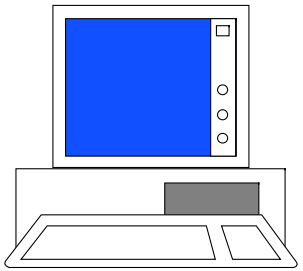
Jak działa komputer

Organizacja przetwarzania

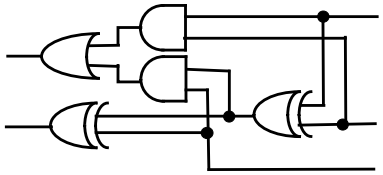
- **ustalona** jednostka informacji: **słowo (1 bajt, 2 bajty, 4 bajty, ...)**
  - procesor przetwarza **ciągi bitów (1 bajt=8bitów)**
  - **treść słowa zależy od interpretacji** przez CPU (stan – instrukcja)
  - treść **słowa/słów rozkazu** (polecenia, instrukcji)
    określa **lokalizację słów argumentów i wyniku**
- zbiór informacji jednostkowych jest **uporządkowany**
  – każde słowo musi mieć przypisany **adres (**numer porządkowy)
- konieczne wskazanie **lokalizacji** (adresu) słowa/słów kodu kolejnego rozkazu
  - automatycznie przez domniemanie (kolejne):
    – **wskaźnik instrukcji** (licznik programu, licznik rozkazów)
  - przez wymuszenie wskazane w treści kodu działania
    – **rozgałęzienia (**instrukcje warunkowe: **jeśli, to ..)**
    – **skoki** (wymuszenie nowego stanu wskaźnik instrukcji)
    – **przekazanie sterowania** (wywołanie/zakończenie funkcji/procedury)
- musi być **ustalona lokalizacja pierwszego rozkazu**

Architektura i organizacja komputera

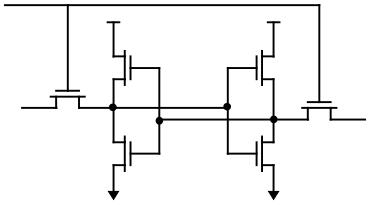
ARCHITEKTURA KOMPUTERA
specyfikacja funkcjonalnych cech komputera
opisanych za pomocą listy rozkazów
(wskazanie argumentów i działania)
(architektura listy rozkazów – ISA)



ORGANIZACJA KOMPUTERA
struktura (logiczna)
odwzorowująca cechy funkcjonalne
i nadająca architekturze kształt operacyjny
(architektura układów – HSA)



WYKONANIE
(TECHNOLOGIA)



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

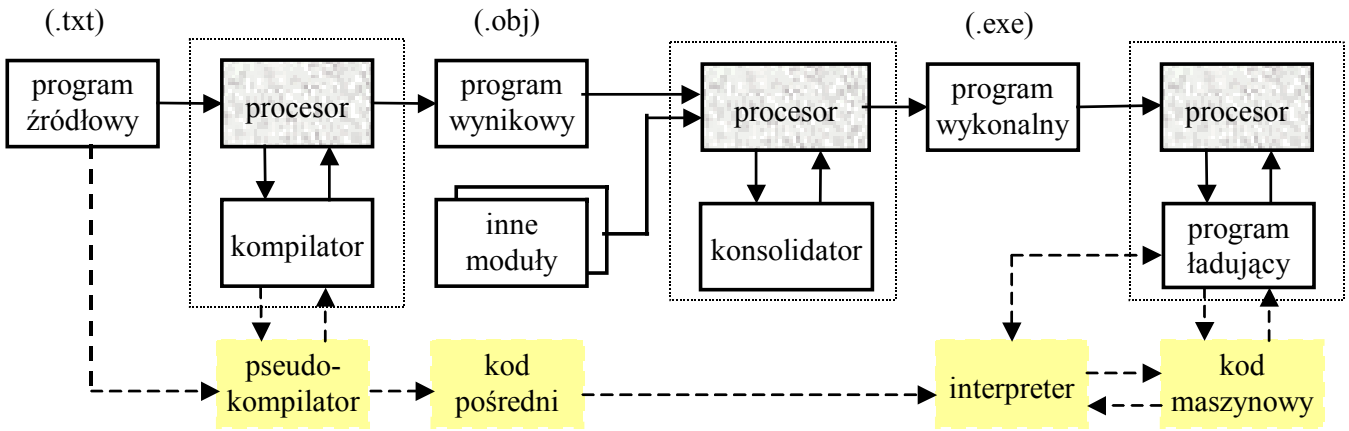
Załadowanie kodu do pamięci alokowanej (ang. *loading*)

- 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



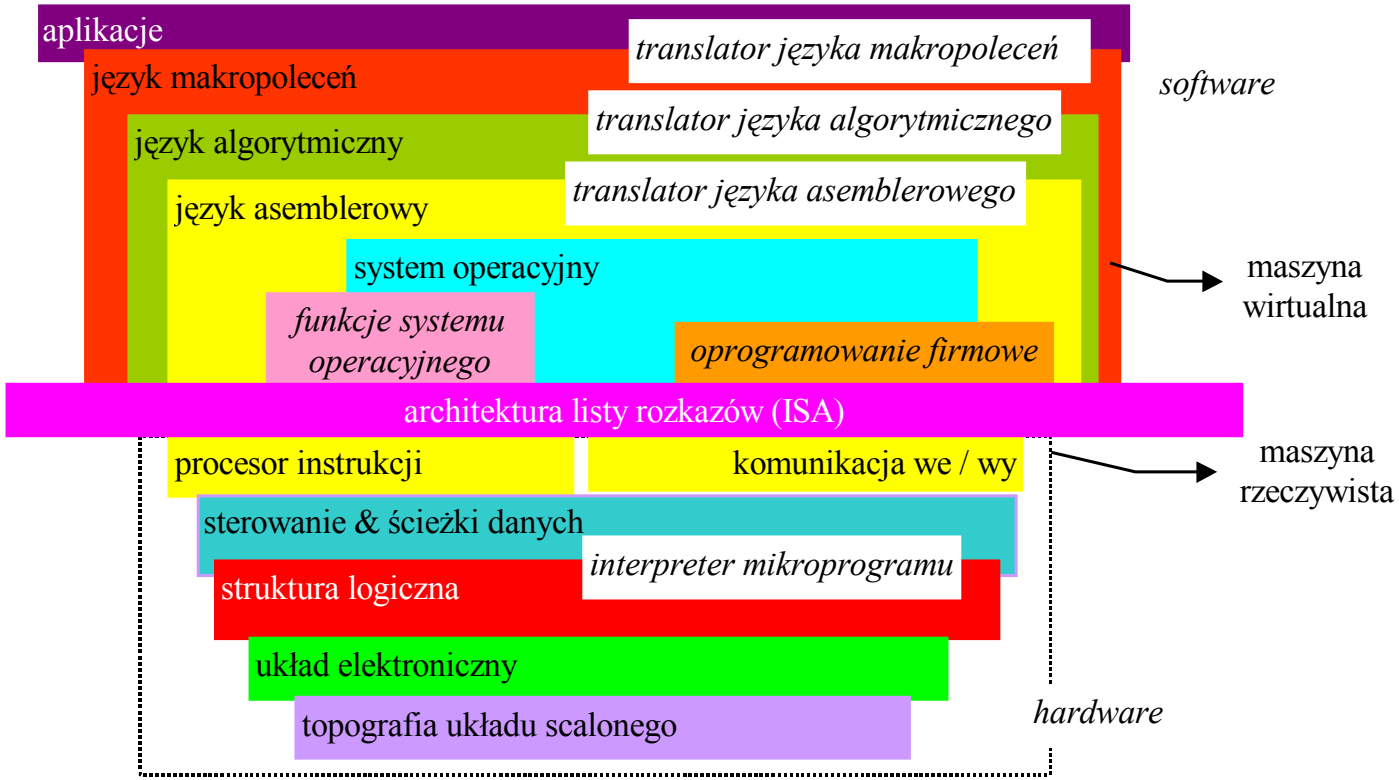
Cechy architektury ułatwiające kompilację

- *ortogonalność* zestawu działań i sposobów adresowania argumentów
- *kompletność* – szeroki repertuar działań
- *regularność* – jednorodność opisu działań
- *oszczędność* – niewiele prostych rozkazów i sposobów adresowania

dużo rejestrów – łatwość przypisania rejestrów

→ cechy dobrej (przejrzystej) architektury

Poziomy maszynowe*)



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

Poziomy maszynowe i zapis działań (1)*)

L4 – PASCAL

var A, B, C: integer;

....

A:=B+C;

L4 – C

integer A, B, C

...

