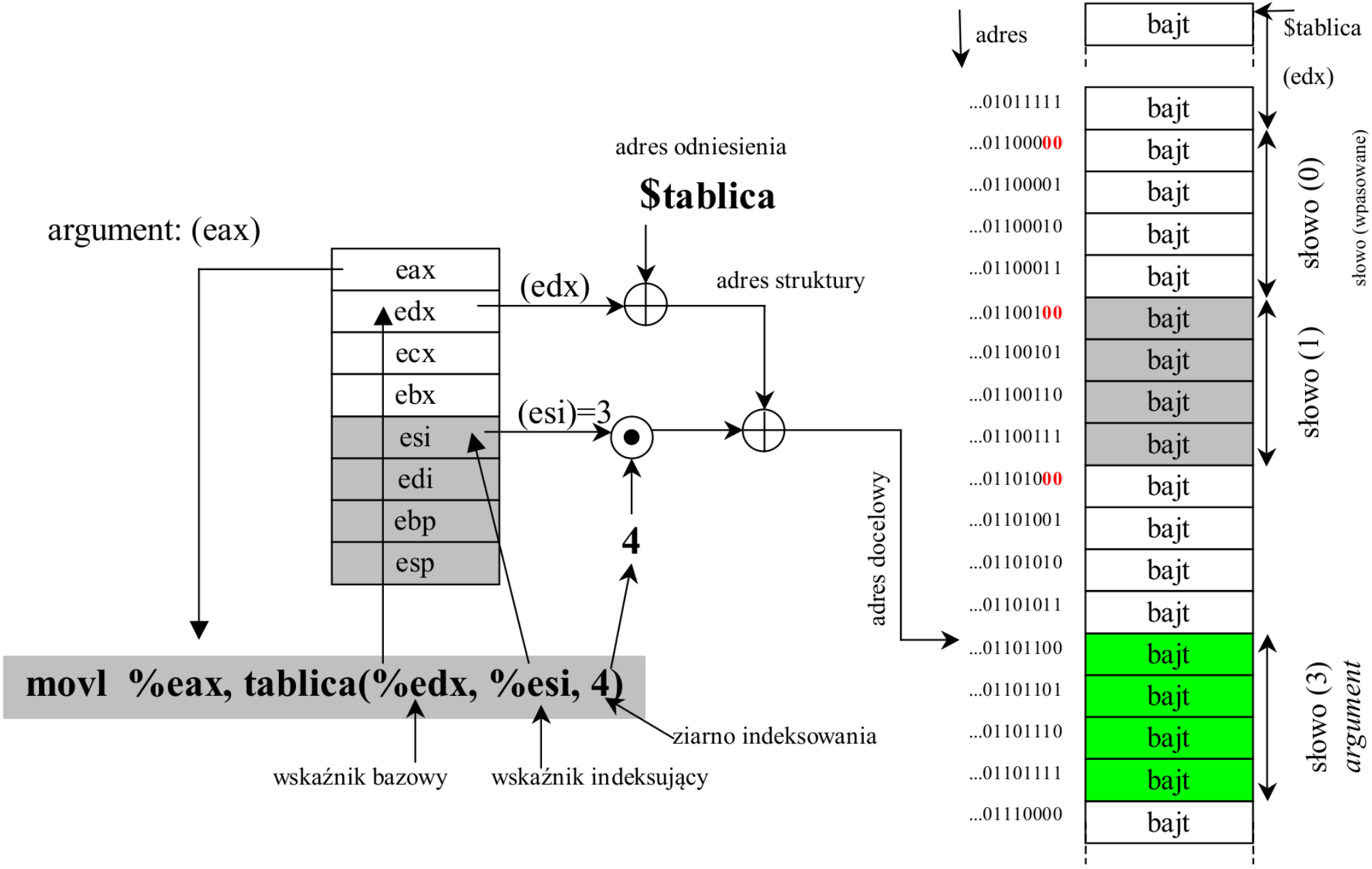
UWAGA:
to bardzo duże uproszczenie, szczegóły w dalszej części semestru !!!

Argumenty instrukcji – tryby adresowania



Offset = Base + (Index * Scale) + Displacement

Architektura IA-32/IA-32e – schemat adresowania



Argumenty instrukcji – tryby adresowania

- Symbole

  https://sourceware.org/binutils/docs/as/Symbol-Intro.html
  https://sourceware.org/binutils/docs/as/Symbols.html

```
WRITE_NR = 4
```

- Etykiety

```
_start:
     mov $WRITE_NR, %eax
```

- Symbol .

Symbole są zastępowane wartościami podczas asemblacji, linkowania, ładowania programu, a nawet jego wykonywania (*ang. lazy binding/linking*)

Argumenty instrukcji – tryby adresowania

```
EXIT_NR  = 1
READ_NR  = 3
WRITE_NR = 4
STDOUT = 1
EXIT_CODE_SUCCESS = 0

.text
msg: .ascii "Hello, world!\n"
msgLen = . - msg

.global _start
_start:

mov $WRITE_NR, %eax
mov $STDOUT  , %ebx
mov $msg     , %ecx
mov $msgLen  , %edx
int $0x80

mov $EXIT_NR          , %eax
mov $EXIT_CODE_SUCCESS, %ebx
int $0x80
```

```
$ nm p.o
0000000000000000 a EXIT_CODE_SUCCESS
0000000000000001 a EXIT_NR
0000000000000003 a READ_NR
0000000000000001 a STDOUT
0000000000000004 a WRITE_NR
000000000000000e T _start
0000000000000000 t msg
000000000000000e a msgLen
$ nm p
0000000000000000 a EXIT_CODE_SUCCESS
0000000000000001 a EXIT_NR
0000000000000003 a READ_NR
0000000000000001 a STDOUT
0000000000000004 a WRITE_NR
0000000000601000 A __bss_start
0000000000601000 A _edata
0000000000601000 A _end
0000000000400086 T _start
0000000000400078 t msg
000000000000000e a msgLen
```

Argumenty instrukcji – tryby adresowania

```
EXIT_NR  = 1
READ_NR  = 3
WRITE_NR = 4
STDOUT = 1
EXIT_CODE_SUCCESS = 0

.text
msg: .ascii "Hello, world!\n"
msgLen = . - msg

.global _start
_start:

mov $WRITE_NR, %eax
mov $STDOUT  , %ebx
mov $msg     , %ecx
mov $msgLen  , %edx
int $0x80

mov $EXIT_NR          , %eax
mov $EXIT_CODE_SUCCESS, %ebx
int $0x80
```

```
$ objdump -d rw

rw:     file format elf64-x86-64


Disassembly of section .text:

0000000000400078 <.text>:
  400078: 48                    rex.W
  400079: 65                    gs
  40007a: 6c                    insb   (%dx),%es:(%rdi)
  40007b: 6c                    insb   (%dx),%es:(%rdi)
  40007c: 6f                    outsl  %ds:(%rsi),(%dx)
  40007d: 2c 20                 sub    $0x20,%al
  40007f: 77 6f                 ja     0x4000f0
  400081: 72 6c                 jb     0x4000ef
  400083: 64 21 0a              and    %ecx,%fs:(%rdx)
  400086: b8 04 00 00 00        mov    $0x4,%eax
  40008b: bb 01 00 00 00        mov    $0x1,%ebx
  400090: b9 78 00 40 00        mov    $0x400078,%ecx
  400095: ba 0e 00 00 00        mov    $0xe,%edx
  40009a: cd 80                 int    $0x80
  40009c: b8 01 00 00 00        mov    $0x1,%eax
  4000a1: bb 00 00 00 00        mov    $0x0,%ebx
  4000a6: cd 80                 int    $0x80
```

9.16.3.1 AT&T Syntax versus Intel Syntax

Notable differences between the two syntaxes are:

- AT&T immediate operands are preceded by '$'; Intel immediate operands are undelimited (Intel 'push 4' is AT&T 'pushl $4').
  AT&T register operands are preceded by '%'; Intel register operands are undelimited.
- AT&T and Intel syntax use the opposite order for source and destination operands. Intel 'add eax, 4' is 'addl $4, %eax'. The 'source, dest' convention is maintained for compatibility with previous Unix assemblers. Note that 'bound', 'invlpga', and instructions with 2 immediate operands, such as the 'enter' instruction, do not have reversed order.
- In AT&T syntax the size of memory operands is determined from the last character of the instruction mnemonic. Mnemonic suffixes of 'b', 'w', 'l' and 'q' specify byte (8-bit), word (16-bit), long (32-bit) and quadruple word (64-bit) memory references. Mnemonic suffixes of 'x', 'y' and 'z' specify xmm (128-bit vector), ymm (256-bit vector) and zmm (512-bit vector) memory references, only when there's no other way to disambiguate an instruction. Intel syntax accomplishes this by prefixing memory operands (not the instruction mnemonics) with 'byte ptr', 'word ptr', 'dword ptr', 'qword ptr', 'xmmword ptr', 'ymmword ptr' and 'zmmword ptr'. Thus, Intel syntax 'mov al, byte ptr foo' is 'movb foo, %al' in AT&T syntax. In Intel syntax, 'fword ptr', 'tbyte ptr' and 'oword ptr' specify 48-bit, 80-bit and 128-bit memory references.

źródło: https://sourceware.org/binutils/docs/as/

9.16.7 Memory References

An Intel syntax indirect memory reference of the form

section:[base + index*scale + disp]

is translated into the AT&T syntax

`section:disp(base, index, scale)`

where base and index are the optional 32-bit base and index registers, disp is the optional displacement, and scale, taking the values 1, 2, 4, and 8, multiplies index to calculate the address of the operand. If no scale is specified, scale is taken to be 1. section specifies the optional section register for the memory operand (...)

Here are some examples of Intel and AT&T style memory references:

AT&T: '`-4(%ebp)`', Intel: '[ebp - 4]'

base is '`%ebp`'; disp is '`-4`'. section is missing, and the default section is used ('%ss' for addressing with '%ebp' as the base register). index, scale are both missing.

AT&T: '`foo(,%eax,4)`', Intel: '[foo + eax*4]'

index is '`%eax`' (scaled by a scale 4); disp is '`foo`'. All other fields are missing. The section register here defaults to '`%ds`'.

AT&T: '`foo(,1)`'; Intel '[foo]'

This uses the value pointed to by '`foo`' as a memory operand. Note that base and index are both missing, but there is only one ','. This is a syntactic exception.

Any instruction that has a memory operand, but no register operand, must specify its size (byte, word, long, or quadruple) with an instruction mnemonic suffix ('b', 'w', 'l' or 'q', respectively).

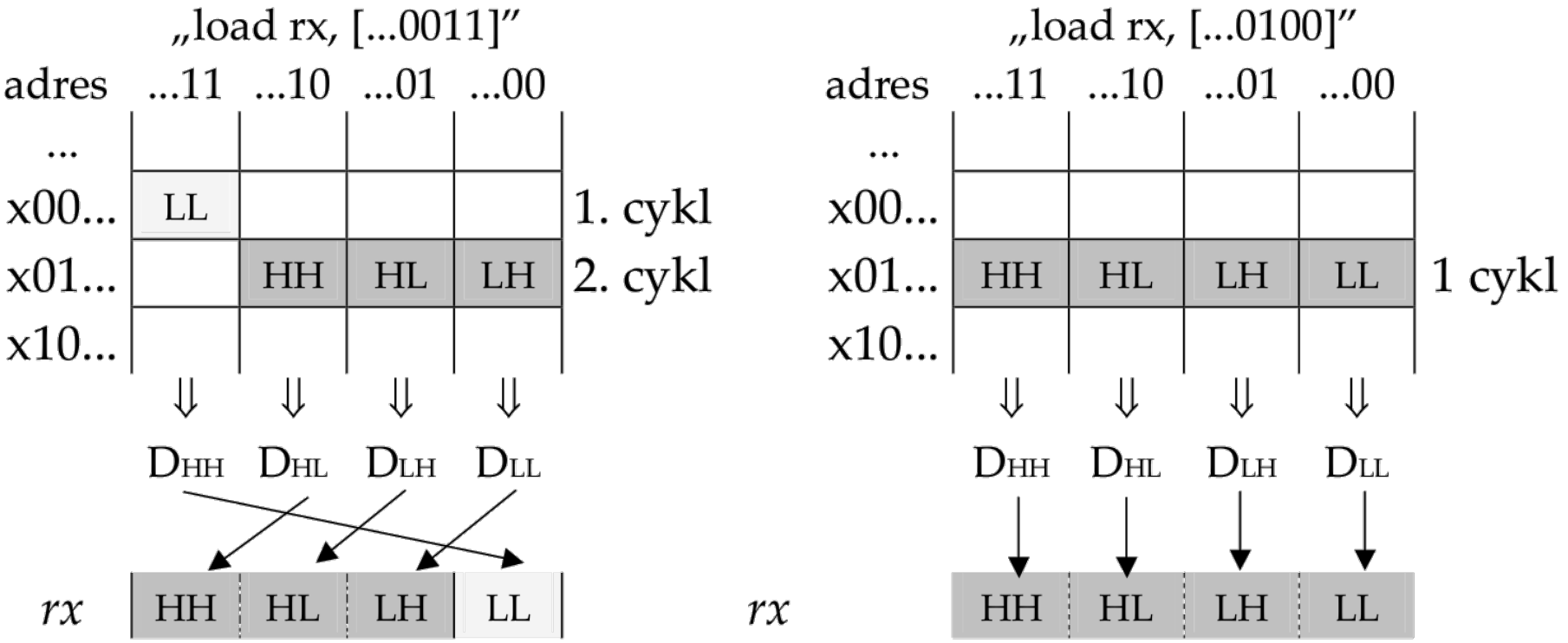
Czasowe aspekty adresowania – wyrównanie adresów

Rozdzielczość adresowania – fizycznie adresowalna jednostka informacji

- bit – szczególne przypadki (CISC – specjalne rozkazy dostępu)
- bajt – jednostka standardowa w pamięci fizycznej (*organizacja bajtowa*)
- słowo – jednostka logicznej organizacji programu (rozkazów i danych)

wyrównanie adresu(ang. *alignment*)

– umieszczenie obiektu począwszy od adresu podzielnego przez jego rozmiar



7.3 .align [abs-expr[, abs-expr[, abs-expr]]]

Pad the location counter (in the current subsection) to a particular storage boundary.

The first expression (which must be absolute) is the alignment required, as described below. If this expression is omitted then a default value of 0 is used, effectively disabling alignment requirements.

The second expression (also absolute) gives the fill value to be stored in the padding bytes. It (and the comma) may be omitted. If it is omitted, the padding bytes are normally zero. However, on most systems, if the section is marked as containing code and the fill value is omitted, the space is filled with no-op instructions.

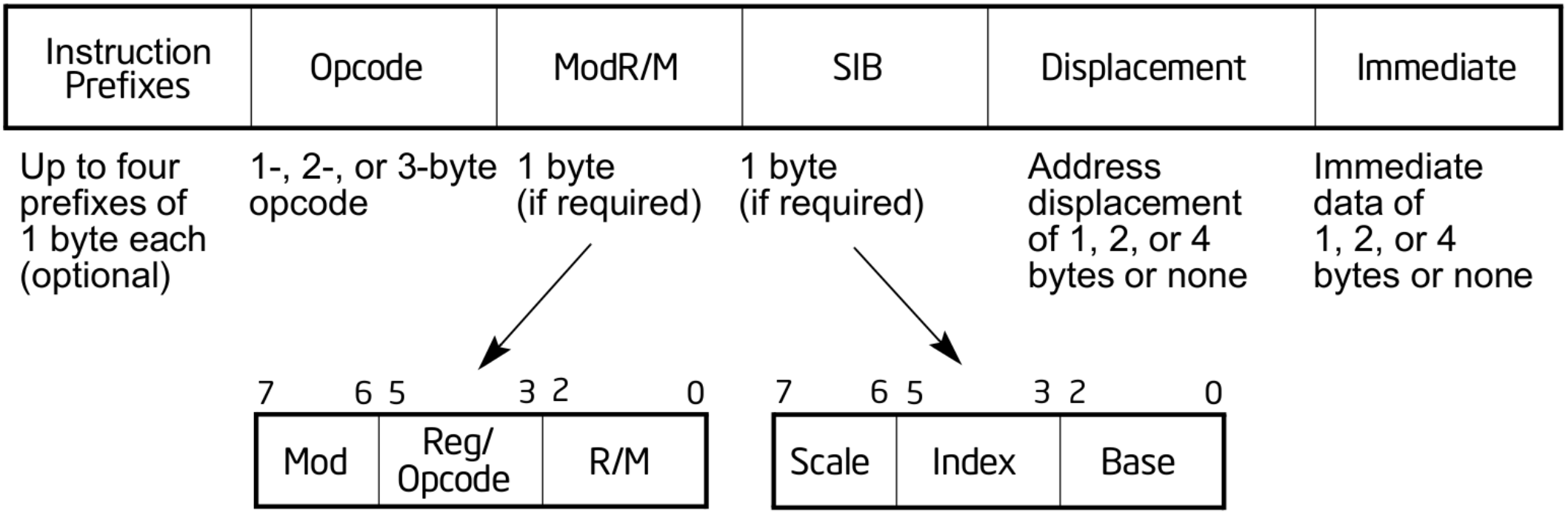
The third expression is also absolute, and is also optional. If it is present, it is the maximum number of bytes that should be skipped by this alignment directive. If doing the alignment would require skipping more bytes than the specified maximum, then the alignment is not done at all. You can omit the fill value (the second argument) entirely by simply using two commas after the required alignment; this can be useful if you want the alignment to be filled with no-op instructions when appropriate.