A=B+C

L3 – Asembler (MC 680x0)

L1 – kod maszynowy (MC 680x0)

- A DS.W ; (adres A = 2002h)
- B DS.W ; (adres B = 2004h)
- C DS.W ; (adres C = 2006h)
- MAIN

• (*): MOVE.W B, D1	0011 0010	0011 1000	(32 38)
	0000 0000	0000 0000	(00 00)
	0010 0000	0000 0100	(20 04)
• (*+6): ADD.W C, D1	1101 0010	0111 1000	(D2 78)
	0000 0000	0000 0000	(00 00)
	0010 0000	0000 0110	(20 06)
• (*+C): MOVE.W D1, A	0011 0001	1100 0001	(31 C1)
	0000 0000	0000 0000	(00 00)
	0010 0000	0000 0010	(20 02)

Poziomy maszynowe i zapis działań (2)*)

L3 – asm (Intel/Borland)	L3 – asembler (AT&T/GNU)	L1 – kod maszynowy (Intel x86)	
segment data (.data)	.data	offset (hex):	
cc dd (?)	cc	.space 4	; (cc = 00 00 01 03)
bb dd (?)	bb	.space 4	; (bb = 00 00 01 05)
aa dd (?)	aa	.space 4	; (aa = 00 00 01 07)
segment code (.code)	.text		
...			
(*) mov eax, dword_ptr ds:cc	movl cc, %eax	1100 1100	(66)
		1010 0001	(A1)
		0000 0011	(03)
		0000 0001	(01)
(*+3) add eax, dword_ptr ds:bb	addl bb, %eax	1100 1100	
		0000 0011	
		0000 0110	(66 03 06)
		0000 0101	
		0000 0001	(01 05)
(*+9) mov dword_ptr ds:aa, eax	movl %eax, aa	1100 1100	
		1010 0011	(66 A3)
		0000 0111	
		0000 0001	(01 07)

Symboliczny zapis algorytmu

alfabet (ang. *alphabet*) – zbiór zdefiniowanych symboli,
najczęściej ***alfanumerycznych*** i ***specjalnych*** ($, %, &,...)

semantyka (ang. *semantic*) – znaczenie ciągów symboli (wyrazów)

- **mnemonik** – symboliczny zapis działania,
  repertuar specyficzny dla architektury, precyzyjnie określone efekty
- ***nazwy własne*** obiektów /zmiennych

składnia (ang. *syntax*) – reguły symbolicznego zapisu działań

mnemonik argument, argument, ...

identyfikacja (adresowanie, wskazanie) argumentów – tryby adresowania

- **stałe** (kod skrócony) – w kodzie rozkazu (wartość) (adr. natychmiastowe)
- **zmienne**
  - tymczasowe (w rejestrze – adr. bezpośrednie) – nazwa/numer rejestru
  - nazwane (specyfikowane) (w pamięci – adr. pośrednie) – adres w pamięci:
    składowe i sposób obliczenia adresu oraz rozmiar jednostki (typ)

Konwencje opisu działań

składnia – sposób zapisu działania

języki symboliczne:

wynik = *argument*-1 DZIAŁANIE *argument*-2 DZIAŁANIE …

albo

wynik = FUNKCJA (*argument*-1, *argument*-2,…)

asembler (poziom maszyny rzeczywistej):

mnemonik *argument*-1, *argument*-2,…,

wywołanie *adres* – *konwencje przekazywania argumentów i wyniku*

!! każdy argument/zmienna ma przypisane **atrybuty**: ***adres***, ***typ***, ***rozmiar***

wskazywanie argumentów – tryby (sposoby) adresowania

języki symboliczne: *wskaźniki / nazwy zmiennych (obiektów), stałe*

asembler: *adresy słów w pamięci, nazwy rejestrów procesora, stałe*

Model programowy maszyny rzeczywistej

architektura listy rozkazów

specyfikacja argumentów

- ***argumenty nazwane*** – **zmienne** (lokacje) w pamięci (tryby adresowania)
  tryb adresowania – sposób identyfikacji /wskazywania argumentów
- ***argumenty tymczasowe*** – **rejestry** procesora

specyfikacja działań

- ***sterowanie***
  → rozgałęzienie – warunkowy wybór ścieżki przetwarzania
  → skok ze śladem
  → warunkowe wykonanie działania
- ***funkcje*** (działania zwracające wynik):
  → arytmetyczne
  → logiczne
  → zmiany formatu i przekodowania
  →kopiowanie

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 – specyfikacja argumentów i działań

specyfikacja rejestrów

- rejestry ogólnego przeznaczenia *GPR* (*general purpose registers*)
- rejestry specyficzne (niespójność architektury)

tryby adresowania

- sposoby tworzenia adresu w logicznej przestrzeni adresowej
- ograniczenia (niespójność architektury)

specyfikacja działań

- sposób tworzenia wyniku i jego syndromów
  - zasady
  - odstępstwa od zasad (niespójności architektury)
- ograniczenia użycia argumentów
  - nakaz użycia
  - zakaz użycia
- interpretacja argumentów
  - interpretacja kodów liczb
  - sposób tworzenia i przekształcania stałych (rozszerzenia kodu)

Struktura programu w języku asemblera

Definicje stałych symbolicznych

- lokalne
- tymczasowe

Dyrektywy – polecenia dla kompilatora i linkera

- dyrektywy organizacyjne – opis struktury programu
- makrodefinicje i makroinstrukcje

Deklaracje zmiennych i buforów pamięci

- *nazwa* typ *lista*
- *nazwa* dyrektywa *rozmiar*

Instrukcje procesora – składnia i semantyka

– składnia

mnemonik argument_akumulujący, argument_swobodny

– specyfikacja typu argumentów

- przez domniemanie zgodności rozmiaru
- przez jawne wskazanie rozmiaru elementu w pamięci

Asembler TASM/MASM/NASM - składnia firmowa Intel

Dyrektywy

– SEGMENT, MODEL, END, MACRO,…

Deklaracje zmiennych i buforów pamięci

nazwa db/dw/dd/dq *lista – zmienne inicjowane*

nazwa ds *rozmiar – bufory nieinicjowane oraz inicjowane*

Instrukcje procesora – składnia i semantyka

– składnia

mnemonik argument_akumulujący, argument_swobodny

– jawna specyfikacja typu argumentów: (.._ptr)

– pamięć segmentowana – kompletna specyfikacja elementu pamięci

adres = nazwa_segmentu : zmienna(rejestr_bazowy, rejestr_indeksujący, skala):

dword_ptr es:tab[eax, ebx, 4]

- segment kodu – inicjowany automatycznie: wskaźnik CS
- segmenty danych – wymagają zainicjowania: wskaźniki DS, ES, FS, GS
- segment stosu – inicjowany automatycznie: wskaźnik SS

Asembler AT&T (IA-32/IA-32e)

as /gcc – liniowy model pamięci, składnia AT&T (Linux/UNIX)

Dyrektywy – słowa poprzedzone kropką: .globl, .type, .skip,

Deklaracje zmiennych i buforów pamięci
– deklaracje zmiennych inicjowanych
nazwa: .*typ* *lista*
– deklaracje buforów nieinicjowanych
nazwa: .space *rozmiar*
Instrukcje procesora – składnia i semantyka
– składnia
mnemonik[*rozm*] *argument_swobodny*, *argument_akumulujący*
– specyfikacja rozmiaru argumentów – za pomocą przyrostka [rozm]

Pamięć liniowa
tryby adresowania – reguły
zmienna(***rejestr_bazowy***, ***rejestr_indeksujący***, skala)

Struktura programu w składni AT&T / GNU

– sekcje o różnych uprawnieniach dostępu

.text – sekcja programu – tylko wykonanie

.data – sekcja zmiennych – dozwolony zapis

nazwa ***.typ*** *lista* – deklaracja zmiennej

.bss – sekcja bufora – poza pamięcią programu

nazwa ***.space*** *rozmiar* – deklaracja bufora (rozmiar w bajtach)

– składnia

mnemonik[*rozm*] *argument_swobodny*, *argument_akumulujący*

– *argument_swobodny*: $const, %reg, nazwa(%baza, %index, skala)

– *argument_akumulujący*: %reg, nazwa(%baza, %index, skala)

– **tylko jeden argument** może być specyfikowany jako **słowo w pamięci!!**

– [*rozm*]: b – bajt, l – (long) słowo 4-bajtowe, q – (quadbyte) słowo 8-bajtowe

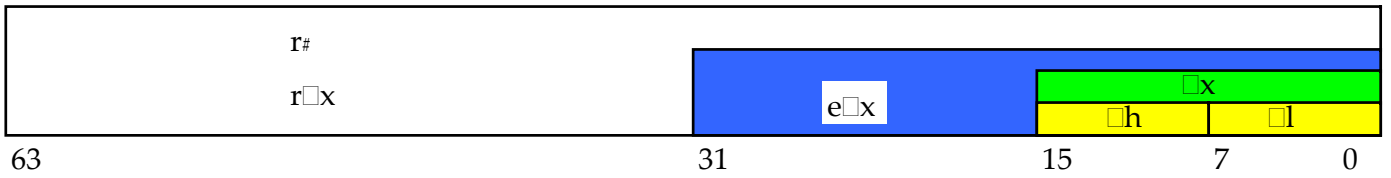
stałe:

- wartość poprzedzona znakiem $ (z oczywistymi odstępstwami, np. skala)
- 0x..., 0b..., 0q... – w notacji szesnastkowej, binarnej, ósemkowej

Architektura IA-32, IA-32e (1)

model programowy –

- rejestry stałoprzecinkowe a, b, c, d w wersjach:
  - 8-bitowych □h, ?l (al, ah, bl, bh, cl, ch, dl, dh)
  - 16-bitowych □x (ax, bx, cx, dx)
  - 32-bitowych e□x (eax, ebx, ecx, edx)
  - 64-bitowych r□x (rax, rbx, rcx, rdx) albo r# (r0, r1,..., r15)



- rejestry adresowe
  - 16-bitowe (si, di, bp, sp)
  - 32-bitowe (esi, edi, ebp, esp)
  - 64-bitowe (rsi, rdi, rbp, rsp)
- rejestry segmentowe (adres bloku)
  - 16-bitowe (cs, ds, es, fs, gs, ss)

Architektura IA-32, IA-32e (2)

- rejestry specyficzne – flagi (cf, of, sf, zf), indeksowanie łańcucha (if, df)

rejestry innych jednostek

- rejestry zmiennoprzecinkowe (FPU): ST(0), ST(1),..., ST(7)
- rejestry wektorów stałoprzecinkowych (MMX) – mm0, mm1, …, mm7
- rejestry wektorów zmiennoprzecinkowych (SSE) – xmm0,…xmm7

konwencje domniemanego użycia rejestrów

- rejestr akumulatora (al, ah, ax, eax, rax, dx:ax, edx:eax)
- rejestry zliczające (cx, ecx, rcx)
- wskaźniki elementów łańcuchów słów (esi, edi)

Architektura IA-32, IA-32e

AT&T / Linux / UNIX – tzw. płaski (liniowy) model pamięci

zmienna(*rejestr_bazowy*, *rejestr_indeksujący*, skala)

- zmienna – zadeklarowana nazwa zmiennej indeksowanej
- baza: eax, ebx, ..., (IA-32e: (rax, rbx, ...)/(r0, r1, r2, ...), r8,…,r15)
- indeks: jak baza oprócz esp/rsp
- skala: rozmiar słowa: 1 (bajt), ~~2(półsłowo)~~, **4 (słowo)**, 8 (dwusłowo 64b)

tab(%eax, %ebx, 4)

movb tab(%eax, %ebx, 4), %cl

(najniższy bajt słowa – konwencja *LE* (ang. *little endian*))

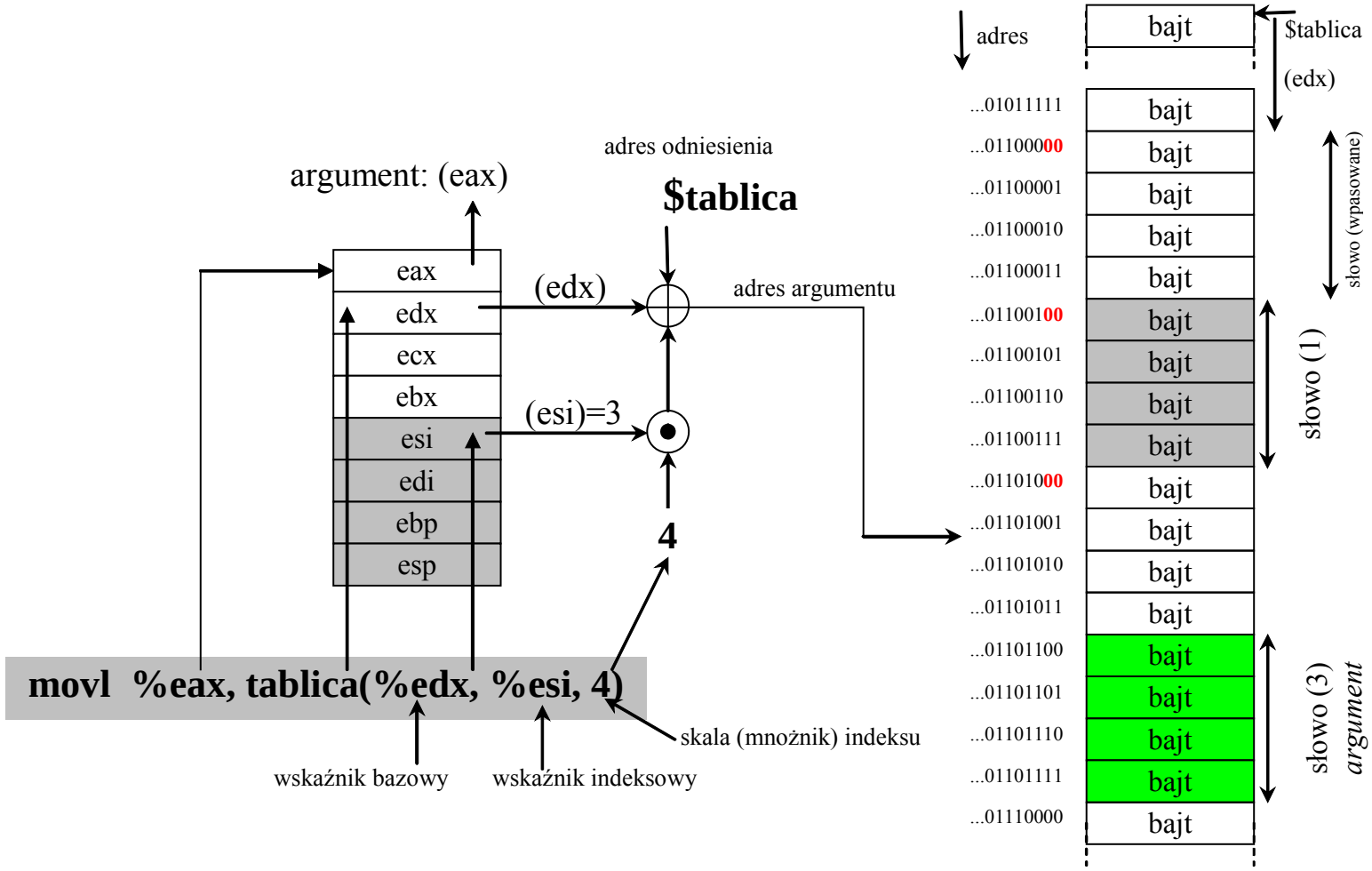
movl tab(%eax, %ebx, 4), %ecx

(słowo 4-bajtowe)