The way the required alignment is specified varies from system to system. For the arc, hppa, i386 using ELF, iq2000, m68k, or1k, s390, sparc, tic4x and xtensa, the first expression is the alignment request in bytes. For example ‘.align 8’ advances the location counter until it is a multiple of 8. If the location counter is already a multiple of 8, no change is needed. For the tic54x, the first expression is the alignment request in words.

For other systems, including ppc, i386 using a.out format, arm and strongarm, it is the number of low-order zero bits the location counter must have after advancement. For example ‘.align 3’ advances the location counter until it is a multiple of 8. If the location counter is already a multiple of 8, no change is needed.

This inconsistency is due to the different behaviors of the various native assemblers for these systems which GAS must emulate. GAS also provides .balign and .p2align directives, described later, which have a consistent behavior across all architectures (but are specific to GAS).

Format instrukcji



```
                    .text
0000 48656C6C   msg: .ascii "Hello, world!\n"
     6F2C2077
     6F726C64
     210A
                 msgLen = . - msg
                 .global _start

                 _start:

000e B804000000   mov $4, %eax
0013 BB01000000   mov $1, %ebx
0018 B900000000   mov $msg     , %ecx
001d BA0E000000   mov $msgLen  , %edx
0022 CD80         int $0x80

0024 B801000000   mov $1          , %eax
0029 BB00000000   mov $0, %ebx
002e CD80         int $0x80
```

Indeks

Chapter 4 Instruction Set Reference, N-Z 1073
4.1 Imm8 Control Byte Operation for PCMPESTRI / PCMPESTRM / ... 1073
4.2 Instructions (N-Z) 1077
NEG—Two's Complement Negation 1078
NOP—No Operation 1080
NOT—One's Complement Negation 1081
OR—Logical Inclusive OR 1083
ORPD—Bitwise Logical OR of Double-Precision Floating-Point ... 1085
ORPS—Bitwise Logical OR of Single-Precision Floating-Point V... 1087
OUT—Output to Port 1089
OUTS/OUTSB/OUTSW/OUTSD—Output String to Port 1091
PABSB/PABSW/PABSD — Packed Absolute Value 1095
PACKSSWB/PACKSSDW—Pack with Signed Saturation 1099
PACKUSDW — Pack with Unsigned Saturation 1104
PACKUSWB—Pack with Unsigned Saturation 1107
PADDB/PADDW/PADDD—Add Packed Integers 1110
PADDQ—Add Packed Quadword Integers 1114
PADDSB/PADDSW—Add Packed Signed Integers with Signed ... 1116
PADDUSB/PADDUSW—Add Packed Unsigned Integers with Un... 1119
PALIGNR — Packed Align Right 1122
PAND—Logical AND 1125
PANDN—Logical AND NOT 1127
PAUSE—Spin Loop Hint 1129
PAVGB/PAVGW—Average Packed Integers 1130
PBLENDVB — Variable Blend Packed Bytes 1133
PBLENDW — Blend Packed Words 1137
PCLMULQDQ - Carry-Less Multiplication Quadword 1140
PCMPEQB/PCMPEQW/PCMPEQD— Compare Packed Data for E... 1143
PCMPEQQ — Compare Packed Qword Data for Equal 1147
PCMPESTRI — Packed Compare Explicit Length Strings, Retur... 1149
PCMPESTRM — Packed Compare Explicit Length Strings, Ret... 1151
PCMPGTB/PCMPGTW/PCMPGTD—Compare Packed Signed Int... 1153
PCMPGTQ — Compare Packed Data for Greater Than 1157
PCMPISTRI — Packed Compare Implicit Length Strings, Retur... 1159
PCMPISTRM — Packed Compare Implicit Length Strings, Retu... 1161
PDEP — Parallel Bits Deposit 1163
PEXT — Parallel Bits Extract 1165
PEXTRB/PEXTRD/PEXTRQ — Extract Byte/Dword/Qword 1167
PEXTRW—Extract Word 1170
PHADDW/PHADDD — Packed Horizontal Add 1173
PHADDSW — Packed Horizontal Add and Saturate 1177
PHMINPOSUW — Packed Horizontal Word Minimum 1179
PHSUBW/PHSUBD — Packed Horizontal Subtract 1181
PHSUBSW — Packed Horizontal Subtract and Saturate 1184
PINSRB/PINSRD/PINSRQ — Insert Byte/Dword/Qword 1186
PINSRW—Insert Word 1188
PMADDUBSW — Multiply and Add Packed Signed and Unsign... 1190
PMADDWD—Multiply and Add Packed Integers 1192

NOP—No Operation

Opcode	Instruction	Op/ En	64-Bit Mode	Compat/ Leg Mode	Description
90	NOP	NP	Valid	Valid	One byte no-operation instruction.
0F 1F /0	NOP r/m16	M	Valid	Valid	Multi-byte no-operation instruction.
0F 1F /0	NOP r/m32	M	Valid	Valid	Multi-byte no-operation instruction.

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
NP	NA	NA	NA	NA
M	ModRM:r/m (r)	NA	NA	NA

Description

This instruction performs no operation. It is a one-byte or multi-byte NOP that takes up space in the instruction stream but does not impact machine context, except for the EIP register.

The multi-byte form of NOP is available on processors with model encoding:

- CPUID.01H.EAX[Bytes 11:8] = 0110B or 1111B

The multi-byte NOP instruction does not alter the content of a register and will not issue a memory operation. The instruction's operation is the same in non-64-bit modes and 64-bit mode.

Operation

The one-byte NOP instruction is an alias mnemonic for the XCHG (E)AX, (E)AX instruction.

The multi-byte NOP instruction performs no operation on supported processors and generates undefined opcode exception on processors that do not support the multi-byte NOP instruction.

The memory operand form of the instruction allows software to create a byte sequence of "no operation" as one instruction. For situations where multiple-byte NOPs are needed, the recommended operations (32-bit mode and 64-bit mode) are:

Table 4-9. Recommended Multi-Byte Sequence of NOP Instruction

Length	Assembly	Byte Sequence
2 bytes	66 NOP	66 90H
3 bytes	NOP DWORD ptr [EAX]	0F 1F 00H
4 bytes	NOP DWORD ptr [EAX + 00H]	0F 1F 40 00H
5 bytes	NOP DWORD ptr [EAX + EAX*1 + 00H]	0F 1F 44 00 00H
6 bytes	66 NOP DWORD ptr [EAX + EAX*1 + 00H]	66 0F 1F 44 00 00H
7 bytes	NOP DWORD ptr [EAX + 00000000H]	0F 1F 80 00 00 00 00H
8 bytes	NOP DWORD ptr [EAX + EAX*1 + 00000000H]	0F 1F 84 00 00 00 00 00H
9 bytes	66 NOP DWORD ptr [EAX + EAX*1 + 00000000H]	66 0F 1F 84 00 00 00 00 00H

Flags Affected

None.

Exceptions (All Operating Modes)

#UD	If the LOCK prefix is used.

2

Mnemoniczny opis działań na poziomie architektury rzeczywistej (ISA)

Mnemonik	Pełna nazwa	Rodzaj operacji	Typ
add	*add*	dodaj	A
sub	*subtract*	odejmij	A
mul, mpy	*multiply*	pomnóż	A
div	*divide*	podziel	A
cmp, cp	*compare*	porównaj (określ relację)	A
test	*test*	porównaj (sprawdź zgodność)	L
and	*and*	iloczyn logiczny	L
or	*or*	suma logiczna	L
xor	*exclusive-or*	suma wyłączająca (modulo 2)	L
inc / dec	*increment/decrement*	zwiększ / zmniejsz	K
shr / shl	*shift right/left*	przesuń w prawo / lewo	K
rr / rl	*rotate right/left*	przesuń cyklicznie w prawo / lewo	K
mov(e)	*move*	kopiuj (przemieść)	T
ld, load	*load*	pobierz (z pamięci) do rejestru	T
st, store	*store*	zapisz do pamięci (z rejestru)	T
b*cc*, j*cc*	*branch* | *conditional jump*	rozgałęziaj (wybierz ścieżkę)	T
call, jsr	*call procedure*	wywołaj procedurę	T

Instrukcje

- Przesłania danych (MOV, XCHG, PUSH, POP)
- Operacje logiczne (NOT, NEG, OR, AND, XOR, BT, CLC, STC)
- Zmiana formatu (CWD, CDQ, CQO, przesunięcia SAR, SHR, SAL, SHL, rotacje ROR, RCR, ROL, RCL)
- Operacje arytmetyczne (ADD, ADC, SUB, SBB, MUL, IMUL, DIV, IDIV, INC, DEC)
- Rozgałęzienia (JMP, CALL, RET, skoki warunkowe CMP/TEST + Jcc)

Podstawowy zestaw użytecznych instrukcji (IA-32/IA-32e)

mnemoniki (z pominięciem atrybutu rozmiaru **b/l**)

mov (ang. *move*) – ~~przemieść~~, kopiuj
xor / and / or / not – operacje logiczne
clc / stc (ang. *clear/set carry*) – ustaw przeniesienie CF=0/1
adc (ang. *add with carry*) – dodaj uwzględniając przeniesienie
sbb (ang. *subtract with borrow*) – odejmij uwzględniając przeniesienie
cmp (ang. *compare*) – porównaj i ustaw flagi
(i)mul (ang. (*integer*) *multiply*) – wymnóż jak całkowite / naturalne
(i)div (ang. (*integer*) *divide*) – podziel jak całkowite / naturalne
inc / dec (ang. *increment/decrement*) – zwiększ/zmniejsz o 1
loop (ang. *loop*) – zapętlaj dopóki licznik (cx/ecx/rcx) ≠ 0
j*cond* (ang. *jump conditional*) – skocz jeśli warunek *cond* prawdziwy
push / pop (ang. *push/pop*) – kopiuj na stos / ze stosu programowego
call (ang. *call*) – przekaż sterowanie do funkcji i zapamiętaj adres powrotu
ret (ang. *return*) – zwróć sterowanie z procedury używając adresu powrotu
lea (ang. *load effective address*) – wpisz adres obliczony wg trybu adresowania

MOV—Move

Opcode	Instruction	Op/ En	64-Bit Mode	Compat/ Leg Mode	Description
88 /r	MOV *r/m8,r8*	MR	Valid	Valid	Move *r8* to *r/m8*.
REX + 88 /r	MOV *r/m8***,r8****	MR	Valid	N.E.	Move *r8* to *r/m8*.
89 /r	MOV *r/m16,r16*	MR	Valid	Valid	Move *r16* to *r/m16*.
89 /r	MOV *r/m32,r32*	MR	Valid	Valid	Move *r32* to *r/m32*.
REX.W + 89 /r	MOV *r/m64,r64*	MR	Valid	N.E.	Move *r64* to *r/m64*.
8A /r	MOV *r8,r/m8*	RM	Valid	Valid	Move *r/m8* to *r8*.
REX + 8A /r	MOV *r8***,r/m8****	RM	Valid	N.E.	Move *r/m8* to *r8*.
8B /r	MOV *r16,r/m16*	RM	Valid	Valid	Move *r/m16* to *r16*.
8B /r	MOV *r32,r/m32*	RM	Valid	Valid	Move *r/m32* to *r32*.
REX.W + 8B /r	MOV *r64,r/m64*	RM	Valid	N.E.	Move *r/m64* to *r64*.
8C /r	MOV *r/m16,Sreg***	MR	Valid	Valid	Move segment register to *r/m16*.
REX.W + 8C /r	MOV *r/m64,Sreg***	MR	Valid	Valid	Move zero extended 16-bit segment register to *r/m64*.
8E /r	MOV *Sreg,r/m16***	RM	Valid	Valid	Move *r/m16* to segment register.
REX.W + 8E /r	MOV *Sreg,r/m64***	RM	Valid	Valid	Move *lower 16 bits of r/m64* to segment register.
A0	MOV AL,*moffs8**	FD	Valid	Valid	Move byte at (*seg:offset*) to AL.
REX.W + A0	MOV AL,*moffs8**	FD	Valid	N.E.	Move byte at (*offset*) to AL.
A1	MOV AX,*moffs16**	FD	Valid	Valid	Move word at (*seg:offset*) to AX.
A1	MOV EAX,*moffs32**	FD	Valid	Valid	Move doubleword at (*seg:offset*) to EAX.
REX.W + A1	MOV RAX,*moffs64**	FD	Valid	N.E.	Move quadword at (*offset*) to RAX.
A2	MOV *moffs8*,AL	TD	Valid	Valid	Move AL to (*seg:offset*).
REX.W + A2	MOV *moffs8****,AL	TD	Valid	N.E.	Move AL to (*offset*).
A3	MOV *moffs16**,AX	TD	Valid	Valid	Move AX to (*seg:offset*).
A3	MOV *moffs32**,EAX	TD	Valid	Valid	Move EAX to (*seg:offset*).
REX.W + A3	MOV *moffs64**,RAX	TD	Valid	N.E.	Move RAX to (*offset*).
B0+ *rb ib*	MOV *r8, imm8*	OI	Valid	Valid	Move *imm8* to *r8*.
REX + B0+ *rb ib*	MOV *r8***, imm8*	OI	Valid	N.E.	Move *imm8* to *r8*.
B8+ *rw iw*	MOV *r16, imm16*	OI	Valid	Valid	Move *imm16* to *r16*.
B8+ *rd id*	MOV *r32, imm32*	OI	Valid	Valid	Move *imm32* to *r32*.
REX.W + B8+ *rd io*	MOV *r64, imm64*	OI	Valid	N.E.	Move *imm64* to *r64*.
C6 */0 ib*	MOV *r/m8, imm8*	MI	Valid	Valid	Move *imm8* to *r/m8*.
REX + C6 */0 ib*	MOV *r/m8***, imm8*	MI	Valid	N.E.	Move *imm8* to *r/m8*.
C7 */0 iw*	MOV *r/m16, imm16*	MI	Valid	Valid	Move *imm16* to *r/m16*.
C7 */0 id*	MOV *r/m32, imm32*	MI	Valid	Valid	Move *imm32* to *r/m32*.
REX.W + C7 */0 io*	MOV *r/m64, imm32*	MI	Valid	N.E.	Move *imm32 sign extended to 64-bits* to *r/m64*.

(IA-32/IA-32e) kopiowanie

mov (ang. *move*) – ~~przemieść~~, kopiuj

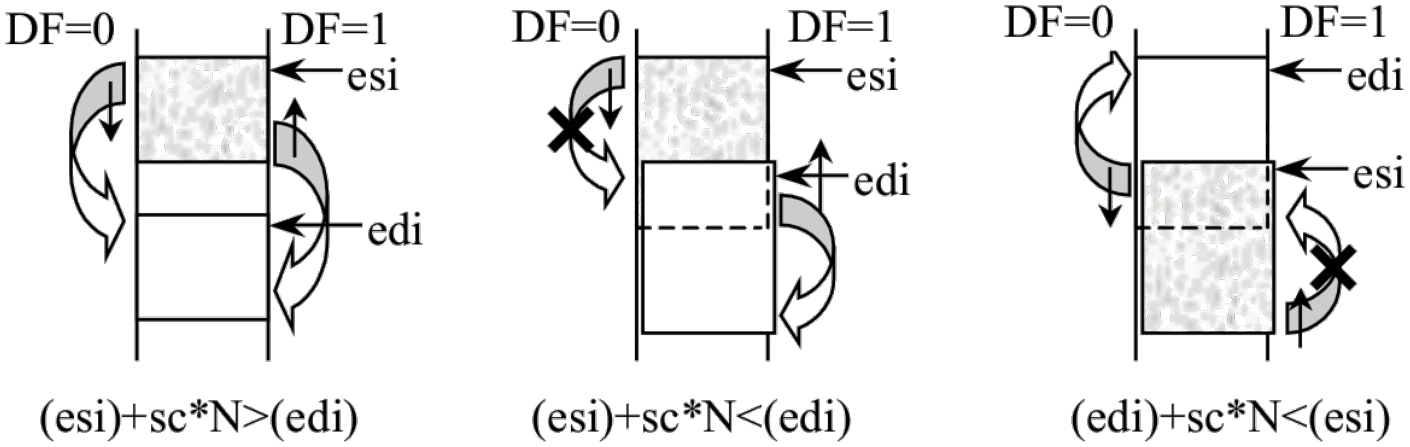
mov[.] arg_swobodny, arg_docelowy
działanie: **arg_docelowy := arg_swobodny**

movl $-1, %edx **- (edx)=11….1 (w rejestrze edx same „1”)**
movb $‘G’, %bl **- (bl)= 010 00111 (w rejestrze bl kod ASCII litery G**
movl (%ecx), %eax **- zawartość słowa o adresie (ecx) skopiowana do rejestru eax**

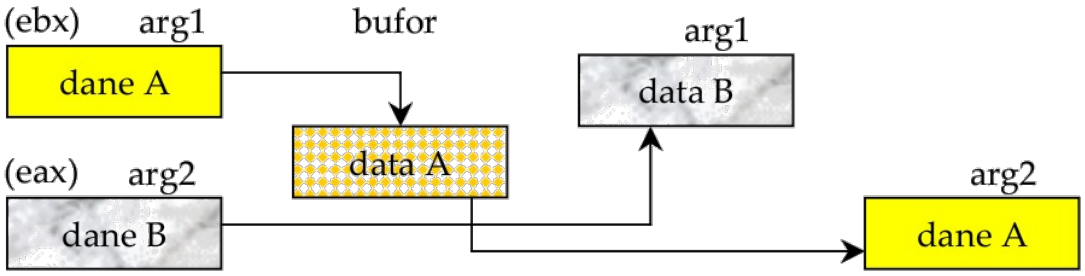
xchg (ang. *exchange*) – wymień, kopiuj wzajemnie

xchgl %eax, %ebx **- zawartość rejestru eax skopiuj do rejestru ebx a zawartość ebx do eax**

Kopiowanie



Transfer łańcucha słów **movs**



wymiana bez bufora:

```
xor %eax, %ebx
xor %ebx, %eax
xor %eax, %ebx
```

Kopiowanie wzajemne (z wymianą): **xchg** %ebx, %eax

ADC—Add with Carry

Opcode	Instruction	Op/ En	64-bit Mode	Compat/ Leg Mode	Description
14 *ib*	ADC AL, *imm8*	I	Valid	Valid	Add with carry *imm8* to AL.
15 *iw*	ADC AX, *imm16*	I	Valid	Valid	Add with carry *imm16* to AX.
15 *id*	ADC EAX, *imm32*	I	Valid	Valid	Add with carry *imm32* to EAX.
REX.W + 15 *id*	ADC RAX, *imm32*	I	Valid	N.E.	Add with carry *imm32 sign extended to 64-bits* to RAX.
80 /2 *ib*	ADC *r/m8, imm8*	MI	Valid	Valid	Add with carry *imm8* to *r/m8.*
REX + 80 /2 *ib*	ADC *r/m8**, *imm8*	MI	Valid	N.E.	Add with carry *imm8* to *r/m8.*
81 /2 *iw*	ADC *r/m16, imm16*	MI	Valid	Valid	Add with carry *imm16* to *r/m16.*
81 /2 *id*	ADC *r/m32, imm32*	MI	Valid	Valid	Add with CF *imm32* to *r/m32.*
REX.W + 81 /2 *id*	ADC *r/m64, imm32*	MI	Valid	N.E.	Add with CF *imm32* sign extended to 64-bits to *r/m64.*
83 /2 *ib*	ADC *r/m16, imm8*	MI	Valid	Valid	Add with CF sign-extended *imm8* to *r/m16.*
83 /2 *ib*	ADC *r/m32, imm8*	MI	Valid	Valid	Add with CF sign-extended *imm8* into *r/m32.*
REX.W + 83 /2 *ib*	ADC *r/m64, imm8*	MI	Valid	N.E.	Add with CF sign-extended *imm8* into *r/m64.*
10 /*r*	ADC *r/m8, r8*	MR	Valid	Valid	Add with carry byte register to *r/m8.*
REX + 10 /*r*	ADC *r/m8**, *r8**	MR	Valid	N.E.	Add with carry byte register to *r/m64.*
11 /*r*	ADC *r/m16, r16*	MR	Valid	Valid	Add with carry *r16* to *r/m16.*
11 /*r*	ADC *r/m32, r32*	MR	Valid	Valid	Add with CF *r32* to *r/m32.*
REX.W + 11 /*r*	ADC *r/m64, r64*	MR	Valid	N.E.	Add with CF *r64* to *r/m64.*
12 /*r*	ADC *r8, r/m8*	RM	Valid	Valid	Add with carry *r/m8* to byte register.
REX + 12 /*r*	ADC *r8**, *r/m8**	RM	Valid	N.E.	Add with carry *r/m64* to byte register.
13 /*r*	ADC *r16, r/m16*	RM	Valid	Valid	Add with carry *r/m16* to *r16.*
13 /*r*	ADC *r32, r/m32*	RM	Valid	Valid	Add with CF *r/m32* to *r32.*
REX.W + 13 /*r*	ADC *r64, r/m64*	RM	Valid	N.E.	Add with CF *r/m64* to *r64.*

ADC

- Adds the destination operand (first operand), the source operand (second operand), and the carry (CF) flag and stores the result in the destination operand.
- The destination operand can be a register or a memory location; the source operand can be an immediate, a register, or a memory location. (However, two memory operands cannot be used in one instruction.)
- The state of the CF flag represents a carry from a previous addition.
- When an immediate value is used as an operand, it is sign-extended to the length of the destination operand format.

ADC

- The ADC instruction does not distinguish between signed or unsigned operands. Instead, the processor evaluates the result for both data types and sets the OF and CF flags to indicate a carry in the signed or unsigned result, respectively.
- The SF flag indicates the sign of the signed result.
- The ADC instruction is usually executed as part of a multibyte or multiword addition in which an ADD instruction is followed by an ADC instruction.

ADC

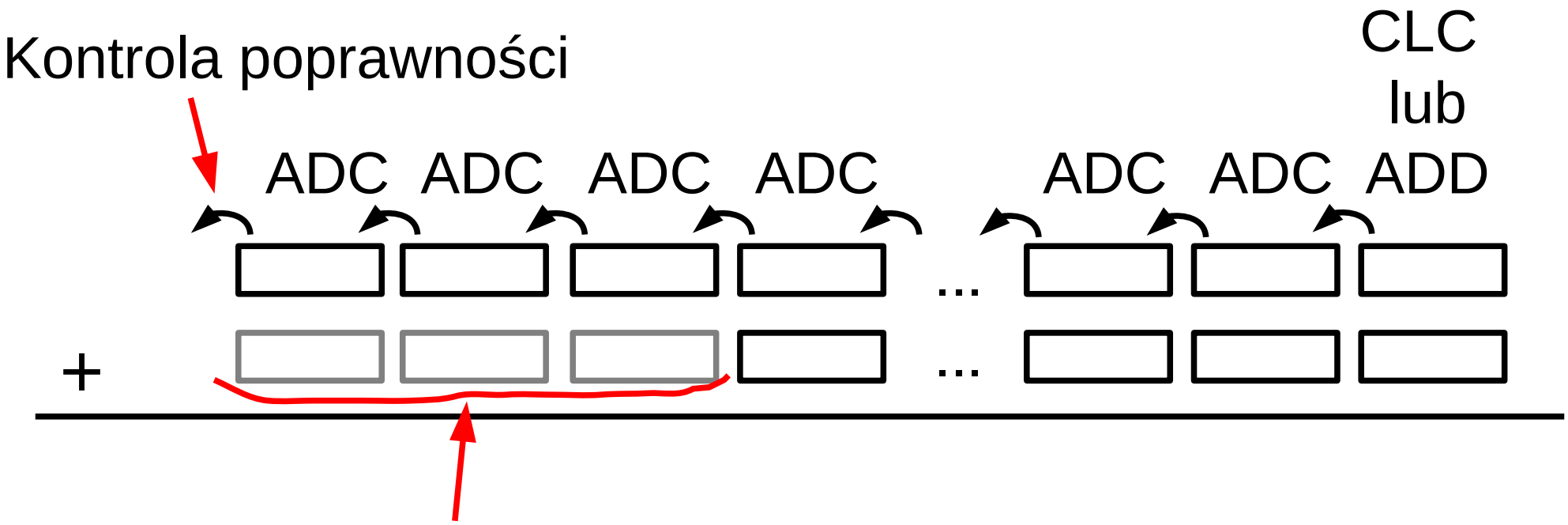
Flags Affected

- The OF, SF, ZF, AF, CF, and PF flags are set according to the result.

ADD—Add

Opcode	Instruction	Op/ En	64-bit Mode	Compat/ Leg Mode	Description
04 *ib*	ADD AL, *imm8*	I	Valid	Valid	Add *imm8* to AL.
05 *iw*	ADD AX, *imm16*	I	Valid	Valid	Add *imm16* to AX.
05 *id*	ADD EAX, *imm32*	I	Valid	Valid	Add *imm32* to EAX.
REX.W + 05 *id*	ADD RAX, *imm32*	I	Valid	N.E.	Add *imm32 sign-extended to 64-bits* to RAX.
80 /0 *ib*	ADD *r/m8, imm8*	MI	Valid	Valid	Add *imm8* to *r/m8.*
REX + 80 /0 *ib*	ADD *r/m8**, *imm8*	MI	Valid	N.E.	Add *sign*-extended *imm8* to *r/m64.*
81 /0 *iw*	ADD *r/m16, imm16*	MI	Valid	Valid	Add *imm16* to *r/m16.*
81 /0 *id*	ADD *r/m32, imm32*	MI	Valid	Valid	Add *imm32* to *r/m32.*
REX.W + 81 /0 *id*	ADD *r/m64, imm32*	MI	Valid	N.E.	Add *imm32 sign-extended to 64-bits* to *r/m64.*
83 /0 *ib*	ADD *r/m16, imm8*	MI	Valid	Valid	Add *sign*-extended *imm8* to *r/m16.*
83 /0 *ib*	ADD *r/m32, imm8*	MI	Valid	Valid	Add *sign*-extended *imm8* to *r/m32.*
REX.W + 83 /0 *ib*	ADD *r/m64, imm8*	MI	Valid	N.E.	Add *sign*-extended *imm8* to *r/m64.*
00 /*r*	ADD *r/m8, r8*	MR	Valid	Valid	Add *r8* to *r/m8.*
REX + 00 /*r*	ADD *r/m8**, *r8**	MR	Valid	N.E.	Add *r8* to *r/m8.*
01 /*r*	ADD *r/m16, r16*	MR	Valid	Valid	Add *r16* to *r/m16.*
01 /*r*	ADD *r/m32, r32*	MR	Valid	Valid	Add r32 to *r/m32.*
REX.W + 01 /*r*	ADD *r/m64, r64*	MR	Valid	N.E.	Add r64 to *r/m64.*
02 /*r*	ADD *r8, r/m8*	RM	Valid	Valid	Add *r/m8* to *r8.*
REX + 02 /*r*	ADD *r8**, *r/m8**	RM	Valid	N.E.	Add *r/m8* to *r8.*
03 /*r*	ADD *r16, r/m16*	RM	Valid	Valid	Add *r/m16* to *r16.*
03 /*r*	ADD *r32, r/m32*	RM	Valid	Valid	Add *r/m32* to *r32.*
REX.W + 03 /*r*	ADD *r64, r/m64*	RM	Valid	N.E.	Add *r/m64* to *r64.*

Dodawanie wielopozycyjne NB/U2



Rozszerzenie lewostronne, na podstawie flagi SF lub

CWD/CDQ/CQO—Convert Word to Doubleword/Convert Doubleword to Quadword

Opcode	Instruction	Op/ En	64-Bit Mode	Compat/ Leg Mode	Description
99	CWD	NP	Valid	Valid	DX:AX ← sign-extend of AX.
99	CDQ	NP	Valid	Valid	EDX:EAX ← sign-extend of EAX.
REX.W + 99	CQO	NP	Valid	N.E.	RDX:RAX← sign-extend of RAX.

Przykład: wytworzenie liczby przeciwnej do danej dużej liczby (N słów)

inverse:	push %ebp	# .....
	movl %esp, %ebp	
	movl 8(%ebp), %ebx	# adres liczby (tablicy)
	movl 12(%ebp), %ecx	# rozmiar liczby
	movl $-1, %esi	# –X = not(X)+ulp
	stc	# ustawienie CF=1 (clc)
pocz:	inc %esi	#
	not (%ebx, %esi,4)	
	adc $0, (%ebx, %esi,4)	
	jnc koniec	
	loop pocz	# licznik pętli w %ecx
koniec:	movl %ebp, %esp	
	pop %ebp	
	ret	

rozwiązanie alternatywne to (tylko fragment zacieniony)

	movl $-1, %esi	# –X = 0–X
	clc	# ustawienie CF=0
pocz:	inc %esi	#
	movl $0, %eax	
	sbbl (%ebx,%esi,4), %eax	# (0–(%ebx, %esi,4) do eax
	movl %eax, (%ebx,%esi,4)	
	loop pocz	# licznik pętli w %ecx

Przykład: dekrementacja dużej liczby (N×32 b)

(etykieta)	Rozkaz	Komentarz
ldecr:	push %ebp	# przez dodanie (1) (czyli –1)
	movl %esp, %ebp	
	movl 8(%ebp), %ebx	# adres zmiennej
	movl 12(%ebp), %ecx	# rozmiar zmiennej
	clc	# ustawienie CF=0
pocz:	inc %esi	
	adcl $–1, (%ebx,%esi,4)	
	jnc end	
end:	movl %ebp, %esp	
	pop %ebp	
	ret	

rozwiązanie alternatywne to (tylko fragment zacieniony):

	stc	# ustawienie CF=1
pocz:	inc %esi	#
	movl $0, %eax	
	sbbl (%ebx,%esi,4), %eax	# (adc $-1, (%ebx, %esi,4)
	movl %eax, (%ebx,%esi,4)	
	jnc end	

(IA-32/IA-32e) indeksowanie

inc (ang. *increment*) – zwiększ (indeks) o 1
dec (ang. *decrement*) – zmniejsz (indeks) o 1

! ustawiane kody w rejestrze flag (F): OF, ZF, SF, bez zmiany CF

incl %eax	**- zwiększ o 1 zawartość rejestru eax, nie zmienia CF**
add $1, %eax	**- zwiększ o 1 zawartość rejestru eax, zmienia CF**
sub $-1, %eax	**- zmniejsz o 1 zawartość rejestru eax, zmienia CF**
decb %bl	**- zmniejsz o 1 zawartość rejestru bl, nie zmienia CF**
add $-1, %eax	**- zmniejsz o 1 zawartość rejestru eax, zmienia CF**
sub $1, %eax	**- zmniejsz o 1 zawartość rejestru eax, zmienia CF**

lea (ang. *load effective address*) – wpisz obliczony adres do wskazanego rejestru

lea (%esi,%eax,4), %esi	**-** zwiększ wartość rejestru esi o 4*(eax), nie zmienia F
lea (%edi,%ecx,1)	**-** edi:= (edi) + (ecx), indeksacja edi skokiem ecx
lea skok(%edi)	**-** edi:= (edi) +skok, indeksacja edi skokiem ecx

MUL

MUL—Unsigned Multiply

Opcode	Instruction	Op/ En	64-Bit Mode	Compat/ Leg Mode	Description
F6 /4	MUL *r/m8*	M	Valid	Valid	Unsigned multiply (AX ← AL * *r/m8*).
REX + F6 /4	MUL *r/m8*[*]	M	Valid	N.E.	Unsigned multiply (AX ← AL * *r/m8*).
F7 /4	MUL *r/m16*	M	Valid	Valid	Unsigned multiply (DX:AX ← AX * *r/m16*).
F7 /4	MUL *r/m32*	M	Valid	Valid	Unsigned multiply (EDX:EAX ← EAX * *r/m32*).
REX.W + F7 /4	MUL *r/m64*	M	Valid	N.E.	Unsigned multiply (RDX:RAX ← RAX * *r/m64*).

Table 3-65. MUL Results

Operand Size	Source 1	Source 2	Destination
Byte	AL	r/m8	AX
Word	AX	r/m16	DX:AX
Doubleword	EAX	r/m32	EDX:EAX
Quadword	RAX	r/m64	RDX:RAX

MUL

- Performs an unsigned multiplication of the first operand (destination operand) and the second operand (source operand) and stores the result in the destination operand.
- The destination operand is an implied operand located in register AL, AX or EAX (depending on the size of the operand); the source operand is located in a general purpose register or a memory location. The action of this instruction and the location of the result depends on the opcode and the operand size as shown in Table 3-65.
- The result is stored in register AX, register pair DX:AX, or register pair EDX:EAX (depending on the operand size), with the high-order bits of the product contained in register AH, DX, or EDX, respectively.
- If the high-order bits of the product are 0, the CF and OF flags are cleared; otherwise, the flags are set.
- In 64-bit mode, the instruction’s default operation size is 32 bits. Use of the REX.R prefix permits access to additional registers (R8-R15). Use of the REX.W prefix promotes operation to 64 bits.

MUL

Flags Affected

- The OF and CF flags are set to 0 if the upper half of the result is 0; otherwise, they are set to 1.
- The SF, ZF, AF, and PF flags are **undefined**.