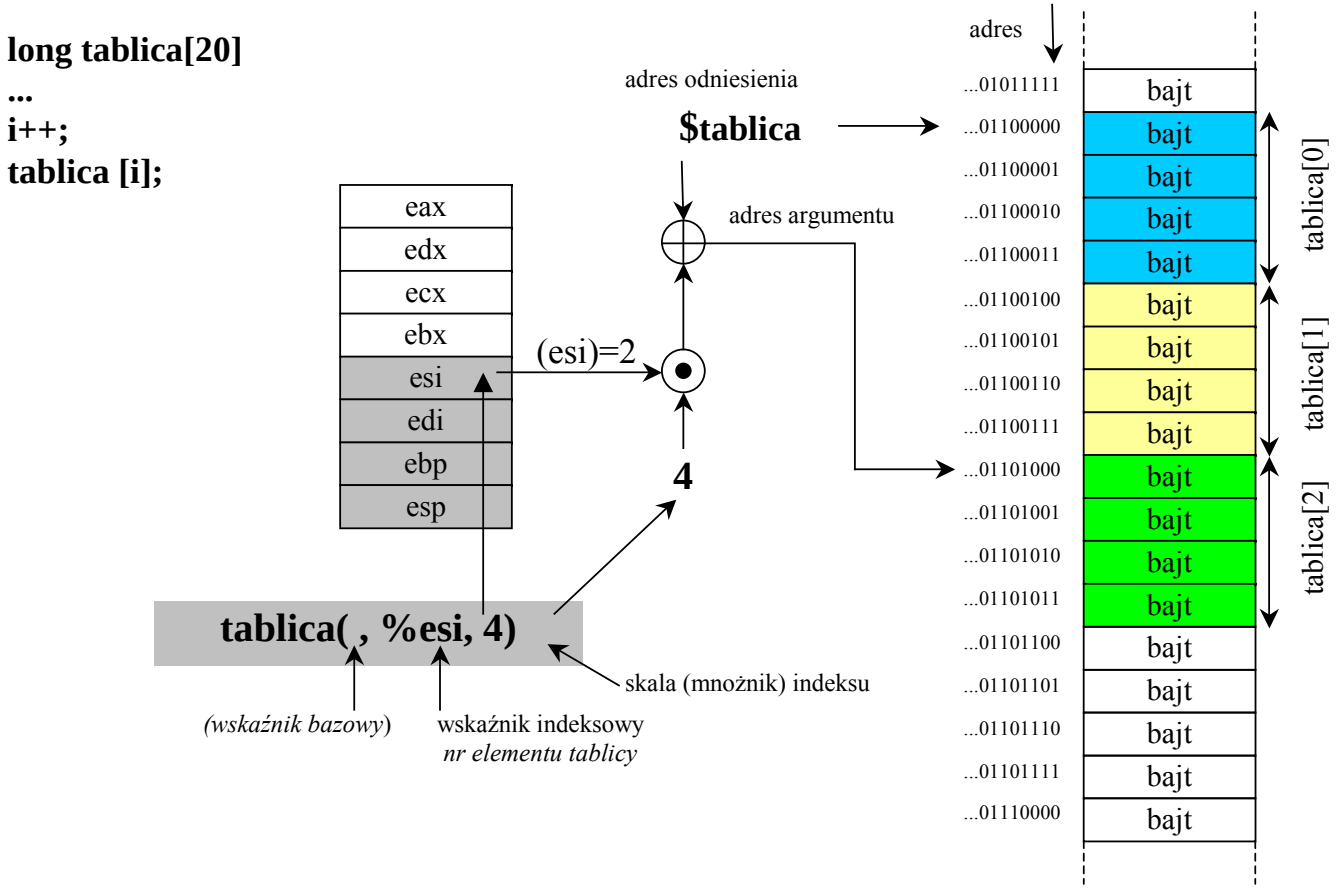
movd tab(%rax, %rbx, 8), %ecx

(dwusłowo 8-bajtowe)

Architektura IA-32/IA-32e – adresowanie



Architektura IA-32/IA-32e – adresowanie tablicy jednowymiarowej



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

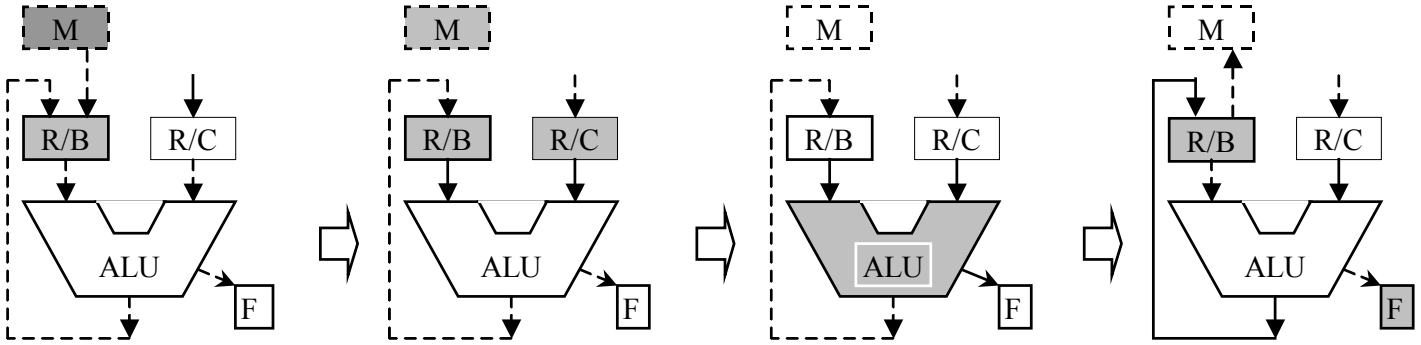
Podstawowy zestaw użytecznych instrukcji (IA-32/IA-32e)

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

Specyfika działań stałoprzecinkowych (IA-32/IA-32e)

architektura akumulatorowa uogólniona:



M – słowo w pamięci, R/B – rejestr/bufor pamięci, R/C – rejestr/stała, F – kody war.

domniemana lokalizacja wyniku – argument akumulujący

wynik (argument akumulujący, argument swobodny) → argument akumulujący

(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**

(IA-32/IA-32e) instrukcje logiczne

not (ang. *and*) – iloczyn logiczny (bitowy)

! nie zmienia kodów w rejestrze flag (F)

or (ang. *or*) – suma logiczna (bitowa)

xor (ang. *exclusive or*) – suma logiczna wyłączająca (bitowa)

and (ang. *and*) – iloczyn logiczny (bitowy)

test (ang. *test*) – iloczyn logiczny (bitowy) bez tworzenia wyniku

! ustawiane kody w rejestrze flag (F): CF=0, OF=0, ZF, SF,

xorl %ebx, %ecx	**- (ecx)=wynik operacji xor na każdej parze bitów rejestrów ebx oraz ecx** ($ecx_i := ecx_i \oplus ebx_i$)
xorb $-1, %al	**- negacja bitów rejestru al, pozostałe bity rejestru eax/rax bez zmian, !ustawia kody w rejestrze flag**
orb $0x20, %cl	**- wymuszenie „1” na bicie nr 5 w rejestrze cl**
andl $0x71, %eax	**- wyzerowanie wszystkich bitów rejestru eax oprócz bitów nr 0,4,5,6**

(IA-32/IA-32e) dodawanie, odejmowanie, porównanie

add (ang. *add*) – sumuj
adc (ang. *add with carry*) – sumuj wraz zprzeniesieniem (CF)
sub (ang. *subtract*) – oblicz różnicę
sbb (ang. *subtract with borrow*) – oblicz różnicę wraz z pożyczką (CF)
cmp (ang. *compare*) – oblicz różnicę nie tworząc wyniku

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

!nie ma instrucji cmpc (with carry)
– porównanie długich liczb musi być wykonane za pomocą sbb

add %esi, (%eax) **- suma zawartości słowa z pamięci o adresie (eax) oraz rejestru esi zapisana do pamięci pod adresem (eax), F!**
sbb (%esi), %edi **- różnica zawartości rejestru edi i słowa z o adresie (esi) minus pożyczka (CF) zapisana w rejestrze edi, F!**
cmp (%esi), %edi **- różnica zawartości rejestru edi i słowa o adresie (esi) nie zapisana w rejestrze edi, F!**

(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

(IA-32/IA-32e) mnożenie

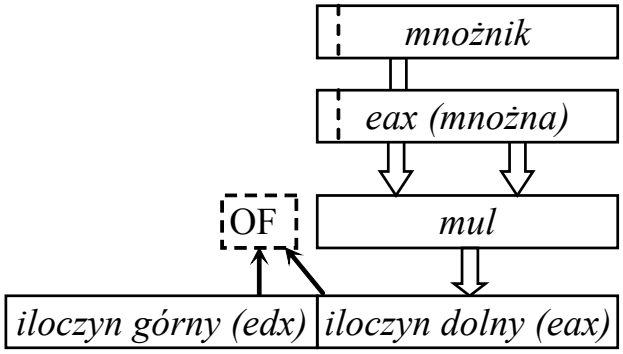
mul (ang. *multiply*) – wymnóż jak liczby naturalne

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**)

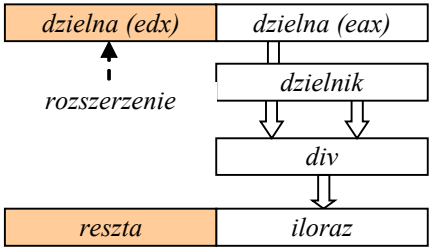
imull %ecx, $const ; (**edx:eax**):=(**eax**)*(**ecx**)