IMUL—Signed Multiply

Opcode	Instruction	Op/ En	64-Bit Mode	Compat/ Leg Mode	Description
F6 /5	IMUL *r/m8**	M	Valid	Valid	AX← AL ∗ *r/m* byte.
F7 /5	IMUL *r/m16*	M	Valid	Valid	DX:AX ← AX ∗ *r/m* word.
F7 /5	IMUL *r/m32*	M	Valid	Valid	EDX:EAX ← EAX ∗ *r/m*32.
REX.W + F7 /5	IMUL *r/m64*	M	Valid	N.E.	RDX:RAX ← RAX ∗ *r/m*64.
0F AF /*r*	IMUL *r16, r/m16*	RM	Valid	Valid	word register ← word register ∗ *r/m*16.
0F AF /*r*	IMUL *r32, r/m32*	RM	Valid	Valid	doubleword register ← doubleword register ∗ *r/m32.*
REX.W + 0F AF /*r*	IMUL *r64, r/m64*	RM	Valid	N.E.	Quadword register ← Quadword register ∗ *r/m64.*
6B /*r ib*	IMUL *r16, r/m16, imm8*	RMI	Valid	Valid	word register ← *r/m16* ∗ sign-extended immediate byte.
6B /*r ib*	IMUL *r32, r/m32, imm8*	RMI	Valid	Valid	doubleword register ← *r/m32* ∗ sign-extended immediate byte.
REX.W + 6B /*r ib*	IMUL *r64, r/m64, imm8*	RMI	Valid	N.E.	Quadword register ← *r/m64* ∗ sign-extended immediate byte.
69 /*r iw*	IMUL *r16, r/m16, imm16*	RMI	Valid	Valid	word register ← *r/m16* ∗ immediate word.
69 /*r id*	IMUL *r32, r/m32, imm32*	RMI	Valid	Valid	doubleword register ← *r/m32* ∗ immediate doubleword.
REX.W + 69 /*r id*	IMUL *r64, r/m64, imm32*	RMI	Valid	N.E.	Quadword register ← *r/m64* ∗ immediate doubleword.

(IA-32/IA-32e) mnożenie

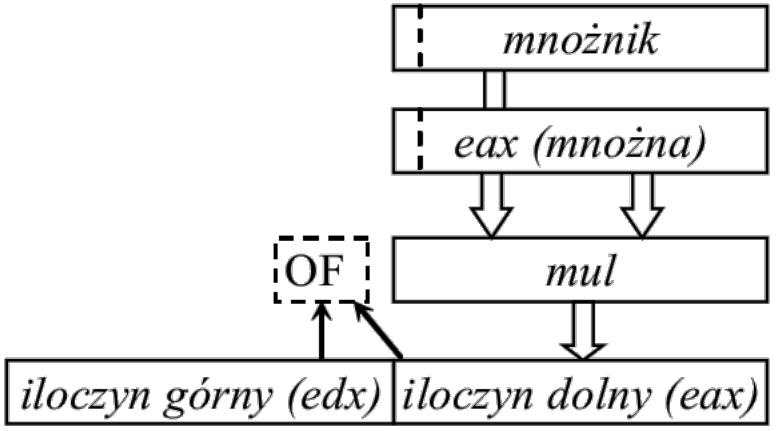
mul (ang. *multiply*) – wymnóż jak liczby naturalne

imul (ang. *integer multiply*) – wymnóż jak liczby całkowite (ze znakiem)

! ustawiane kody w rejestrze flag (F): CF, OF

iloczyn liczb 1-cyfrowych jest 2-cyfrowy

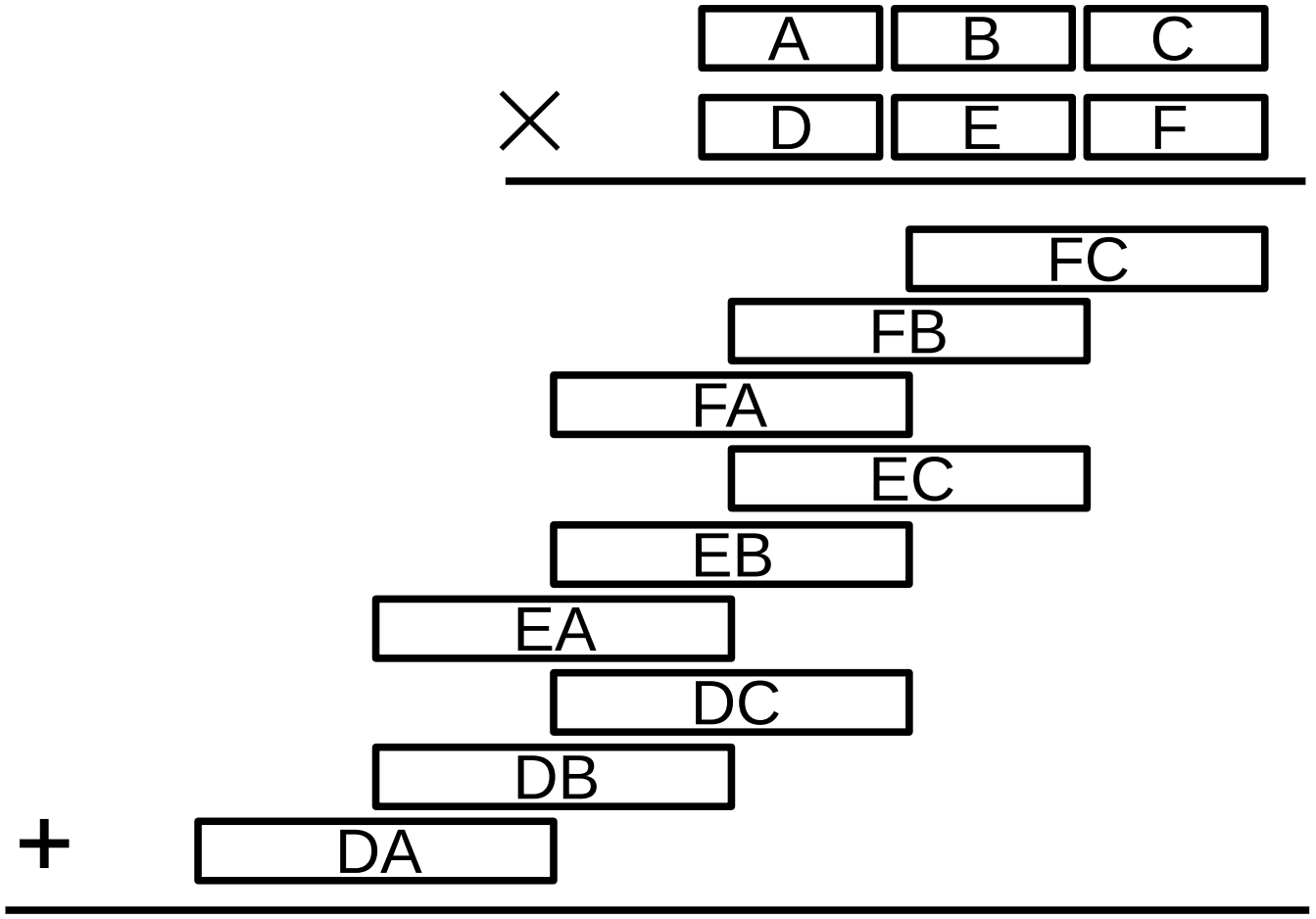
[i]mul arg ; **(edx:eax):=(eax)*arg**



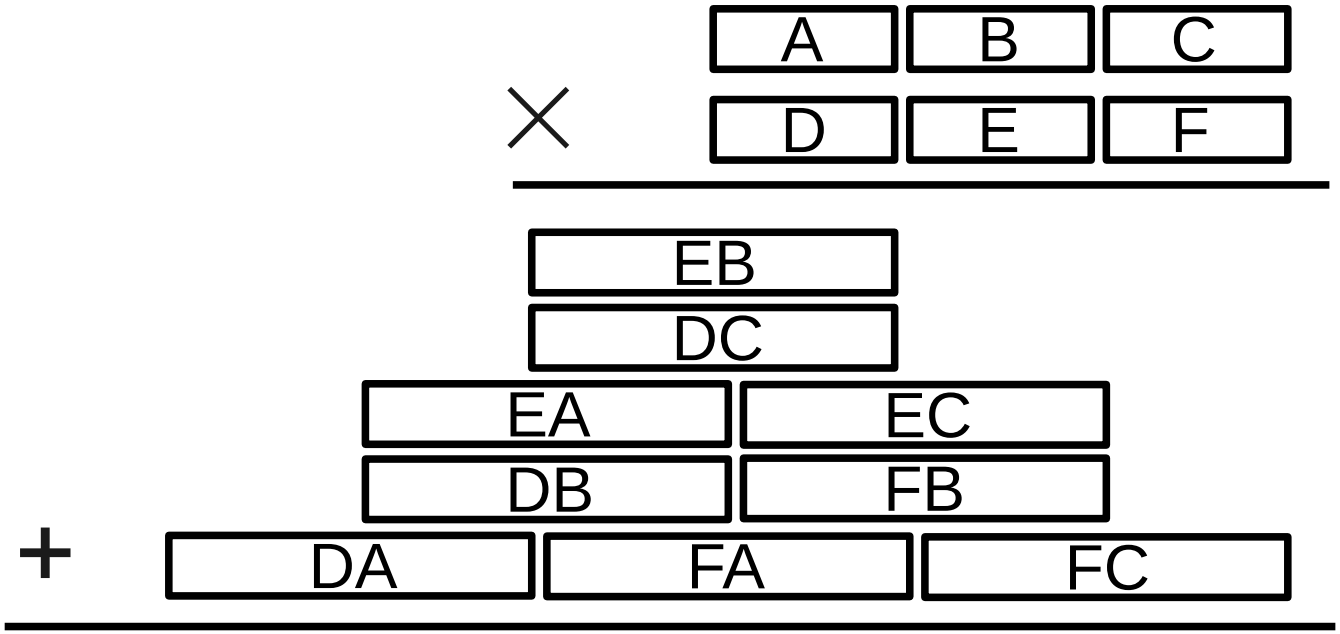
[i]mull %ecx ; **(edx:eax):=(eax)*(ecx)**

imull %ecx, $const ; **(edx:eax):=(eax)*(ecx)**

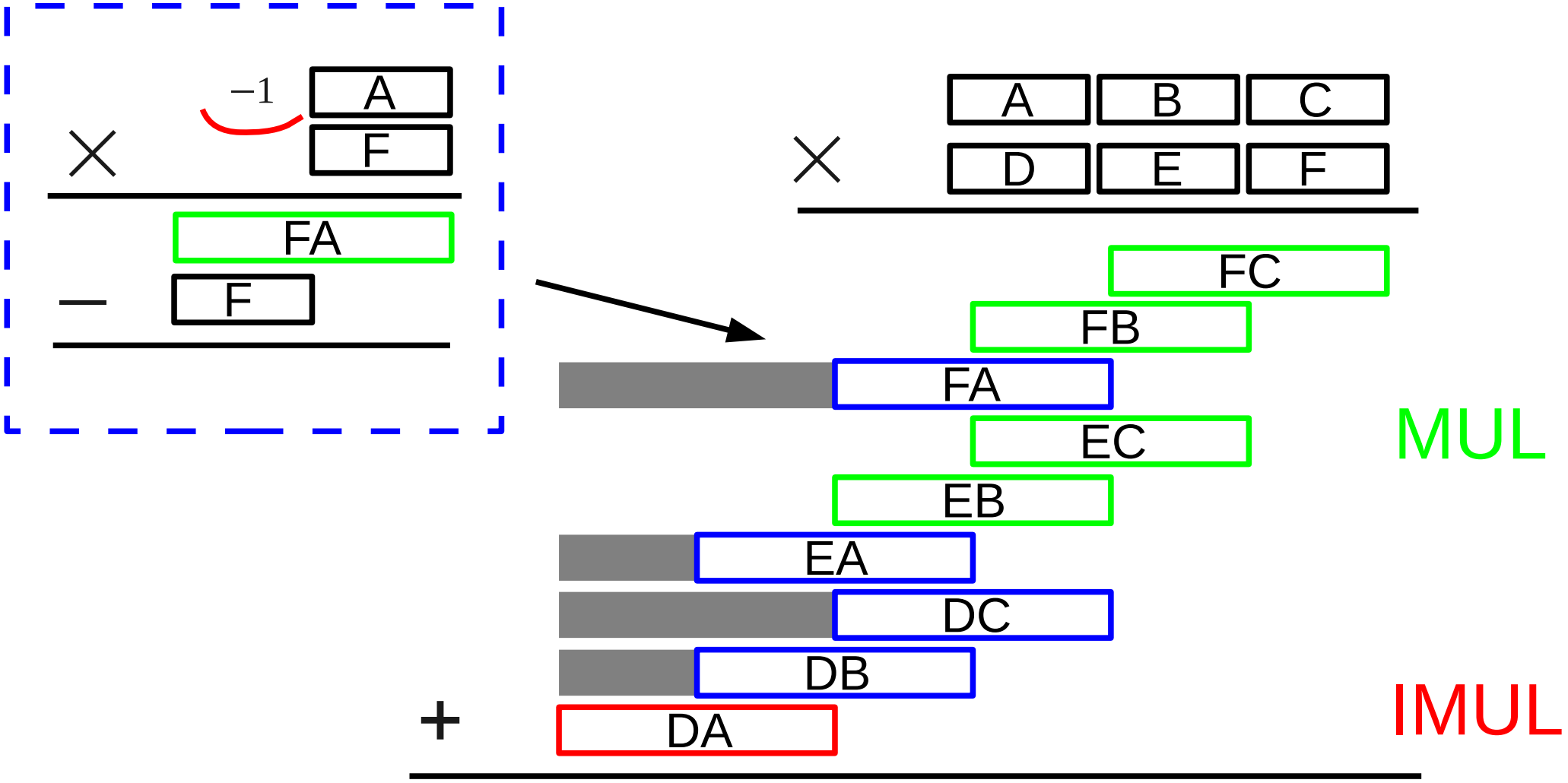
Mnożenie wielopozycyjne NB



Mnożenie wielopozycyjne NB



Mnożenie wielopozycyjne U2



DIV—Unsigned Divide

Opcode	Instruction	Op/ En	64-Bit Mode	Compat/ Leg Mode	Description
F6 /6	DIV *r/m8*	M	Valid	Valid	Unsigned divide AX by *r/m8*, with result stored in AL ← Quotient, AH ← Remainder.
REX + F6 /6	DIV *r/m8*[*]	M	Valid	N.E.	Unsigned divide AX by *r/m8*, with result stored in AL ← Quotient, AH ← Remainder.
F7 /6	DIV *r/m16*	M	Valid	Valid	Unsigned divide DX:AX by *r/m16*, with result stored in AX ← Quotient, DX ← Remainder.
F7 /6	DIV *r/m32*	M	Valid	Valid	Unsigned divide EDX:EAX by *r/m32*, with result stored in EAX ← Quotient, EDX ← Remainder.
REX.W + F7 /6	DIV *r/m64*	M	Valid	N.E.	Unsigned divide RDX:RAX by *r/m64*, with result stored in RAX ← Quotient, RDX ←

Description

Divides unsigned the value in the AX, DX:AX, EDX:EAX, or RDX:RAX registers (dividend) by the source operand (divisor) and stores the result in the AX (AH:AL), DX:AX, EDX:EAX, or RDX:RAX registers. The source operand can be a general-purpose register or a memory location. The action of this instruction depends on the operand size (dividend/divisor). Division using 64-bit operand is available only in 64-bit mode.

Non-integral results are truncated (chopped) towards 0. The remainder is always less than the divisor in magnitude. Overflow is indicated with the #DE (divide error) exception rather than with the CF flag.

In 64-bit mode, the instruction's default operation size is 32 bits. Use of the REX.R prefix permits access to additional registers (R8-R15). Use of the REX.W prefix promotes operation to 64 bits. In 64-bit mode when REX.W is applied, the instruction divides the unsigned value in RDX:RAX by the source operand and stores the quotient in RAX, the remainder in RDX.

See the summary chart at the beginning of this section for encoding data and limits. See Table 3-24.

Table 3-24. DIV Action

Operand Size	Dividend	Divisor	Quotient	Remainder	Maximum Quotient
Word/byte	AX	r/m8	AL	AH	255
Doubleword/word	DX:AX	r/m16	AX	DX	65,535
Quadword/doubleword	EDX:EAX	r/m32	EAX	EDX	$2^{32} - 1$
Doublequadword/ quadword	RDX:RAX	r/m64	RAX	RDX	$2^{64} - 1$

DIV

Flags Affected

- The CF, OF, SF, ZF, AF, and PF flags are **undefined**.

IDIV—Signed Divide

Opcode	Instruction	Op/ En	64-Bit Mode	Compat/ Leg Mode	Description
F6 /7	IDIV *r/m8*	M	Valid	Valid	Signed divide AX by *r/m8*, with result stored in: AL ← Quotient, AH ← Remainder.
REX + F6 /7	IDIV *r/m8**	M	Valid	N.E.	Signed divide AX by *r/m8*, with result stored in AL ← Quotient, AH ← Remainder.
F7 /7	IDIV *r/m16*	M	Valid	Valid	Signed divide DX:AX by *r/m16*, with result stored in AX ← Quotient, DX ← Remainder.
F7 /7	IDIV *r/m32*	M	Valid	Valid	Signed divide EDX:EAX by *r/m32*, with result stored in EAX ← Quotient, EDX ← Remainder.
REX.W + F7 /7	IDIV *r/m64*	M	Valid	N.E.	Signed divide RDX:RAX by *r/m64*, with result stored in RAX ← Quotient, RDX ← Remainder.