(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)
```

(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

(IA-32/IA-32e) wywołanie procedury/funkcji

call (ang. *call*) – skocz (przekaż sterowanie) ze śladem (adres powrotu)
ret (ang. *return*) – zwróc sterowanie do miejsca wywołania (adres powrotu)

przekazywanie parametrów:
- przez rejestry (szybkie), wykluczone w rekurencji
- przez stos (uniwersalne)
  push
  pop

blok aktywacji / kontekst funkcji:
- przekazywane parametry
- adres powrotu
- wskaźnik do kontekstu poziomu wywołującego
- dynamiczna struktura poziomu wywołanego

Instrukcje pomocnicze

cpuid (ang. ***cpu id****entification*) – identyfikacja procesora i specyfikacja wersji
rdtsc (ang. ***read time stamp counter***) – odczyt licznika cykli procesora
movsx (ang. ***move sign extended***) – kopiowanie z roszerzeniem znakowym (U2)
movzx (ang. ***move zero extended***) – kopiowanie z roszerzeniem zerowym (NB)
stc (ang. ***set carry***) – ustawienie CF=1
clc (ang. ***clear carry***) – ustawienie CF=0
bswap (ang. ***byte swap***) – odwrócenie kolejności bajtów w słowie
cwde / cdq – rozszerzenie znakowe akumulatora **a** (eax na edx:eax/rax na rdx:rax)

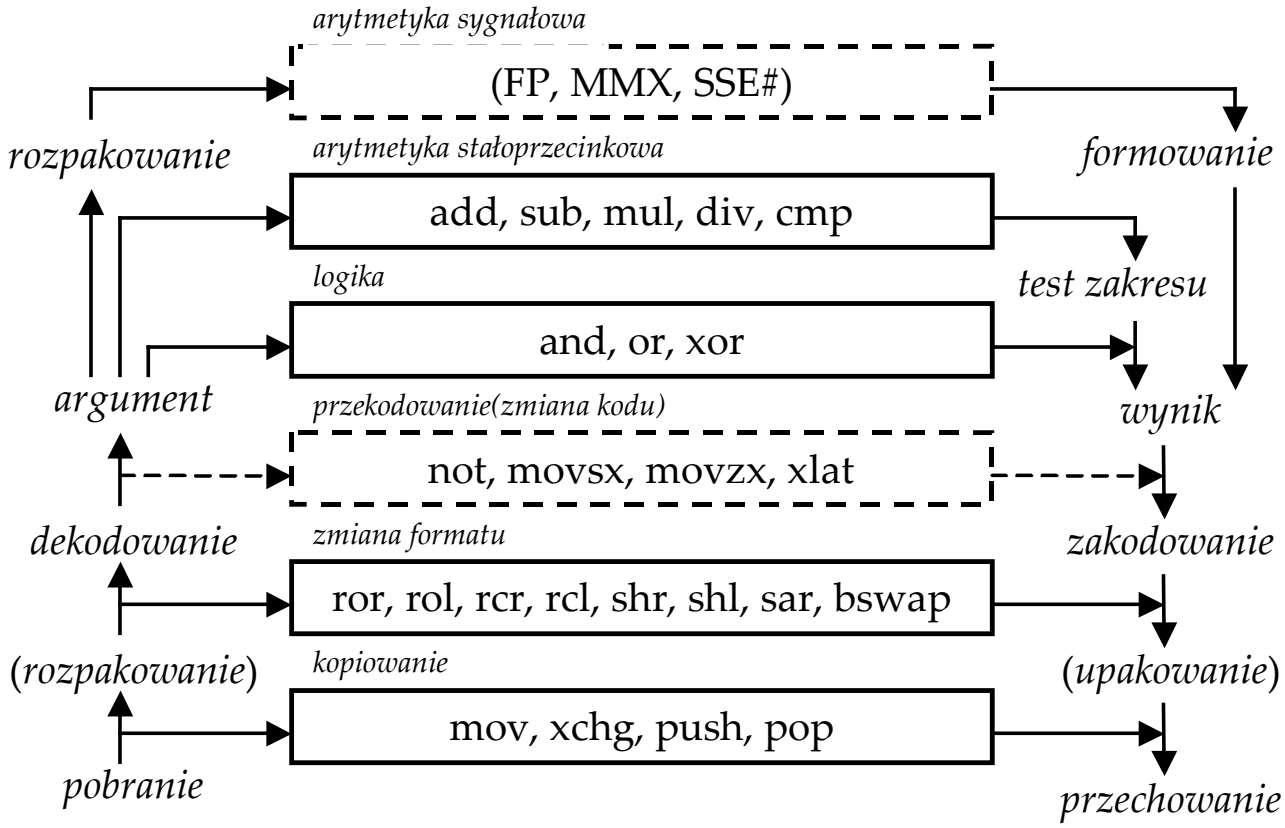
rotacje (przesunięcia cykliczne) i przesunięcia
rol (ang. *rotate left through **c**arry*) – przesunięcie cykliczne bitów w lewo
ror (ang. *rotate right through **c**arry*) – przesunięcie cykliczne bitów w prawo
rlc (ang. *rotate left through **c**arry*) – przesunięcie bitów w lewo z dołączonym CF
rrc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

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*) –

Działania na danych (1)



Klasyfikacja działań i fazy ich wykonania (format = struktura)

Działania na danych (2)

kopiowanie

- niszczące, nieodwracalne – ***mov, push, pop***
- kopiowanie bloków – ***movs*** (! kolejność przesłań)
- wymiana (*exchange*) – ***xchg*** (odwracalne)

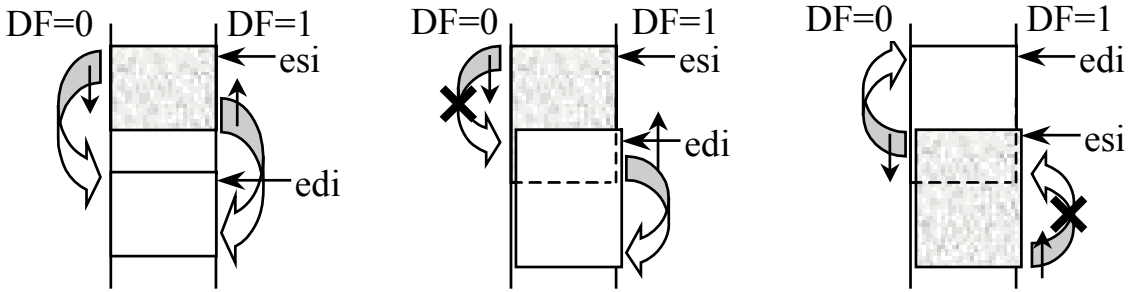
zmiana formatu (zmiana wzajemnego położenia bitów w słowie)

- przemieszczenie pól (rekordów) – ***bswap*** (przestawienie bajtów)
- systematyczne przemieszczenie bitów
  - zwykłe – przesunięcie arytmetyczne lub logiczne – ***sar, shr, shl***
  - cykliczne – rotacja prosta i rozszerzona – ***rol, ror, rcl, rcr***

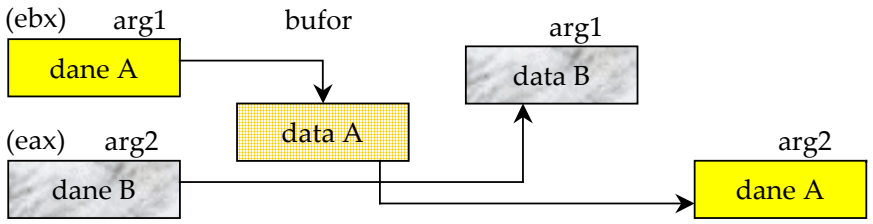
zmiana kodu (zmiana wartości niektórych bitów poza ALU)

- rozszerzanie kodów liczb – ***movsx, movzx***
- negacja bitów – ***not***
- konwersje formatów liczb (zmiennoprzecinkowy ↔ stałoprzecinkowy)
- przekodowanie tablicowe – ***xlat***
- upakowanie i rozpakowanie kodu BCD

Kopiowanie



Transfer łańcucha słów **movs**

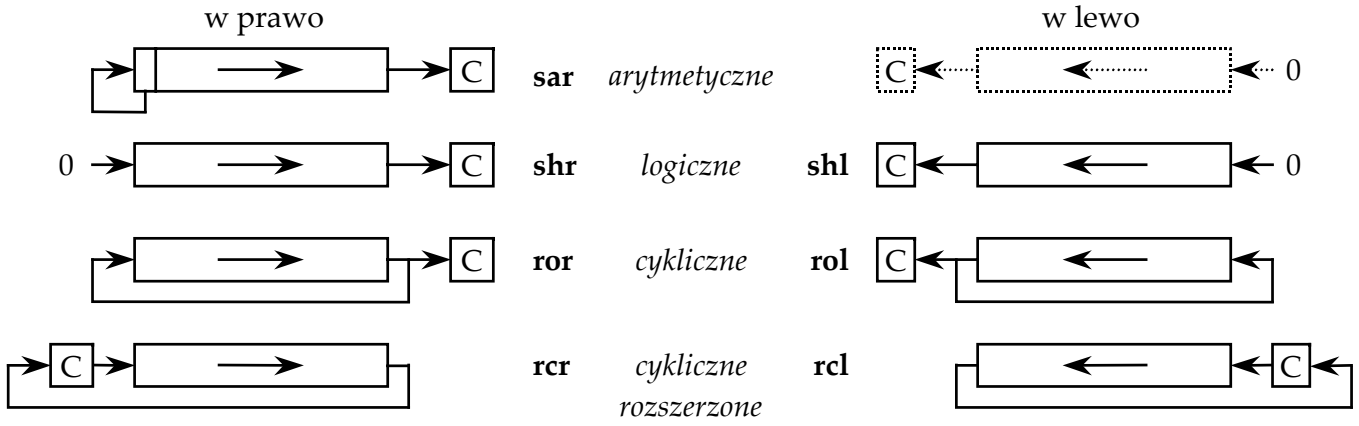


wymiana bez bufora:

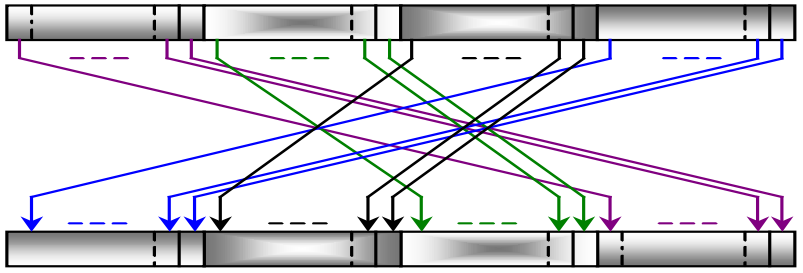
```
xor %eax, %ebx
xor %ebx, %eax