Table 3-59. IDIV Results

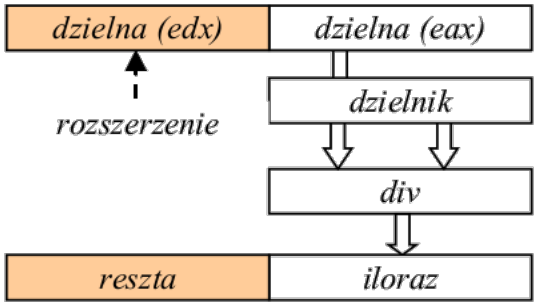
Operand Size	Dividend	Divisor	Quotient	Remainder	Quotient Range
Word/byte	AX	r/m8	AL	AH	–128 to +127
Doubleword/word	DX:AX	r/m16	AX	DX	–32,768 to +32,767
Quadword/doubleword	EDX:EAX	r/m32	EAX	EDX	-2^{31} to $2^{31} - 1$
Doublequadword/ quadword	RDX:RAX	r/m64	RAX	RDX	-2^{63} to $2^{63} - 1$

(IA-32/IA-32e) dzielenie

div (ang. *divide*) – podziel naturalne tworząc iloraz i resztę
idiv (ang. *integer divide*) – podziel całkowite tworząc iloraz i resztę

– jeśli iloraz zbyt duży do zapisu w rejestrze **a** – błąd *Divide Overflow*

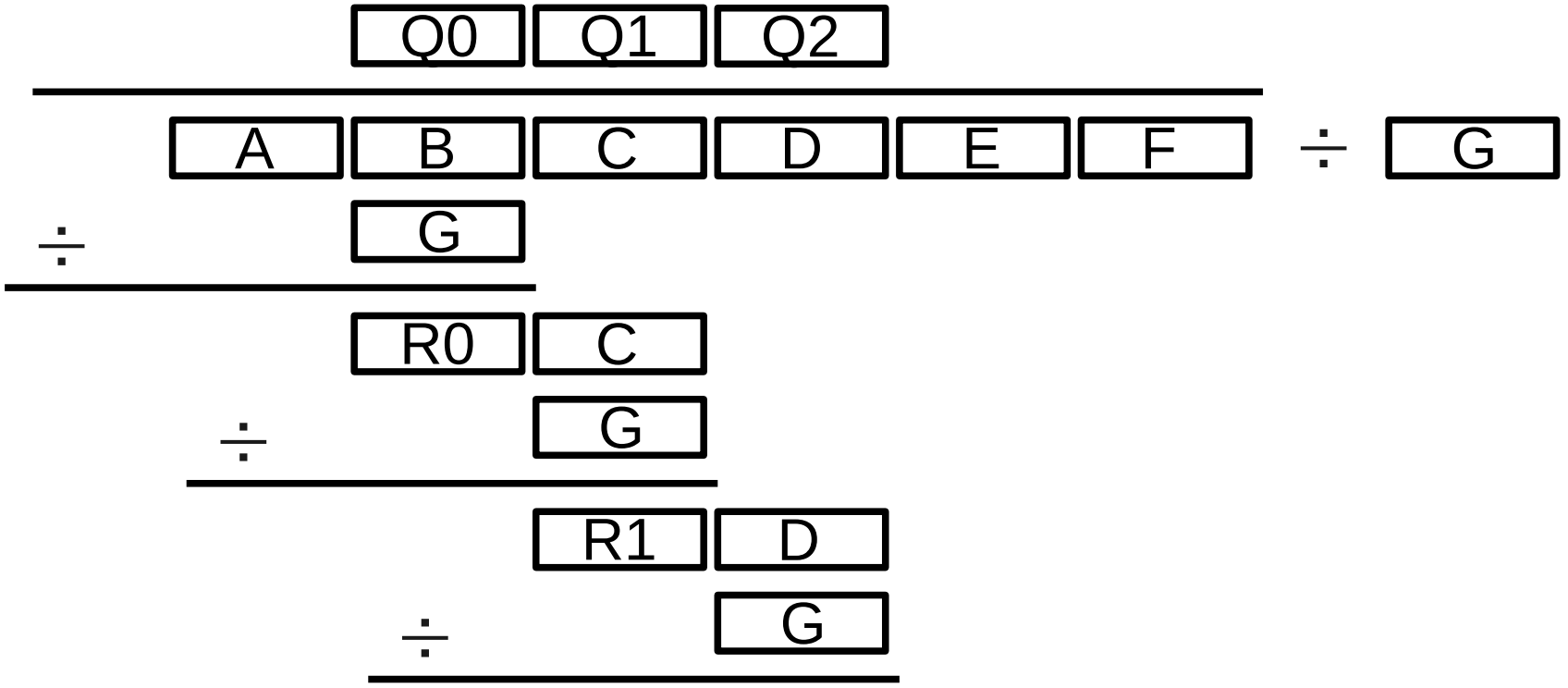
[i]div arg ; (**eax**):=(**edx:eax**)/**arg** – iloraz, ; (**edx**):=(**edx:eax**) **mod arg** – reszta



dzielenie krótkie: – konieczne rozszerzenie dzielnej

```
movl $0, %edx     ; (edx):= 0 (rozszerzenie zerowe)
div  %ecx         ; (eax):=(edx:eax) /(ecx)

cwde              ; rozszerzenie znakowe eax na edx
idiv %ecx         ; (eax):=(edx:eax) /(ecx)
```

Dzielenie „rozszerzone”



$$Q0 = AB \div G, \quad R0 = AB \mod G$$

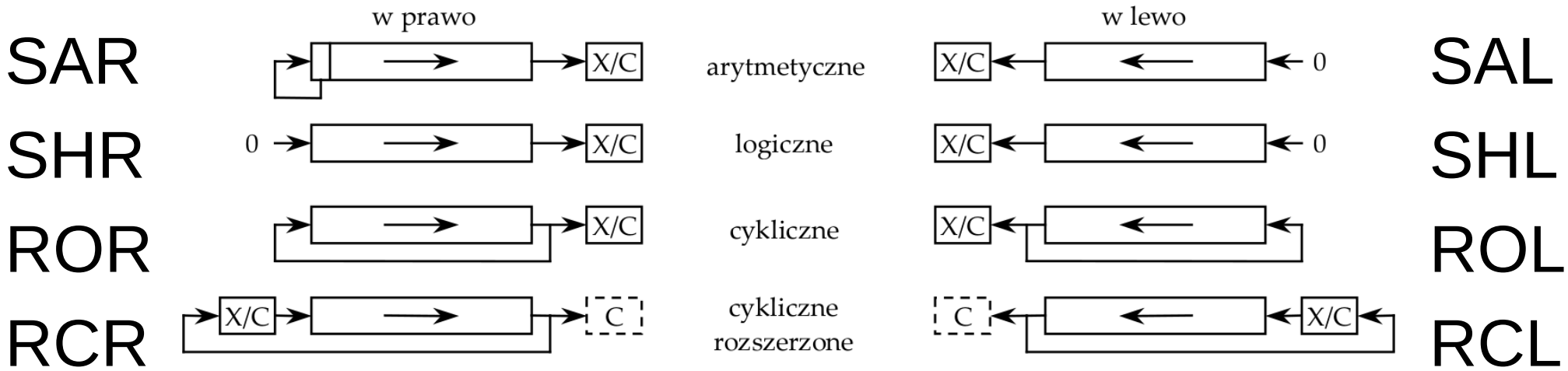
$$Q1 = R0\,C \div G, \quad R1 = R0\,C \mod G$$

Instrukcje pomocnicze

cpuid (ang. ***cpu id****entification*) – identyfikacja procesora i specyfikacja wersji
rdtsc (ang. ***read t****ime* ***s****tamp* ***c****ounter*) – odczyt licznika cykli procesora
movsx (ang. ***mov****e* ***s****ign* ***ex****tended*) – kopiowanie z roszerzeniem znakowym (U2)
movzx (ang. ***mov****e* ***z****ero* ***ex****tended*) – kopiowanie z roszerzeniem zerowym (NB)
stc (ang. ***st*** ***c****arry*) – ustawienie CF=1
clc (ang. ***cl****ear* ***c****arry*) – ustawienie CF=0
bswap (ang. ***b****yte* ***swap***) – odwrócenie kolejności bajtów w słowie
cwde / cdq – rozszerzenie znakowe akumulatora **a** (eax na edx:eax/rax na rdx:rax)

rotacje (przesunięcia cykliczne) i przesunięcia
rol (ang. *rotate left through* ***c****arry*) – przesunięcie cykliczne bitów w lewo
ror (ang. *rotate right through* ***c****arry*) – przesunięcie cykliczne bitów w prawo
rlc (ang. *rotate left through* ***c****arry*) – przesunięcie bitów w lewo z dołączonym CF
rrc (ang. *rotate right through* ***c****arry*) – przesunięcie bitów w prawo z dołączonym CF
shl (ang. *rotate left through* ***c****arry*) –przesunięcie bitów w lewo
shr (ang. *rotate left through* ***c****arry*) –przesunięcie bitów w prawo
sar (ang. *rotate left through* ***c****arry*) –przesunięcie bitów w prawo z powieleniem

Przesunięcia i rotacje



Schematy przesunięć i rotacji (przesunięć cyklicznych)

DP4A

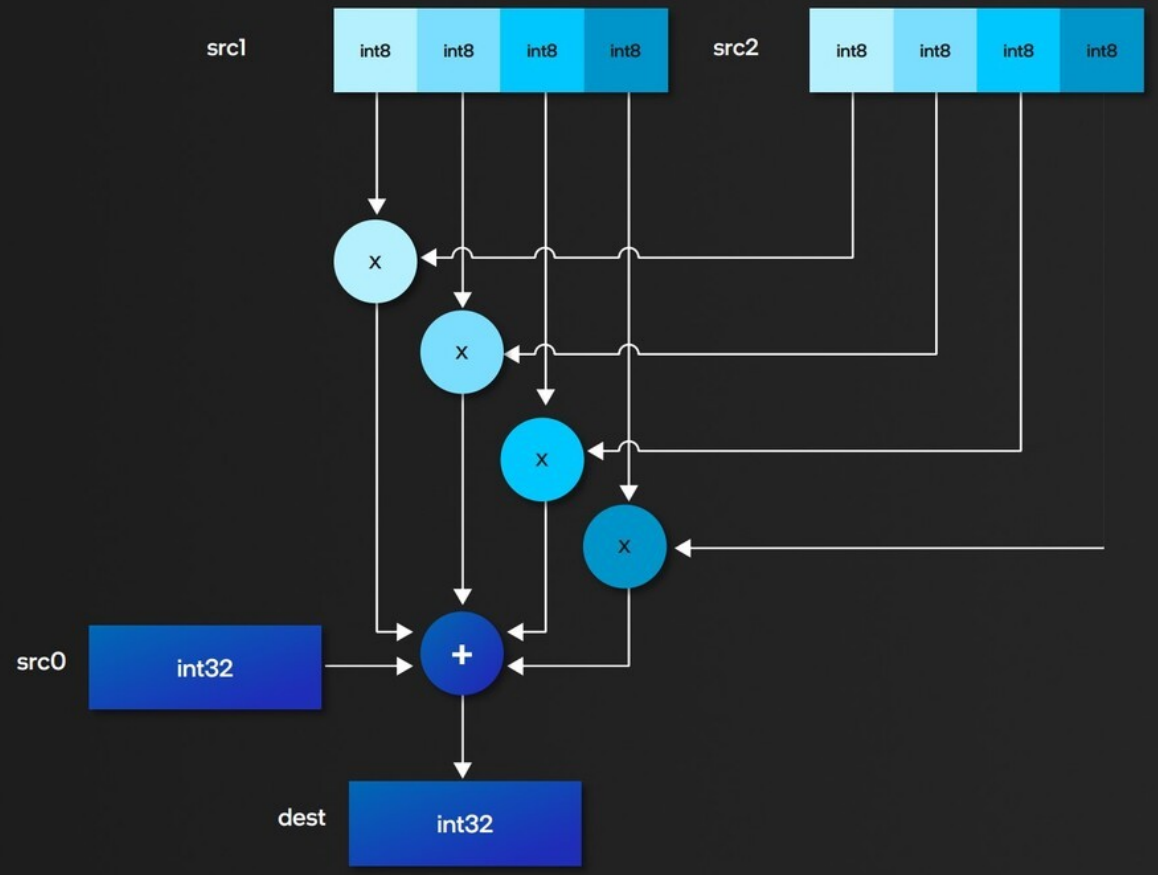
AI Acceleration

DP4a instruction support

- First introduced with Xᵉ LP (TGL)
- **64 INT8** ops/clock
- **32-bit** accumulation

New with Xᵉ LPG

- Larger engine, higher clocks
- Co-issue with FP pipe



6.2. STACK

The stack (see Figure 6-1) is a contiguous array of memory locations. It is contained in a segment and identified by the segment selector in the SS register. (When using the flat memory model, the stack can be located anywhere in the linear address space for the program.) A stack can be up to 4 gigabytes long, the maximum size of a segment.

Items are placed on the stack using the PUSH instruction and removed from the stack using the POP instruction. When an item is pushed onto the stack, the processor decrements the ESP register, then writes the item at the new top of stack. When an item is popped off the stack, the processor reads the item from the top of stack, then increments the ESP register. In this manner, the stack grows **down** in memory (towards lesser addresses) when items are pushed on the stack and shrinks **up** (towards greater addresses) when the items are popped from the stack.

A program or operating system/executive can set up many stacks. For example, in multitasking systems, each task can be given its own stack. The number of stacks in a system is limited by the maximum number of segments and the available physical memory.

When a system sets up many stacks, only one stack—the **current stack**—is available at a time. The current stack is the one contained in the segment referenced by the SS register.

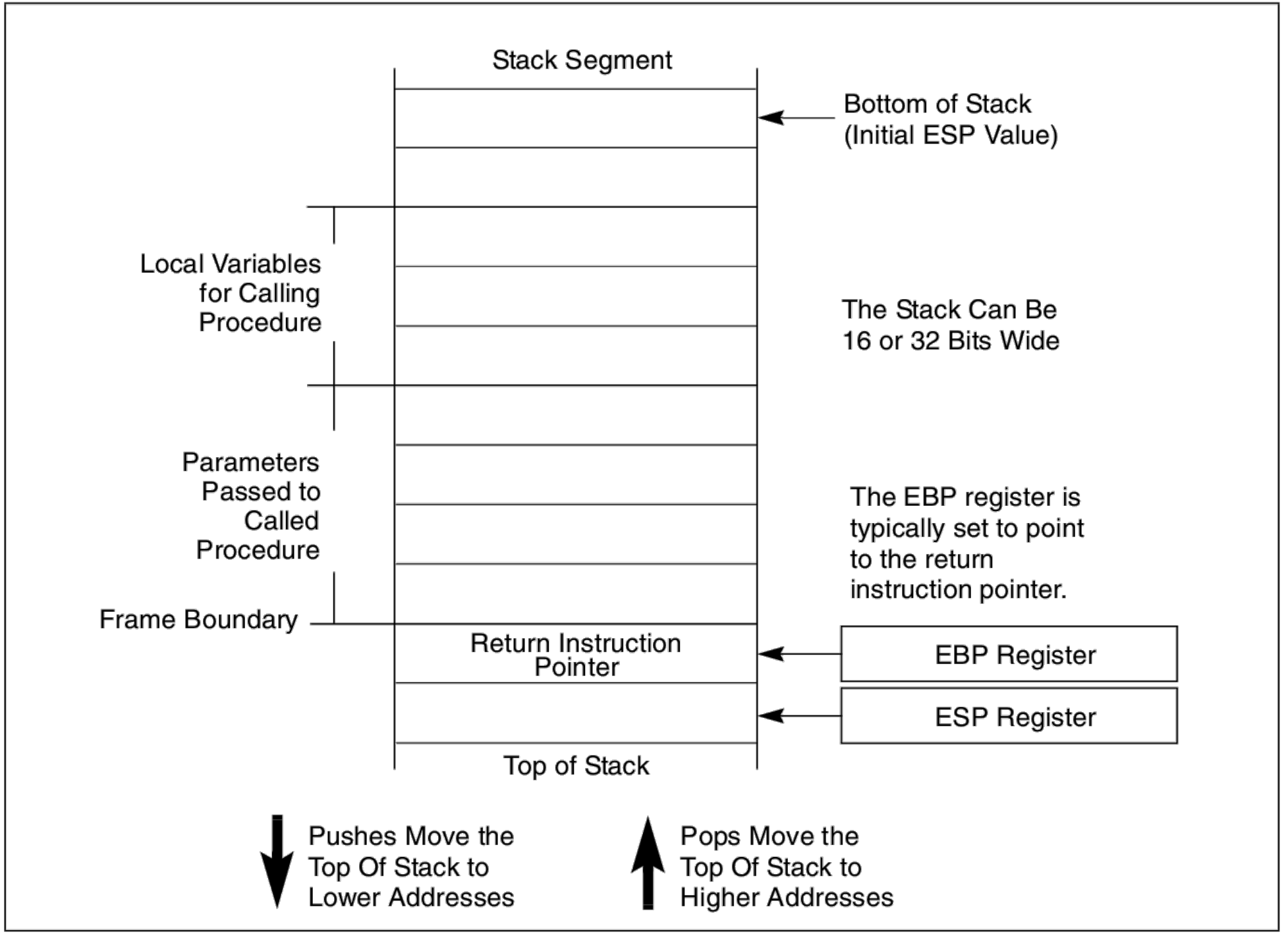


Figure 6-1. Stack Structure

PUSH—Push Word, Doubleword, or Quadword Onto the Stack

Opcode[1]	Instruction	Op/ En	64-Bit Mode	Compat/ Leg Mode	Description
FF /6	PUSH r/m16	M	Valid	Valid	Push r/m16.
FF /6	PUSH r/m32	M	N.E.	Valid	Push r/m32.
FF /6	PUSH r/m64	M	Valid	N.E.	Push r/m64.
50+rw	PUSH r16	O	Valid	Valid	Push r16.
50+rd	PUSH r32	O	N.E.	Valid	Push r32.
50+rd	PUSH r64	O	Valid	N.E.	Push r64.
6A ib	PUSH imm8	I	Valid	Valid	Push imm8.
68 iw	PUSH imm16	I	Valid	Valid	Push imm16.
68 id	PUSH imm32	I	Valid	Valid	Push imm32.
0E	PUSH CS	ZO	Invalid	Valid	Push CS.
16	PUSH SS	ZO	Invalid	Valid	Push SS.
1E	PUSH DS	ZO	Invalid	Valid	Push DS.
06	PUSH ES	ZO	Invalid	Valid	Push ES.
0F A0	PUSH FS	ZO	Valid	Valid	Push FS.
0F A8	PUSH GS	ZO	Valid	Valid	Push GS.