xor %eax, %ebx
```

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

Zmiana formatu

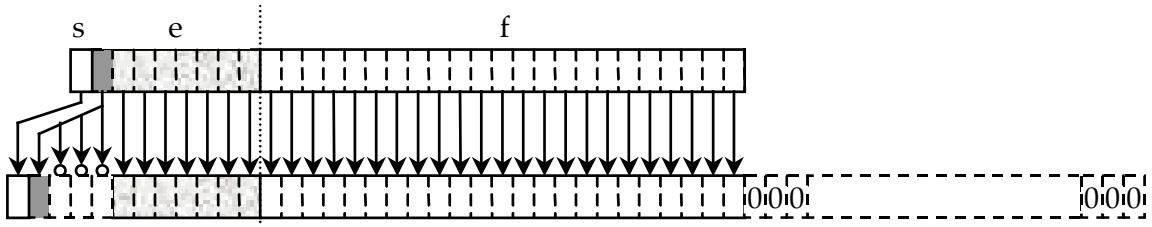


Schematy przesunięć i rotacji (przesunięć cyklicznych)

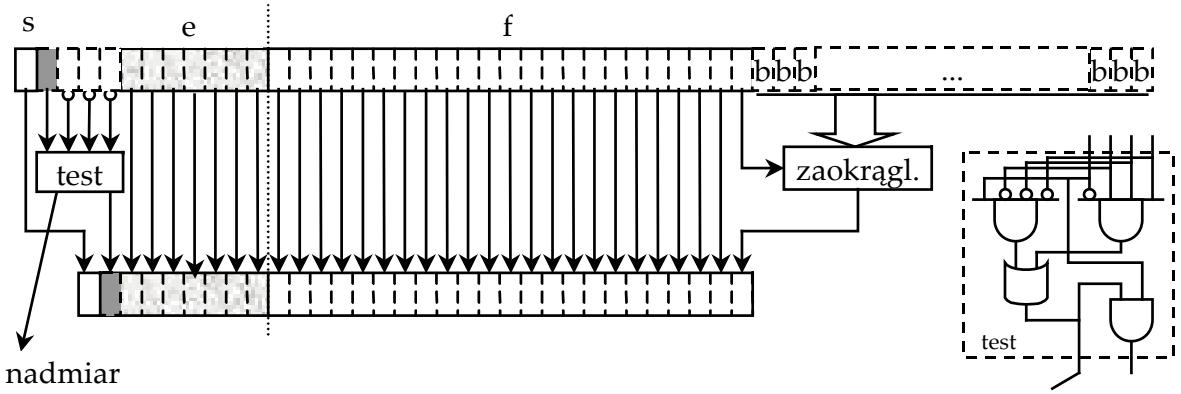


Przestawienie bajtów (bswap)

Zmiana kodu

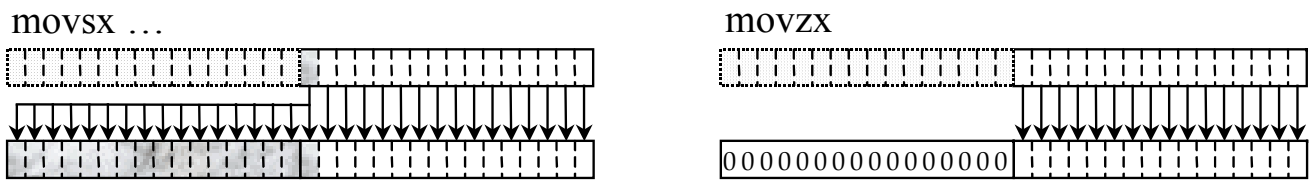


Konwersja formatu zmiennoprzecinkowego (o – negowanie bitu)

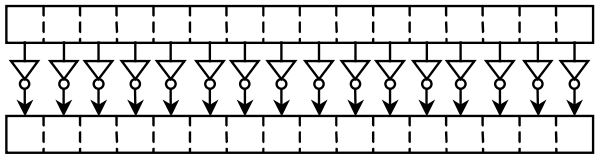


Konwersja zmiennoprzecinkowa zawężająca

Zmiana kodu



Rozszerzenia kodu: znakowe (integer) i zerowe (natural)



Negacja bitów

Arytmetyka stałoprzecinkowa

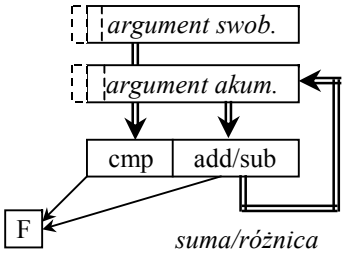
interpretacja:
– działania podstawowe: 1-pozycyjne w podstawie $2^{op\text{-}size}$ (2^8, 2^{16}, 2^{32}, 2^{64})
– działania rozszerzonej precyzji: wielopozycyjne według reguł
arytmetyki stałoprzecinkowej (naturalnej lub uzupełnieniowej)

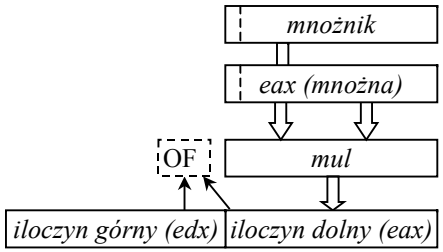
mnożenie: iloczyn liczb 1-cyfrowych jest 2-cyfrowy

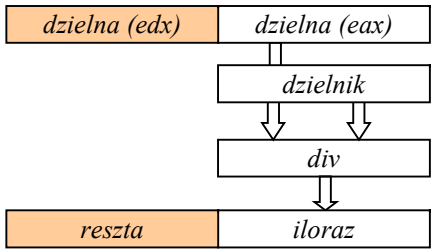
[i]mul arg ; (**edx:eax**):=(**eax**)*__arg__

dzielenie: iloraz liczby 2-cyfrowej przez 1-cyfrową
może być 1-cyfrowy lub 2-cyfrowy – konieczna kontrola

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







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

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)**
	j*war* 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)**
	j*not_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)

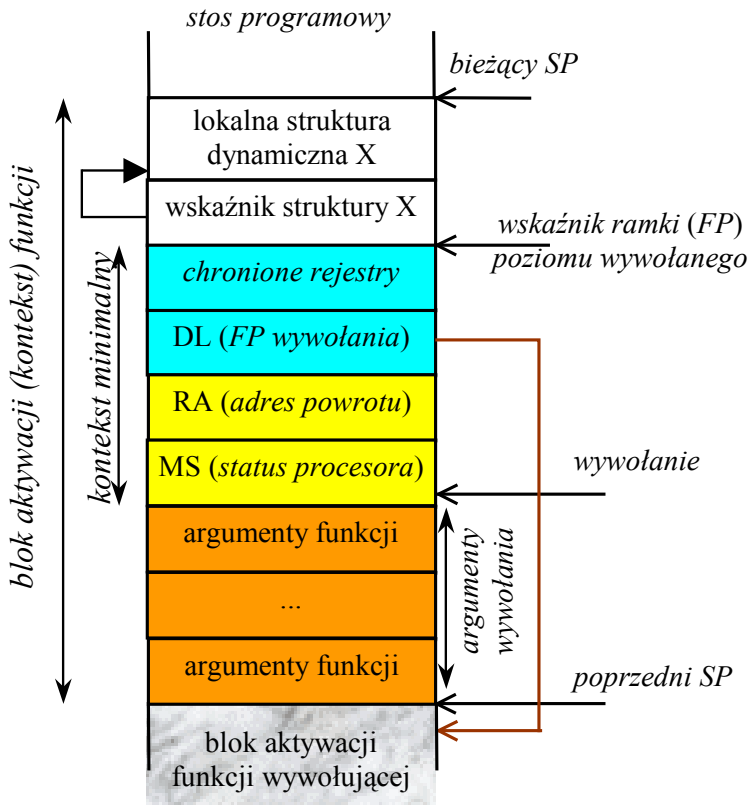
– powtarzaj dopóki: **repeat** *polecenie* **until** *warunek*=TRUE

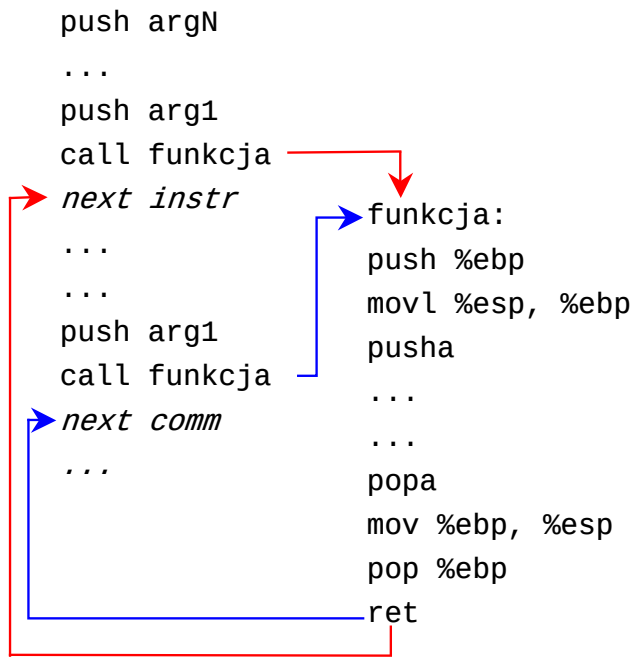
	Intel (MASM)		**AT&T/Linux/UNIX**	Komentarz
start:	*polecenie* (*A*, *B*)	start:	*polecenie* (*A*, *B*)	
	mov eax, A		**movl** $A, %eax	
	cmp eax, B		**cmpl** $B, %eax	
	jgt start		**jgt** start	

– 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:		

Funkcja – kontekst i blok aktywacji





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*)
	`j`*`cc`* `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*)
	`j`*`cc`* `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: algorytm Euklidesa przez wskaźniki

(etykieta)	Rozkaz	Komentarz
gcd:	push %ebp	# GCD(a,b)=GCD(b, a mod b)
	movl %esp, %ebp	
	movl 8(%ebp), %eax	# adres argumentu „a” w rejestrze eax
	movl 12(%ebp), %ebx	# adres argumentu „b” w rejestrze ebx
	movl (%eax), %edx	# wielokrotne odejmowanie “b”
pocz:	subl (%ebx), %edx	# dopóki różnica dodatnia
	ja rem	
	xchg %eax, %ebx	# różnica ujemna, korekcja reszty
	addl (%ebx), %edx	# wymiana wskaźników
	movl %edx, (%eax)	
	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	

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: 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: 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
```