Description

Decrements the stack pointer and then stores the source operand on the top of the stack. Address and operand sizes are determined and used as follows:

- Address size. The D flag in the current code-segment descriptor determines the default address size; it may be overridden by an instruction prefix (67H).

  The address size is used only when referencing a source operand in memory.

- Operand size. The D flag in the current code-segment descriptor determines the default operand size; it may be overridden by instruction prefixes (66H or REX.W).

  The operand size (16, 32, or 64 bits) determines the amount by which the stack pointer is decremented (2, 4 or 8).

  If the source operand is an immediate of size less than the operand size, a sign-extended value is pushed on the stack. If the source operand is a segment register (16 bits) and the operand size is 64-bits, a zero-extended value is pushed on the stack; if the operand size is 32-bits, either a zero-extended value is pushed on the stack or the segment selector is written on the stack using a 16-bit move. For the last case, all recent Intel Core and Intel Atom processors perform a 16-bit move, leaving the upper portion of the stack location unmodified.

POP—Pop a Value From the Stack

Opcode	Instruction	Op/ En	64-Bit Mode	Compat/ Leg Mode	Description
8F /0	POP r/m16	M	Valid	Valid	Pop top of stack into m16; increment stack pointer.
8F /0	POP r/m32	M	N.E.	Valid	Pop top of stack into m32; increment stack pointer.
8F /0	POP r/m64	M	Valid	N.E.	Pop top of stack into m64; increment stack pointer. Cannot encode 32-bit operand size.
58+ rw	POP r16	O	Valid	Valid	Pop top of stack into r16; increment stack pointer.
58+ rd	POP r32	O	N.E.	Valid	Pop top of stack into r32; increment stack pointer.
58+ rd	POP r64	O	Valid	N.E.	Pop top of stack into r64; increment stack pointer. Cannot encode 32-bit operand size.
1F	POP DS	ZO	Invalid	Valid	Pop top of stack into DS; increment stack pointer.
07	POP ES	ZO	Invalid	Valid	Pop top of stack into ES; increment stack pointer.
17	POP SS	ZO	Invalid	Valid	Pop top of stack into SS; increment stack pointer.
0F A1	POP FS	ZO	Valid	Valid	Pop top of stack into FS; increment stack pointer by 16 bits.
0F A1	POP FS	ZO	N.E.	Valid	Pop top of stack into FS; increment stack pointer by 32 bits.
0F A1	POP FS	ZO	Valid	N.E.	Pop top of stack into FS; increment stack pointer by 64 bits.
0F A9	POP GS	ZO	Valid	Valid	Pop top of stack into GS; increment stack pointer by 16 bits.
0F A9	POP GS	ZO	N.E.	Valid	Pop top of stack into GS; increment stack pointer by 32 bits.
0F A9	POP GS	ZO	Valid	N.E.	Pop top of stack into GS; increment stack pointer by 64 bits.

Description

Loads the value from the top of the stack to the location specified with the destination operand (or explicit opcode) and then increments the stack pointer. The destination operand can be a general-purpose register, memory location, or segment register.

Address and operand sizes are determined and used as follows:

- Address size. The D flag in the current code-segment descriptor determines the default address size; it may be overridden by an instruction prefix (67H).

  The address size is used only when writing to a destination operand in memory.

- Operand size. The D flag in the current code-segment descriptor determines the default operand size; it may be overridden by instruction prefixes (66H or REX.W).

  The operand size (16, 32, or 64 bits) determines the amount by which the stack pointer is incremented (2, 4 or 8).

- Stack-address size. Outside of 64-bit mode, the B flag in the current stack-segment descriptor determines the size of the stack pointer (16 or 32 bits); in 64-bit mode, the size of the stack pointer is always 64 bits.

  The stack-address size determines the width of the stack pointer when reading from the stack in memory and when incrementing the stack pointer. (As stated above, the amount by which the stack pointer is incremented is determined by the operand size.)

Instrukcje mało użyteczne

Arytmetyka dziesiętna

aaa (ang. *ascii adjust after addition*) –
aad (ang. *adjust before division*) –
aam (ang. *adjust after multiplication*) –
aas (ang. *ascii adjust after subtraction*) –
daa (ang. *decimal adjust for addition*) –
das (ang. *decimal adjust for subtraction*) –

xlat (ang. …) – tablicowa translacja kodu
neg (ang. *negate*) – wytworzenie liczby przeciwnej
cmc (ang. *complement carry*) –
cmpxchg (ang. *compare and exchange*) –
xadd (ang. *exchange and add*) –
lods (ang. *load string*) –
stos (ang. *store string*) –
movs (ang. *move string*) –

Rozgałęzienia

(skoki, funkcje)

Przepływ sterowania

powiązanie instrukcji tworzących program

- *funkcjonalne* – jaka jest następna instrukcja

  konstrukcje: **repeat** (powtarzaj), **execute** (wykonaj).

- *lokacyjne* – gdzie jest następna instrukcja
  - *sekwencyjne* (ang. *sequential*) – wyklucza sterowanie,
    - ❖ domniemana lokacja kolejnego rozkazu (porządek liniowy)

      → krótszy kod
  - *łańcuchowe* (ang. *chained*) – umożliwia sterowanie,
    - ❖ konieczne wskazanie lokacji kolejnego rozkazu

      → dłuższy kod (musi zawierać wskazanie kolejnego rozkazu)

Kompromis:

- domniemany liniowy porządek instrukcji → *powiązanie sekwencyjne*
- sterowanie (zmiana porządku instrukcji) → *powiązanie łańcuchowe*

Sterowanie

rozgałęzienie (skok warunkowy) – wybór alternatywnej ścieżki przetwarzania

j*war dest_adr*

– implementacja podstawowej instrukcji warunkowej (*war* = *warunek*=TRUE):

if *warunek*=TRUE **then goto** *dest_adr* (**else** *continue*)

skok ze śladem (**call**) – wywołanie funkcji (procedury)

funkcja:

- przekazywanie argumentów
- kontekst (blok aktywacji)
- kapsułkowanie
- zakończenie i zwrot wyniku

struktury danych funkcji powinny być tworzone *dynamicznie*

rozwiązanie: użycie stosu programowego

(IA-32/IA-32e) rozgałęzienia i tworzenie pętli

jmp (ang. *jump*) – skocz (przekaż sterowanie)
j*cc* (ang. *jump conditionally*) – przekaż sterowanie jeśli warunek *cc* prawdziwy

loop (ang. *loop*) – zapętlaj (przekaż sterowanie) dopóki licznik ≠ 0
loopz – zapętlaj (przekaż sterowanie) dopóki licznik ≠ 0 lub ZF = 0
loopnz

jcxz/jecxz/jrcxz – zabezpieczenie przed wykonaniem pętli przy zerowym stanie początkowym licznika

JMP—Jump

Opcode	Instruction	Description
EB *cb*	JMP *rel8*	Jump short, relative, displacement relative to next instruction
E9 *cw*	JMP *rel16*	Jump near, relative, displacement relative to next instruction
E9 *cd*	JMP *rel32*	Jump near, relative, displacement relative to next instruction
FF /4	JMP *r/m16*	Jump near, absolute indirect, address given in *r/m16*
FF /4	JMP *r/m32*	Jump near, absolute indirect, address given in *r/m32*
EA *cd*	JMP *ptr16:16*	Jump far, absolute, address given in operand
EA *cp*	JMP *ptr16:32*	Jump far, absolute, address given in operand
FF /5	JMP *m16:16*	Jump far, absolute indirect, address given in *m16:16*
FF /5	JMP *m16:32*	Jump far, absolute indirect, address given in *m16:32*

Description

Transfers program control to a different point in the instruction stream without recording return information. The destination (target) operand specifies the address of the instruction being jumped to. This operand can be an immediate value, a general-purpose register, or a memory location.

This instruction can be used to execute four different types of jumps:

- Near jump—A jump to an instruction within the current code segment (the segment currently pointed to by the CS register), sometimes referred to as an intrasegment jump.
- Short jump—A near jump where the jump range is limited to –128 to +127 from the current EIP value.
- Far jump—A jump to an instruction located in a different segment than the current code segment but at the same privilege level, sometimes referred to as an intersegment jump.
- Task switch—A jump to an instruction located in a different task.

JMP

Near and Short Jumps.

- When executing a near jump, the processor jumps to the address (within the current code segment) that is specified with the target operand. The target operand specifies either an absolute offset (that is an offset from the base of the code segment) or a relative offset (a signed displacement relative to the current value of the instruction pointer in the EIP register). A near jump to a relative offset of 8-bits (rel8) is referred to as a short jump. The CS register is not changed on near and short jumps.
- An absolute offset is specified indirectly in a general-purpose register or a memory location (r/m16 or r/m32). The operand-size attribute determines the size of the target operand (16 or 32 bits). Absolute offsets are loaded directly into the EIP register. If the operand-size attribute is 16, the upper two bytes of the EIP register are cleared, resulting in a maximum instruction pointer size of 16 bits.
- A relative offset (rel8, rel16, or rel32) is generally specified as a label in assembly code, but at the machine code level, it is encoded as a signed 8-, 16-, or 32-bit immediate value. This value is added to the value in the EIP register. (Here, the EIP register contains the address of the instruction following the JMP instruction). When using relative offsets, the opcode (for short vs. near jumps) and the operand-size attribute (for near relative jumps) determines the size of the target operand (8, 16, or 32 bits).

Figure 3-45: Branch Instruction, All Models

C

```
label:
        . . .
        goto label;
```

Assembly

```
.L01:
        . . .
        jmp     .L01
```

Figure 3-46: Absolute `switch` Code

C

```
switch (j)
{
case 0:
     . . .
case 2:
     . . .
case 3:
     . . .
default:
     . . .
}
```

Assembly

```
        cmpl    $3, %eax
        ja      .Ldef
        jmp     *.Ltab(,%eax,4)
.Ltab:  .long   .Lcase0
        .long   .Ldef
        .long   .Lcase2
        .long   .Lcase3
```

Skoki warunkowe

- decyzja o skoku na podstawie flag
- zazwyczaj konieczne dwa rozkazy
  - ustawienie wartości flag: CMP lub TEST
  - wykonanie skoku warunkowego: Jcc
- nazwy poprawne jedynie dla instrukcji CMP
- różne instrukcje dla porównań w NB (JA/JB) i U2 (JG/JL)

J*cc*—Jump if Condition Is Met

Opcode	Instruction	Description
77 *cb*	JA *rel8*	Jump short if above (CF=0 and ZF=0)
73 *cb*	JAE *rel8*	Jump short if above or equal (CF=0)
72 *cb*	JB *rel8*	Jump short if below (CF=1)
76 *cb*	JBE *rel8*	Jump short if below or equal (CF=1 or ZF=1)
72 *cb*	JC *rel8*	Jump short if carry (CF=1)
E3 *cb*	JCXZ *rel8*	Jump short if CX register is 0
E3 *cb*	JECXZ *rel8*	Jump short if ECX register is 0
74 *cb*	JE *rel8*	Jump short if equal (ZF=1)
7F *cb*	JG *rel8*	Jump short if greater (ZF=0 and SF=OF)
7D *cb*	JGE *rel8*	Jump short if greater or equal (SF=OF)
7C *cb*	JL *rel8*	Jump short if less (SF<>OF)
7E *cb*	JLE *rel8*	Jump short if less or equal (ZF=1 or SF<>OF)
76 *cb*	JNA *rel8*	Jump short if not above (CF=1 or ZF=1)
72 *cb*	JNAE *rel8*	Jump short if not above or equal (CF=1)
73 *cb*	JNB *rel8*	Jump short if not below (CF=0)
77 *cb*	JNBE *rel8*	Jump short if not below or equal (CF=0 and ZF=0)
73 *cb*	JNC *rel8*	Jump short if not carry (CF=0)
75 *cb*	JNE *rel8*	Jump short if not equal (ZF=0)
7E *cb*	JNG *rel8*	Jump short if not greater (ZF=1 or SF<>OF)
7C *cb*	JNGE *rel8*	Jump short if not greater or equal (SF<>OF)
7D *cb*	JNL *rel8*	Jump short if not less (SF=OF)
7F *cb*	JNLE *rel8*	Jump short if not less or equal (ZF=0 and SF=OF)
71 *cb*	JNO *rel8*	Jump short if not overflow (OF=0)
7B *cb*	JNP *rel8*	Jump short if not parity (PF=0)
79 *cb*	JNS *rel8*	Jump short if not sign (SF=0)
75 *cb*	JNZ *rel8*	Jump short if not zero (ZF=0)
70 *cb*	JO *rel8*	Jump short if overflow (OF=1)
7A *cb*	JP *rel8*	Jump short if parity (PF=1)
7A *cb*	JPE *rel8*	Jump short if parity even (PF=1)
7B *cb*	JPO *rel8*	Jump short if parity odd (PF=0)
78 *cb*	JS *rel8*	Jump short if sign (SF=1)
74 *cb*	JZ *rel8*	Jump short if zero (ZF = 1)
0F 87 *cw/cd*	JA *rel16/32*	Jump near if above (CF=0 and ZF=0)
0F 83 *cw/cd*	JAE *rel16/32*	Jump near if above or equal (CF=0)
0F 82 *cw/cd*	JB *rel16/32*	Jump near if below (CF=1)
0F 86 *cw/cd*	JBE *rel16/32*	Jump near if below or equal (CF=1 or ZF=1)
0F 82 *cw/cd*	JC *rel16/32*	Jump near if carry (CF=1)
0F 84 *cw/cd*	JE *rel16/32*	Jump near if equal (ZF=1)

CMP—Compare Two Operands

Opcode	Instruction	Description
3C *ib*	CMP AL, *imm8*	Compare *imm8* with AL
3D *iw*	CMP AX, *imm16*	Compare *imm16* with AX
3D *id*	CMP EAX, *imm32*	Compare *imm32* with EAX
80 /7 *ib*	CMP *r/m8, imm8*	Compare *imm8* with *r/m8*
81 /7 *iw*	CMP *r/m16, imm16*	Compare *imm16* with *r/m16*
81 /7 *id*	CMP *r/m32,imm32*	Compare *imm32* with *r/m32*
83 /7 *ib*	CMP *r/m16,imm8*	Compare *imm8* with *r/m16*
83 /7 *ib*	CMP *r/m32,imm8*	Compare *imm8* with *r/m32*
38 /*r*	CMP *r/m8,r8*	Compare *r8* with *r/m8*
39 /*r*	CMP *r/m16,r16*	Compare *r16* with *r/m16*
39 /*r*	CMP *r/m32,r32*	Compare *r32* with *r/m32*
3A /*r*	CMP *r8,r/m8*	Compare *r/m8* with *r8*
3B /*r*	CMP *r16,r/m16*	Compare *r/m16* with *r16*
3B /*r*	CMP *r32,r/m32*	Compare *r/m32* with *r32*

Description

Compares the first source operand with the second source operand and sets the status flags in the EFLAGS register according to the results. The comparison is performed by subtracting the second operand from the first operand and then setting the status flags in the same manner as the SUB instruction. When an immediate value is used as an operand, it is sign-extended to the length of the first operand.

The CMP instruction is typically used in conjunction with a conditional jump (J*cc*), condition move (CMOV*cc*), or SET*cc* instruction. The condition codes used by the J*cc*, CMOV*cc*, and SET*cc* instructions are based on the results of a CMP instruction. Appendix B, *EFLAGS Condition Codes*, in the *IA-32 Intel Architecture Software Developer's Manual, Volume 1*, shows the relationship of the status flags and the condition codes.

TEST—Logical Compare

Opcode	Instruction	Description
A8 *ib*	TEST AL,*imm8*	AND *imm8* with AL; set SF, ZF, PF according to result
A9 *iw*	TEST AX,*imm16*	AND *imm16* with AX; set SF, ZF, PF according to result
A9 *id*	TEST EAX,*imm32*	AND *imm32* with EAX; set SF, ZF, PF according to result
F6 /0 *ib*	TEST *r/m8,imm8*	AND *imm8* with *r/m8*; set SF, ZF, PF according to result
F7 /0 *iw*	TEST *r/m16,imm16*	AND *imm16* with *r/m16*; set SF, ZF, PF according to result
F7 /0 *id*	TEST *r/m32,imm32*	AND *imm32* with *r/m32*; set SF, ZF, PF according to result
84 /*r*	TEST *r/m8,r8*	AND *r8* with *r/m8*; set SF, ZF, PF according to result
85 /*r*	TEST *r/m16,r16*	AND *r16* with *r/m16*; set SF, ZF, PF according to result
85 /*r*	TEST *r/m32,r32*	AND *r32* with *r/m32*; set SF, ZF, PF according to result

Description

Computes the bit-wise logical AND of first operand (source 1 operand) and the second operand (source 2 operand) and sets the SF, ZF, and PF status flags according to the result. The result is then discarded.

LOOP/LOOP*cc*—Loop According to ECX Counter

Opcode	Instruction	Description
E2 *cb*	LOOP *rel8*	Decrement count; jump short if count ≠ 0
E1 *cb*	LOOPE *rel8*	Decrement count; jump short if count ≠ 0 and ZF=1
E1 *cb*	LOOPZ *rel8*	Decrement count; jump short if count ≠ 0 and ZF=1
E0 *cb*	LOOPNE *rel8*	Decrement count; jump short if count ≠ 0 and ZF=0
E0 *cb*	LOOPNZ *rel8*	Decrement count; jump short if count ≠ 0 and ZF=0

Description

Performs a loop operation using the ECX or CX register as a counter. Each time the LOOP instruction is executed, the count register is decremented, then checked for 0. If the count is 0, the loop is terminated and program execution continues with the instruction following the LOOP instruction. If the count is not zero, a near jump is performed to the destination (target) operand, which is presumably the instruction at the beginning of the loop. If the address-size attribute is 32 bits, the ECX register is used as the count register; otherwise the CX register is used.

The target instruction is specified with a relative offset (a signed offset relative to the current value of the instruction pointer in the EIP register). This offset is generally specified as a label in assembly code, but at the machine code level, it is encoded as a signed, 8-bit immediate value, which is added to the instruction pointer. Offsets of –128 to +127 are allowed with this instruction.

Some forms of the loop instruction (LOOP*cc*) also accept the ZF flag as a condition for terminating the loop before the count reaches zero. With these forms of the instruction, a condition code (*cc*) is associated with each instruction to indicate the condition being tested for. Here, the

Konstrukcje sterujące

*// **if** warunek **then** operacja*

cmp *{warunek}*

j*{warunek **przeciwny**}* koniec

{operacja}

koniec:

*// **if** warunek **then***
// operacja1
*// **else***
// operacja2

cmp *{warunek}*

j*{warunek **przeciwny**}* operacja2

{operacja1}

jmp koniec

operacja2:

{operacja2}

koniec:

Wzorce sterowania – rozgałęzienie (1)

– akcja warunkowa: **if** *warunek*=TRUE **then** *polecenie_T* **else** *polecenie_F*

przykład: jeśli (ebx)≤(eax) zmniejsz eax, w przeciwnym razie zwiększ eax

	Intel (MASM)		**AT&T/Linux/UNIX**	Komentarz
	cmp arg1, arg2		**cmpl** %eax, %ebx	# (ebx) ≤ (eax) → ZF=1 ∨ CF=1 (**be=T**)
	jwar alt		**jbe** alt	#
	polecenie F		**incl** %eax	# (ebx) > (eax), **be=F** (polecenie_F)
	jmp cont		**jmp** cont	
alt:	*polecenie* T	alt:	**decl** %eax	# (ebx) ≤ (eax), **be=T** (polecenie_T)
cont:		cont:		

– ominięcie: **if** *warunek*=TRUE **then** *polecenie T*

przykład: jeśli (ebx)≤5 wpisz wartość „wart" do ebx

	Intel (MASM)		**AT&T/Linux/UNIX**	Komentarz
	cmp arg1, arg2		**cmpl** $5, %ebx	# (ebx) ≤ 5 → ZF=1 ∨ CF=1 (**gt=F**)
	jnot_war alt		**jgt** alt	
	polecenie T		**movl** $, %ebx	# (ebx) ≤ 5, **ngt=T** (polecenie_T)
alt:		alt:		# **ngt=F**

Wzorce sterowania – rozgałęzienie (2)

– zapętlenie: **for** *i*:= *st* **step** –1 **until** *end* **do** *polecenie* (*i*)

przykład: jeśli (ebx)≤5 wpisz wartość „wart" do ebx

	Intel (MASM)		AT&T/Linux/UNIX	Komentarz
	mov ecx, *end*		**movl** $*end*, %ecx	
	sub ecx, *st*–1		**subl**	
powt:	*polecenie* (*i*)	powt:	**movl** zm(,%esi,4), %ecx	
			xorl $mask, %ecx	
			movl %ecx, zm(,%esi,4)	
	i:=*i*-1		**decl** %esi	
	loop powt		**loop** powt	

– zapętlenie: **for** *i*:= *st* **step** –1 **until** *end* **do** *polecenie* (*i*)

	Intel (MASM)		AT&T/Linux/UNIX	Komentarz
	mov ecx, *end*		**movl** $*end*, %ecx	
	sub ecx, *st*–1		**subl**	
powt:	*polecenie* (*i*)	powt:	**movl** zm(,%esi,4), %ecx	
			xorl $mask, %ecx	
			movl %ecx, zm(,%esi,4)	
	i:=*i*-1		**decl** %esi	
	loop powt		**loop** powt	

Wzorce sterowania – rozgałęzienie (3)

– powtarzaj dopóki: **repeat** *polecenie* **until** *warunek*=TRUE

	Intel (MASM)		AT&T/Linux/UNIX	Komentarz
start:	*polecenie* (*A*, *B*)	start:	*polecenie* (*A*, *B*)	
	mov eax, A		**movl** $A, %eax	
	cmp eax, B		**cmpl** $B, %eax	
	jgt start		**jgt** start	

– wykonaj jeśli: **while** *warunek*=TRUE **do** *polecenie*

	Intel (MASM)		AT&T/Linux/UNIX	Komentarz
start:	**mov** eax, *B*	start:	**movl** *B*, %eax	
	cmp eax, *A*		**cmp** *A*, %eax	
	jle skip		**jle** skip	
	polecenie (*A*, *B*)		*polecenie* (*A*, *B*)	
	jmp start		**jmp** start	
skip:		skip:		

Indeksowanie zmiennej i organizacja pętli (1)

- działanie podstawowe algorytmu

(etykieta)	Rozkaz	Komentarz
	[movl $wsk, %esi]	# wybrana wartość wskaźnika (do testu)
	movb buf(,%esi,1), %al	# argument1 (bajt) z pamięci do rejestru
	movb tab(,%esi,1), %bl	# argument2 (bajt) z pamięci do rejestru
	...	# przetwarzanie
	...	# wynik w rejestrze ah
	movb %ah, wyn(,%esi,1)	# wynik (1 bajt) do pamięci

- indeksacja (wspólny indeks)

(etykieta)	Rozkaz	Komentarz
	[inc %esi]	# (aktualizacja wskaźnika) - preindeksacja
	movb buf(,%esi,1), %al	# argument1 (bajt) z pamięci do rejestru
	movb tab(,%esi,1), %bl	# argument2 (bajt) z pamięci do rejestru
	...	# wynik w rejestrze ah
	movb %ah, wyn(,%esi,1)	# wynik (1 bajt) do pamięci
	[inc %esi]	# (aktualizacja wskaźnika) – postindeksacja

Indeksowanie zmiennej i organizacja pętli (2)

- utworzenie pętli

(etykieta)	Rozkaz	Komentarz
		#
	movs $init, %esi	# inicjalizacja wskaźnika
pocz:	...	# stałe parametry jednego przebiegu
	[incl/decl %esi]	# (aktualizacja wskaźnika) - preindeksacja
	movb buf(,%esi,1), %al	# argument1 (bajt) z pamięci do rejestru
	movb tab(,%esi,1), %bl	# argument2 (bajt) z pamięci do rejestru
	...	# wynik w rejestrze ah
	movb %ah, wyn(,%esi,1)	# wynik (1 bajt) do pamięci
	[incl/decl %esi]	# (aktualizacja wskaźnika) – postindeksacja
	...	# stałe parametry jednego przebiegu
	cmpl $zakres, %esi	# (esi)—zakres → F (warunek *cc*)
	jcc pocz	

Indeksowanie zmiennej i organizacja pętli (3)

- dowolna ustalona aktualizacja wskaźnika elementu zmiennej indeksowanej

(etykieta)	Rozkaz	Komentarz
	`movs $rozmiar, %ebx`	
	`movs $init, %esi`	
`pocz:`	`...`	# stałe parametry jednego przebiegu
	`[lea (%esi, %ebx, 1), %esi]`	# preindeksacja: esi := (esi)+(ebx)
	`...`	
	`...`	# treść algorytmu – pojedyncze wykonanie
	`...`	
	`[lea (%esi, %ebx, 1), %esi]`	# postindeksacja
	`...`	# stałe parametry jednego przebiegu
	`[cmpl $zakres, %esi]`	# (esi)—zakres → F (warunek *cc*)
	`jcc pocz`	

Przykład: algorytm Euklidesa (NWP(a,b)=NWP(b,a mod b))

(etykieta)	Rozkaz	Komentarz
gcd:	push %ebp	# GCD(a,b)=GCD(b, a mod b)
	movl %esp, %ebp	
	movl 8(%ebp), %eax	# argument „a" w rejestrze eax
	movl 12(%ebp), %ebx	# argument „b" w rejestrze ebx
pocz:	movl $0, %edx	# 0 do edx
	divl %ebx	# reszta a mod b w edx
	movl %ebx, %eax	# b w miejsce a (do eax)
	movl %edx, %ebx	# a mod b w miejsce b (do ebx)
	andl %edx, %edx	# czy reszta = 0
	jnz pocz	# powtarzaj dopóki reszta ≠ 0
	movl %eax, 8(%ebp)	# GCD(a,b) z rejestru eax na stos
end:	movl %ebp, %esp	
	pop %ebp	
	ret	

rozwiązanie alternatywne to (tylko fragment zacieniony)

pocz:	xchg %ebx, %eax	# dopóki różnica dodatnia
	subl %ebx, %eax	
	ja rem	# różnica ujemna, korekcja reszty
	addl %ebx, %eax	
	jnz pocz	# powtarzaj dopóki reszta ≠ 0

Przykład: obliczenie wartości dużej liczby (schemat Hornera)

Obliczenie wartości *N*-cyfrowej liczby dziesiętnej (tablica ASCII jej kolejnych cyfr $x_{n-1},\ldots,x_1, x_0$ w konwencji **big endian** – schemat Hornera: $X=(\ldots((x_{n-1}*10+x_{n-2})*10+x_{n-3})*10+\ldots+x_1)*10+x_0$.

```
.type asci2int @function                    #konwencja big endian!
asc2int:  push %ebp                         #
          movl %esp, %ebp                   #
          movl 8(%ebp), %ebx                # adres liczby
          movl 12(%ebp), %ecx               # rozmiar liczby
          movl $10, %edi                    # podstawa systemu liczenia
          movl $0, %eax                     # wartość początkowa sumy
pocz:     mull %edi                         # suma=suma*10 (edx wyzerowany!)
          movb (%ebx), %dl                  # wartość kolejnej cyfry
          andl $0x0f, %edx                  # ascii 2 int (w rej. 32-b)
          addl %edx, %eax                   # suma:=suma+kolejna cyfra
          incl %ebx,                        # indeks kolejnej cyfry
          loop pocz                         #
          movl %ebp, %esp                   #
          pop %ebp
          ret
```

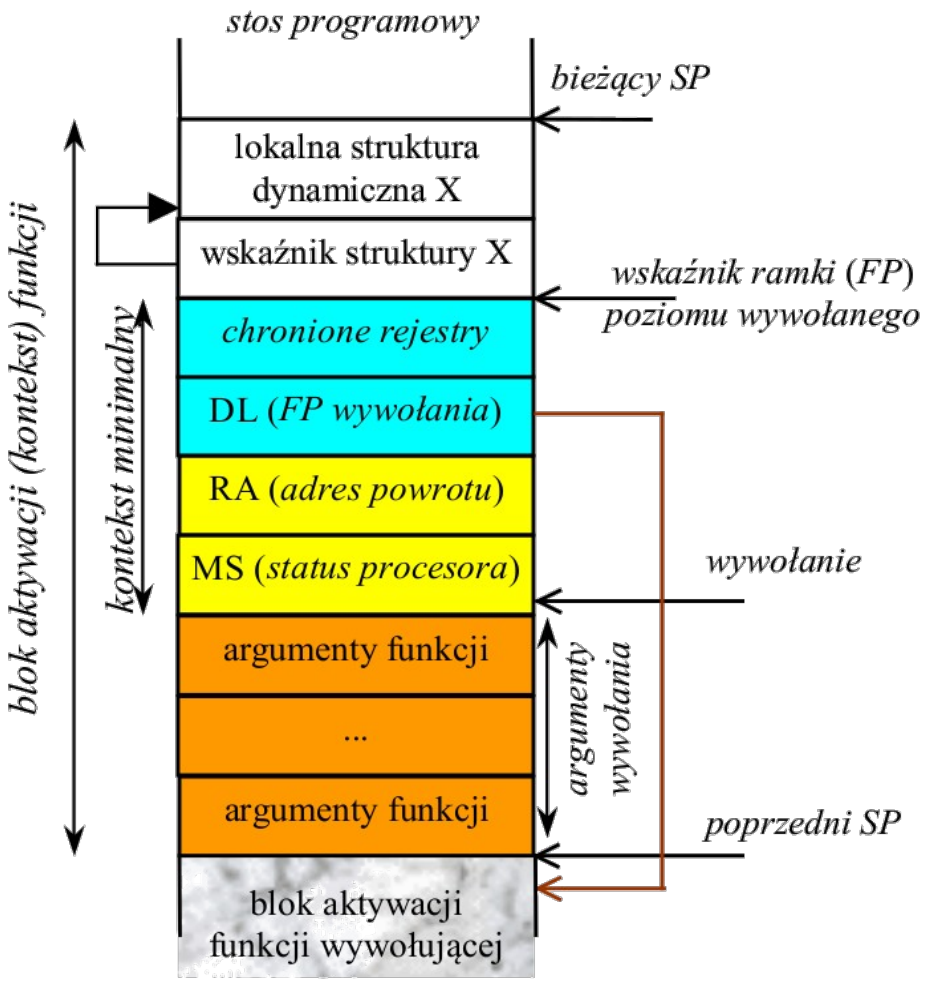
Przykład: obliczenie N-tej liczby Fibonacciego

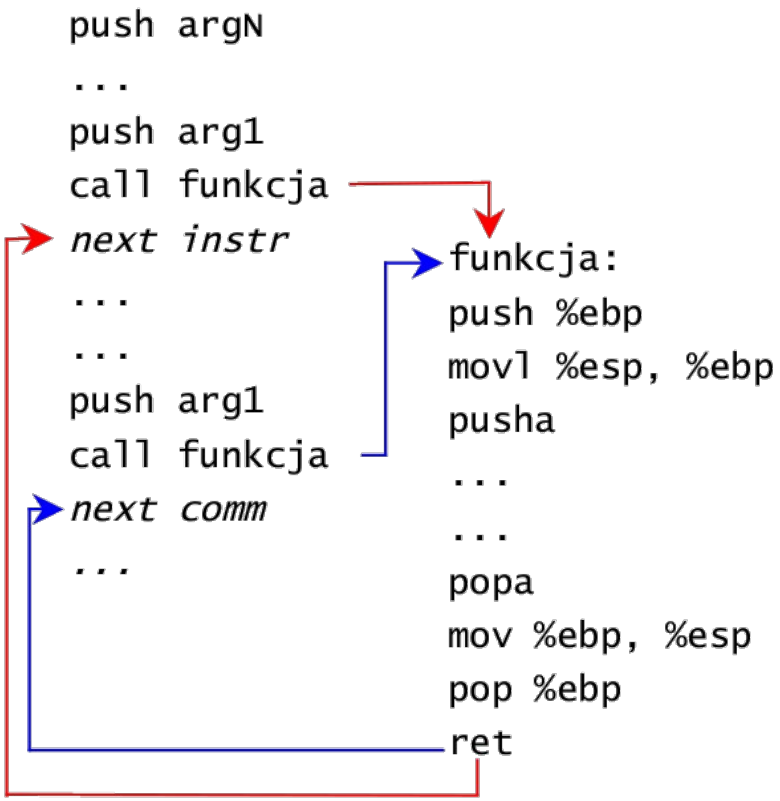
```
fibonac: push %ebp                    #
         movl %esp, %ebp              #
         movl 8(%ebp), %ecx           # numer liczby
         movl $0, %eax                # wartość początkowa F(0)=0
         movl $1, %ebx                # wartość początkowa F(1)=1
         subl $1, %ecx                # zignoruj obliczenia, jeśli N=1
         jz endf
begin:   addl %ebx, %eax              # obliczenie F(i+1)=F(i)+F(i–1)
         xchg %ebx, %eax              # przestawienie liczb F(i), F(i–1)
         loop begin                   # licznik pętli w %ecx
         movl %ebx, 8(%ebp)           # wynik na stos
endf:    movl %ebp, %esp              #
         pop %ebp
         ret
```

rozwiązanie alternatywne to (tylko fragment zacieniony)

```
         movl $0, %eax                # wartość początkowa F(0)=0
         movl $1, %ebx                # wartość początkowa F(1)=1
         subl $1, %ecx                # zignoruj obliczenia, jeśli N=1
         jz endf
begin:   xchg %ebx, %eax              # przestawienie liczb F(i), F(i–1)
         addl %eax, %ebx              # obliczenie F(i+1)=F(i)+F(i–1)
         loop begin                   # licznik pętli w %ecx
```

Funkcja – kontekst i blok aktywacji





CALL—Call Procedure

Opcode	Instruction	Description
E8 *cw*	CALL *rel16*	Call near, relative, displacement relative to next instruction
E8 *cd*	CALL *rel32*	Call near, relative, displacement relative to next instruction
FF /2	CALL *r/m16*	Call near, absolute indirect, address given in *r/m16*
FF /2	CALL *r/m32*	Call near, absolute indirect, address given in *r/m32*
9A *cd*	CALL *ptr16:16*	Call far, absolute, address given in operand
9A *cp*	CALL *ptr16:32*	Call far, absolute, address given in operand
FF /3	CALL *m16:16*	Call far, absolute indirect, address given in *m16:16*
FF /3	CALL *m16:32*	Call far, absolute indirect, address given in *m16:32*

Description

Saves procedure linking information on the stack and branches to the procedure (called procedure) specified with the destination (target) operand. The target operand specifies the address of the first instruction in the called procedure. This operand can be an immediate value, a general-purpose register, or a memory location.