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	

Działania zmiennoprzecinkowe

rejestr sterujący FPCR (ang. *Floating-Point Control Register*)

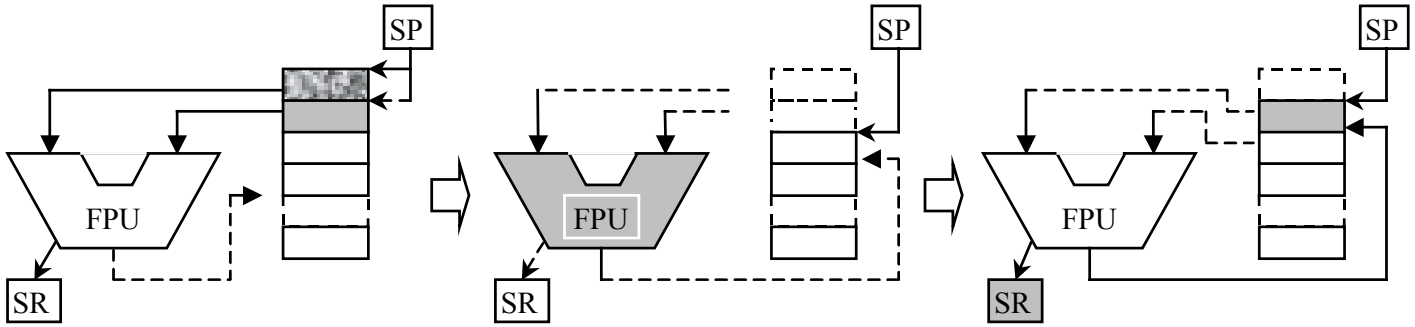
- ustalenie formatu argumentów
- ustalanie sposobu działania jednostki zmiennoprzecinkowej
  - tryb zaokrąglania
  - obsługa kodów specjalnych (NaN, nieskończoności)
  - obsługa liczb zdenormalizowanych
  - reakcja na wyjątki

rejestr stanu FPSR (ang. *Floating-Point Status Register*)

- odwzorowanie stanu
  - flagi wyjątków
  - sygnalizacja niedozwolonych działań
- maskowanie aktualizacji stanu

Działania zmiennoprzecinkowe IA-32 (1)

architektura stosowa



f*mnemo***[p]** – argumenty domniemane na szczycie stosu
f*mnemo***[p]** ST(*i*) – argumenty: wewnątrz stosu ST(*i*) i domniemany (ST(0))
f*mnemo***[p]** ST(*i*), ST(*j*) – argumenty specyfikowane wewnątrz stosu

f*mnemo* – wykonaj działanie
f*mnemo***p** – wykonaj działanie i zdejmij argument ze stosu (zmniejsz wskaźnik)

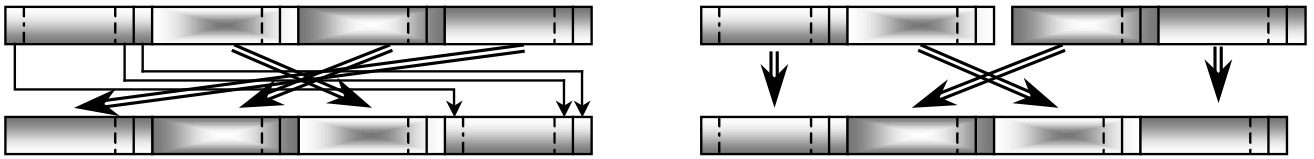
Działania wektorowe IA-32 (MMX/SSE#)

działania wektorowe stałoprzecinkowe

- równolegle na wszystkich polach rejestrów **mm#**
- arytmetyka nasyceniowa lub modularna
- przestawianie pól

działania wektorowe zmiennoprzecinkowe

- równolegle na wszystkich polach rejestrów **xmm#**
- uproszczona arytmetyka zmiennoprzecinkowa
- przestawianie pól



Przestawienie bajtów (**swap**) oraz przeplot (MMX/SSE#)

Organizacja komputera

1. Tryb (schemat) przetwarzania informacji
   - **w rytmie napływu instrukcji** (poleceń) – (ang. *control flow*)
   - **w rytmie napływu danych** – (ang. *data flow*)
2. Organizacja przetwarzania informacji
   - **przetwarzanie potokowe** – (ang. *pipeline*)
   - **przetwarzanie współbieżne** – (ang. *execution unit*)
3. Sterowanie – struktura i układ (**mikroprogram**)
4. Struktura bloków funkcjonalnych
   - **konstrukcja** układów wykonawczych
   - **przepływ danych** pomiędzy blokami funkcjonalnymi (ang. *data path*)
5. **Hierarchia pamięci** – organizacja składowania danych:
   - rejestry procesora (ang. *register file*)
   - bufory informacji (ang. *cache*)
   - **pamięć operacyjna** (główna) (ang. *operational/main memory*)
   - pamięć masowa (wtórna) (ang. *storage*)

Architektura komputera

(ang. *Instruction Set Architecture, ISA*)

1. **Działania elementarne**
   - specyfikacja działań – lista rozkazów
   - sposób wykonania działań – wymagana zgodność rozmiaru
   - syndromy (istotne cechy) wyniku – tworzenie i użycie
2. **Specyfikacja danych (**informacji)
   - **atrybuty** danej – lokacja (adres), rozmiar, typ
     - lokacja (adres) – unikatowy wskaźnik danej
     - rozmiar – liczba jednostek (bajtów/słów)
     - typ – rozmiar jednostki danych (bajt/słowo)
   - **mechanizmy adresowania** i struktury danych
3. **Sterowanie** (przebiegiem programu)
   - kolejność działań – domniemana i wymuszona
   - warunki i decyzje – przesłanki zmiany kolejności działań
   - przekazywanie sterowania – tymczasowe i trwałe

Architektura komputera - dziedziny

Dziedzina (interfejs) kompilatora (struktury danych i algorytmy)

- działania i ich cechy
- typy danych – umowy (ASCII, UNICODE, U2, IEEE 754-2008)
- struktury danych i adresowanie – tryby adresowania
- sterowanie
- procedury i funkcje
- dystrybucja wątków

Dziedzina (interfejs) systemu operacyjnego (zarządzanie i ochrona)

- zarządzanie procesami
- zarządzanie pamięcią
  - ochrona danych
  - implementacja pamięci wirtualnej
- przerwania i obsługa zdarzeń
  - obsługa wyjątków
  - obsługa wejścia i wyjścia

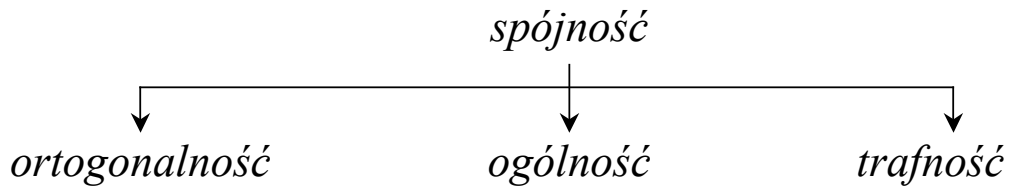
Spójność architektury

– spójność (ang. *consistency*) – jednolite relacje cech

fragment $\Rightarrow$ *całość*

$d \in OP,\ s \in ARG,\ (d,s) \in ISA \subset OP \times ARG \Rightarrow \forall f \in OP,\ \forall x \in ARG:\ (f,x) \in ISA$

add R1, R7, R15	$\Rightarrow$	**add R1, R1, R10**	$\Rightarrow$	*add Rx, Ry, Rz*
$\Downarrow$		$\Downarrow$		$\Downarrow$
sub R1, R7, R15	$\Rightarrow$	*sub R1, R1, R10*	$\Rightarrow$	*sub Rx, Ry, Rz*
$\Downarrow$		$\Downarrow$		$\Downarrow$
mul R1, R7, R15	$\Rightarrow$		$\Rightarrow$	*mul Rx, Ry, Rz*
$\Downarrow$		$\Downarrow$		$\Downarrow$
div R1, R7, R15	$\Rightarrow$		$\Rightarrow$	*div Rx, Ry, Rz*

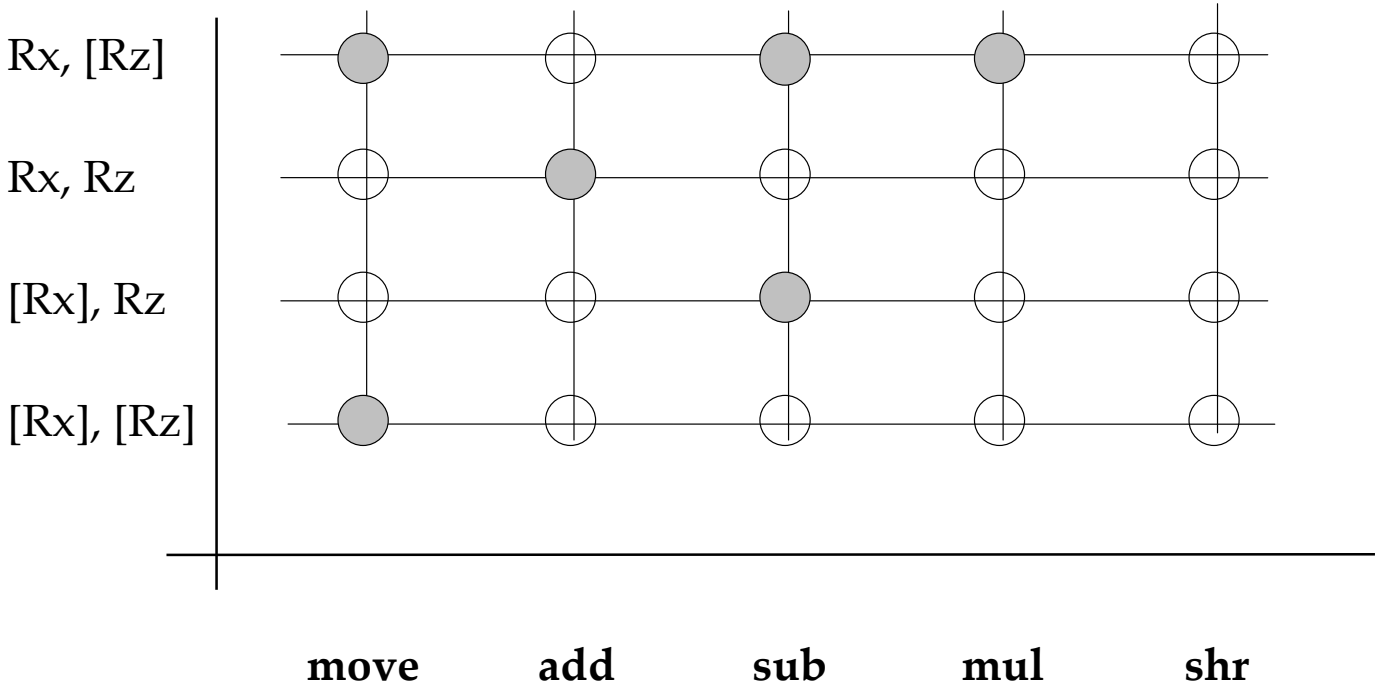


Aspekty przejrzystej architektury – ortogonalność

– ortogonalność (*orthogonality*)

niezależność funkcji i operandów

$$\exists d \in OP: \forall x \in ARG, (d,x) \in ISA \quad \& \quad \exists s \in ARG: \forall f \in OP, (f,s) \in ISA$$
$$\Rightarrow \forall f \in OP, \forall x \in ARG: (f,x) \in ISA \subset OP \times ARG$$



Aspekty przejrzystej architektury – trafność

– trafność (ang. *propriety*)

- ***przeźroczystość*** (ang. *transparency*) – *niezależność od implementacji*
- ***oszczędność*** (ang. *parsimony*) – *minimalizacja repertuaru*

$$\Rightarrow p,q \in OP \;\&\; a=\Phi(p,q),\ b=\Gamma(p,q),\ c=\theta(p,q)\ldots \Rightarrow a,b,c \notin OP$$

add Ra, Rb, Rc **:**	sub Rx, Rx, Rx,	; Rx ← 0 (= Rx – Rx)
	sub Rb, Rx, Rb,	; Rb ← 0 – Rb
	sub Ra, Rb, Rc	; Rc ← Ra – (–Rb)
and Ra, Rb, Rc **:**	nor Ra, Ra, Ra,	; Ra ← ~ Ra
	nor Rb, Rb, Rb,	; Rb ← ~ Rb
	nor Ra, Rb, Rc	; Rc ← ~ (~ Ra ∨ ~ Rb)

⇓

⇒ ISA:	**sub Ra, Rb, Rc**	; Rc ← Ra – Rb
⇒ ISA:	**nor Ra, Rb, Rc**	; Rc ← ~(Ra ∨ Rb)

Aspekty przejrzystej architektury – ogólność

– ogólność (ang. *generality*)

- ***kompletność*** (ang. *completeness*) – *wszystkie możliwe kombinacje*
  sprzeczna z oszczędnością

$$\text{op}\in\text{F},\ \text{arg}\in\text{A}\ \&\ (\text{op},\text{arg})\in\text{ISA}\ \Rightarrow\ \forall\,\text{f}\in\text{F},\ \forall\,\text{x}\in\text{A}:(\text{f},\text{x})\in\text{ISA}$$

- ***otwartość*** (*open-endness*) – *łatwość rozszerzenia ISA,*
  dostępna przestrzeń dla przyszłych rozszerzeń

  - regularna struktura kodu
    - wolne miejsca w przestrzeni kodowej
  - jednolite kodowanie argumentów
    - uniezależnienie działań od rozmiaru argumentów
  - konsekwentny dobór funkcji
    - unikanie ograniczeń implementacyjnych i technologicznych

Przykłady naruszenia spójności – IA-32

– brak ortogonalności

- dedykowane rejestry
- dodawanie/odejmowanie – specyfikowane argumenty rejestrowe, ale mnożenie i dzielenie – argumenty *domniemane* (edx, eax)
- licznik powtórzeń dla organizacji pętli – *domniemany*, tylko ecx

– brak trafności

- ***brak przeźroczystości***: rozkazy zmiennoprzecinkowe
- ***brak oszczędności***: dodawanie/odejmowanie bez przeniesienia, mało użyteczne rozkazy jednocyfrowej arytmetyki dziesiętnej

– brak ogólności

- rozkazy logiczne **and, or, xor** modyfikują ***flagi*** (kody warunkowe), ale rozkaz logiczny **not** nie modyfikuje ***flag***
- rozkaz uzupełniania **neg** nie uwzględnia przeniesienia, co ogranicza jego zastosowanie
- sztuczne rozszerzanie przestrzeni kodowej dla FPU, MMX, SSE#