This instruction can be used to execute four different types of calls:

- Near call—A call to a procedure within the current code segment (the segment currently pointed to by the CS register), sometimes referred to as an intrasegment call.
- Far call—A call to a procedure located in a different segment than the current code segment, sometimes referred to as an intersegment call.
- Inter-privilege-level far call—A far call to a procedure in a segment at a different privilege level than that of the currently executing program or procedure.
- Task switch—A call to a procedure located in a different task.

CALL

Near Call.

- When executing a near call, the processor pushes the value of the EIP register (which contains the offset of the instruction following the CALL instruction) onto the stack (for use later as a return-instruction pointer).
- The processor then branches to the address in the current code segment specified with the target operand.
- The target operand specifies either an absolute offset in the code segment (that is an offset from the base of the code segment) or a relative offset (a signed displacement relative to the current value of the instruction pointer in the EIP register, which points to the instruction following the CALL instruction). The CS register is not changed on near calls.

RET—Return from Procedure

Opcode	Instruction	Description
C3	RET	Near return to calling procedure
CB	RET	Far return to calling procedure
C2 *iw*	RET *imm16*	Near return to calling procedure and pop *imm16* bytes from stack
CA *iw*	RET *imm16*	Far return to calling procedure and pop *imm16* bytes from stack

Description

Transfers program control to a return address located on the top of the stack. The address is usually placed on the stack by a CALL instruction, and the return is made to the instruction that follows the CALL instruction.

The optional source operand specifies the number of stack bytes to be released after the return address is popped; the default is none. This operand can be used to release parameters from the stack that were passed to the called procedure and are no longer needed. It must be used when the CALL instruction used to switch to a new procedure uses a call gate with a non-zero word count to access the new procedure. Here, the source operand for the RET instruction must specify the same number of bytes as is specified in the word count field of the call gate.

The RET instruction can be used to execute three different types of returns:

- Near return—A return to a calling procedure within the current code segment (the segment currently pointed to by the CS register), sometimes referred to as an intrasegment return.
- Far return—A return to a calling procedure located in a different segment than the current code segment, sometimes referred to as an intersegment return.
- Inter-privilege-level far return—A far return to a different privilege level than that of the currently executing program or procedure.

RET

- When executing a near return, the processor pops the return instruction pointer (offset) from the top of the stack into the EIP register and begins program execution at the new instruction pointer.
- The CS register is unchanged.

ABI

- *ang. application binary interface*, interfejs binarny aplikacji
- **nie mylić z API**
- definiuje:
  - typy danych (rozmiar i ułożenie w pamięci)
  - sposób wywoływania funkcji (*ang. calling convention*): przekazywanie parametrów, zwracanie wartości, „właściciela” rejestrów, ramkę stosu
  - sposób wywoływania funkcji **systemowych**
  - formaty i nazwy plików bibliotecznych, nagłówkowych, programów, itd.

https://en.wikipedia.org/wiki/X86_calling_conventions

- cdecl
- syscall
- optlink
- pascal
- stdcall
- Microsoft __fastcall
- Microsoft __vectorcall
- Borland register
- Watcom register
- TopSpeed / Clarion / JPI
- safecall
- thiscall
- ...

SYSTEM V APPLICATION BINARY INTERFACE

Intel386™ Architecture Processor Supplement

Fourth Edition

ABI

Type	C	sizeof	Alignment (bytes)	Intel386 Architecture
Integral	`char` `signed char`	1	1	signed byte
	`unsigned char`	1	1	unsigned byte
	`short` `signed short`	2	2	signed halfword
	`unsigned short`	2	2	unsigned halfword

	`int`
	`signed int`
	`long`
	`signed long`
	`enum`
		4	4	signed word

	`unsigned int` `unsigned long`	4	4	unsigned word
Pointer	*any-type* `*` *any-type* `(*)()`	4	4	unsigned word
Floating-point	`float`	4	4	single-precision (IEEE)
	`double`	8	4	double-precision (IEEE)
	`long double`	12	4	extended-precision (IEEE)

System V Application Binary Interface Intel386™ Architecture Processor Supplement Fourth Edition

Aggregates and Unions

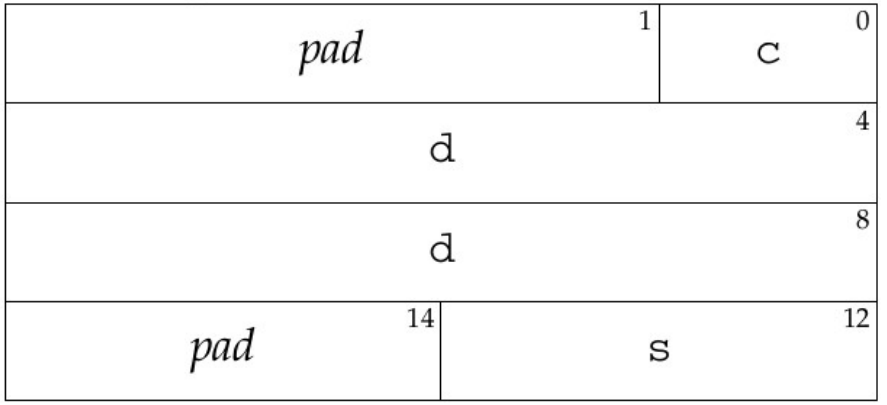
Aggregates (structures and arrays) and unions assume the alignment of their most strictly aligned component. The size of any object, including aggregates and unions, is always a multiple of the object's alignment. An array uses the same alignment as its elements. Structure and union objects can require padding to meet size and alignment constraints. The contents of any padding is undefined.

- An entire structure or union object is aligned on the same boundary as its most strictly aligned member.
- Each member is assigned to the lowest available offset with the appropriate alignment. This may require *internal padding*, depending on the previous member.
- A structure's size is increased, if necessary, to make it a multiple of the alignment. This may require *tail padding*, depending on the last member.

Figure 3-5: Internal and Tail Padding

```
struct {
    char   c;
    double d;
    short  s;
};
```

Word aligned, `sizeof` is 16



Ramka stosu (*ang. stack frame*)

Figure 3-15: Standard Stack Frame

Position	Contents	Frame	
4*n*+8 (%ebp)	argument word *n*	Previous	*High addresses*
	. . .		
8 (%ebp)	argument word 0		
4 (%ebp)	return address	Current	
0 (%ebp)	previous %ebp (optional)		
-4 (%ebp)	unspecified		
	. . .		
0 (%esp)	variable size		*Low addresses*

Ramka stosu (*ang. stack frame*)

- The stack is word aligned. Although the architecture does not require any alignment of the stack, software convention and the operating system requires that the stack be aligned on a word boundary.
- Argument words are pushed onto the stack in reverse order (that is, the rightmost argument in C call syntax has the highest address), preserving the stack's word alignment. All incoming arguments appear on the stack, residing in the stack frame of the caller.
- An argument's size is increased, if necessary, to make it a multiple of words.This may require tail padding, depending on the size of the argument.
- Other areas depend on the compiler and the code being compiled. The standard calling sequence does not define a maximum stack frame size, nor does it restrict how a language system uses the ‘‘unspecified’’ area of the standard stack frame.

```
$ ulimit -a
real-time non-blocking time  (microseconds, -R) unlimited
core file size              (blocks, -c) 0
data seg size               (kbytes, -d) unlimited
scheduling priority                 (-e) 0
file size                   (blocks, -f) unlimited
pending signals                     (-i) 253466
max locked memory           (kbytes, -l) 8126828
max memory size             (kbytes, -m) unlimited
open files                          (-n) 1024
pipe size                (512 bytes, -p) 8
POSIX message queues         (bytes, -q) 819200
real-time priority                  (-r) 0
stack size                  (kbytes, -s) 8192
cpu time                   (seconds, -t) unlimited
max user processes                  (-u) 253466
virtual memory              (kbytes, -v) unlimited
file locks                          (-x) unlimited
$
```

Konwencja wywoływania funkcji

- All registers on the Intel386 are global and thus visible to both a **calling** and a **called** function.
- Registers %ebp, %ebx, %edi, %esi, and %esp "belong" to the **calling** function. In other words, a called function must preserve these registers' values for its caller.
- Remaining registers "belong" to the **called** function. If a calling function wants to preserve such a register value across a function call, it must save the value in its local stack frame.

Konwencja wywoływania funkcji

- A function that returns an integral or pointer value places its result in register %eax.
- A floating-point return value appears on the top of the Intel387 register stack. The caller then must remove the value from the Intel387 stack, even if it doesn't use the value. Failure of either side to meet its obligations leads to undefined program behavior. The standard calling sequence does not include any method to detect such failures nor to detect return value type mismatches. Therefore the user must declare all functions properly. There is no difference in the representation of single-, double- or extended-precision values in floating-point registers.
- Functions that return no value (also called procedures or void functions) put no particular value in any register.
- If a function returns a structure or union, then the caller provides space for the return value and places its address on the stack as argument word zero. In effect, this address becomes a ‘‘hidden’’ first argument. Having the caller supply the return object's space allows re-entrancy.

Figure 3-16: Function Prologue

```
prologue:
        pushl  %ebp             / save frame pointer
        movl   %esp, %ebp       / set new frame pointer
        subl   $80, %esp        / allocate stack space
        pushl  %edi             / save local register
        pushl  %esi             / save local register
        pushl  %ebx             / save local register
```

Figure 3-17: Function Epilogue

```
        movl   %edi, %eax       / set up return value
epilogue:
        popl   %ebx             / restore local register
        popl   %esi             / restore local register
        popl   %edi             / restore local register
        leave                   / restore frame pointer
        ret                     / pop return address
```

Figure 3-21: Integral and Pointer Arguments

Call	Argument	Stack address
`g(1, 2, 3, (void *)0);`	1	8 (%ebp)
	2	12 (%ebp)
	3	16 (%ebp)
	(void *)0	20 (%ebp)

Figure 3-22: Floating-Point Arguments

Call	Argument	Stack address
`h(1.414, 1, 2.998e10);`	word 0, 1.414	8 (%ebp)
	word 1, 1.414	12 (%ebp)
	1	16 (%ebp)
	word 0, 2.998e10	20 (%ebp)
	word 1, 2.998e10	24 (%ebp)

Figure 3-23: Structure and Union Arguments

Call	Argument	Callee
`i(1, s);`	1	8 (%ebp)
	word 0, s	12 (%ebp)
	word 1, s	16 (%ebp)
	...	...

Konwencje wywołania

Konwencja wywołania
– sposób przekazywania parametrów do i z funkcji (ang. *calling conventions*)
Proces przekazywania parametrów – powiązanie procedury (ang. *subroutine linkage*)

Zagnieżdżanie funkcji
– rejestr powiązań (ang. *link register*) lub adres powrotu na stosie

Konwencje:
– przez rejestry procesora
metoda szybka, liczba parametrów ograniczona
– przez obszar powiązania danych (ang. *subroutine linkage*)
adres obszaru przekazywany przez rejestr procesora
– przez stos programowy
w obszarze stosu tworzona ramka dla parametrów (kontekst/blok aktywacji)
adresowana wskaźnik ramki (ang. *frame pointer*)

Mechanizm okien rejestrowych – przekazywanie parametrów i powiązania przez okno rejestrowe

Korzyści z mechanizmu funkcji i problemy

Korzyści:

- *redukcja rozmiaru kodu (usunięcie powtarzalnych fragmentów programu)*
- *redukcja zapotrzebowania na pamięć*
- *ukrycie szczegółów implementowanego algorytmu i struktury danych*
  - możliwość modyfikacji algorytmu bez zmiany sposobu użycia
- *łatwa implementacja wyższego poziomu abstrakcji – maszyny wirtualnej*
  - funkcja = makrorozkaz →
    → lista makrorozkazów = architektura witrualna

Problemy:

- *naruszenie sekwencyjności rozkazów podczas wywołania*
- *nieprzewidywalność lokalizacji kolejnego rozkazu podczas powrotu*

!! *UWAGA:*

Funkcja – makrorozkaz, złożone polecenie wykonywane przez procesor

Makro – tekstowy opis w pliku źródłowym (*makrodefinicja*), przetwarzany przez kompilator (*makrowywołanie*) na sekwencję instrukcji podczas generowania kodu

Generacje języków programowania

- 1GL - kod maszynowy
- 2GL - język asemblera
  - automatyczne obliczanie adresów (symbole, etykiety), udogodnienia zwiększające czytelność (mnemoniki, komentarze, dyrektywy, makra, literały liczbowe i znakowe) i zarządzanie kodem (łatwe łączenie modułów)
- 3GL
  - C, C++, Java, Python, PHP, Perl, C#, BASIC, Pascal, Fortran, ALGOL, COBOL
  - wysokopoziomowe abstrakcje (programowanie strukturalne i obiektowe) ale dopasowane do ograniczeń maszyny, czytelność, przenośność
  - konieczna translacja (kompilator/interpreter)
- 4GL
  - ABAP, Unix Shell, SQL, PL/SQL, Oracle Reports, R
  - abstrakcje dopasowane do dziedziny (problemów), a nie maszyny
- 5GL
  - Prolog, Clipper, LabView
  - Opis problemu i wymagań nałożonych na wynik, brak algorytmu

Języki pierwszej generacji

https://www.liveauctionworld.com/STAG-PP28-EPROM-EEPROM-PROGRAMMER_i13697872

Standard języka

http://www.open-std.org/

Open Standards

The site www.open-std.org is holding a number of web pages for groups producing open standards:

- ISO/IEC JTC1/SC2 - character sets
  - WG3 - 7- and 8-bits character sets
- ISO/IEC JTC1/SC22 - Programming languages and, operating systems
  - WG9 - Ada
  - WG11 - Binding Techniques
  - WG14 - C
  - WG15 - POSIX
  - WG16 - ISLISP
  - WG19 - Formal Specification Languages
  - WG20 - Internationalization
  - WG21 - C++
  - WG23 - Programming Language Vulnerabilities
  - Internationalization Rapporteur Group
  - JSG - Java Study Group
  - prototype WG web pages
- ISO/IEC JTC1/SC34 - Document Description and Processing Languages
  - WG5 - Document Interoperability
- ISO/IEC JTC1/SC35 - User interfaces
  - WG1 - Keyboards and input interfaces
  - WG5 - Cultural, Linguistic and User Requirements
  - WG6 - User interfaces for the disabled
  - WG8 - User Interfaces for Remote Interaction
- ISO/IEC JTC1/IIT-RG - Rapporteur Group on Implementation of IT in JTC1
- ISO/IEC JTC1/CLAUI - Technical Direction on Cultural Adaptability in JTC1

- CEN/TC304 - European Cultural Localization

The main site for this is located at http://www.open-std.org/ and there is live mirrors of the site at http://www7.open-std.org/.

Comments are welcome! We can be reached at webmaster@open-std.org
This site is sponsored by DTU

Standard C

- Rozmiary typów danych
  - `short, int`: co najmniej 16 bitów
  - nadmiar: niezdefiniowany:

    If an exceptional condition occurs during the evaluation of an expression (that is, if the result is not mathematically defined or not in the range of representable values for its type), the behavior is undefined.

    ISO/IEC 9899:2011 N1570 &6.5 p. 5