Architektura komputerów

sem. zimowy 2024/25

cz. 2

Tomasz Dziubich

POLITECHNIKA GDAŃSKA

Kodowanie programów na poziomie rozkazów procesora (1)

- Wykonanie programu przez procesor wymaga uprzedniego załadowania do pamięci danych i rozkazów, zakodowanych w formie ciągów zerojedynkowych, zrozumiałych przez procesor.
- Współczesne kompilatory języków programowania generują takie ciągi w sposób automatyczny na podstawie kodu źródłowego programu.
- Niekiedy jednak celowe jest precyzyjne zakodowanie programu lub fragmentu programu za pomocą pojedynczych rozkazów procesora.

POLITECHNIKA GDAŃSKA

Kodowanie programów na poziomie rozkazów procesora (2)

- Dokumentacja techniczna procesora zawiera zazwyczaj tablice ciągów zerojedynkowych przypisanych poszczególnym operacjom (rozkazom procesora).
- Kodowanie na poziomie zer i jedynek, aczkolwiek możliwe, byłoby bardzo żmudne i podatne na pomyłki.
- Z tego powodu opracowano programy, nazywane *asemblerami*, które na podstawie skrótu literowego (tzw. mnemonika) opisującego czynności rozkazu dokonują zamiany tego skrótu na odpowiedni ciąg zer i jedynek.

POLITECHNIKA GDAŃSKA

Kodowanie programów na poziomie rozkazów procesora (3)

- Asemblery udostępniają wiele innych udogodnień, jak np. możliwość zapisu liczb w systemach o podstawie 2, 8, 10, 16 czy też automatyczną zamianę tekstów znakowych na ciągi bajtów zawierające kody ASCII poszczególnych liter.

POLITECHNIKA GDAŃSKA

Struktura programu w asemblerze

```
.686          ; nagłówek programu, dyrektywy
.model flat   ; dyrektywa nieużywana w trybie 64-bitowym
_  _  _  _  _  _ _  _  _ _  _  _ _  _  _
.data? ; początek danych (niezainicjalizowanych)
; opisy danych niezainicjalizowanych
_  _  _  _  _  _ _  _  _ _  _  _ _  _  _
.data  ; początek danych (zainicjalizowanych)
; opisy danych zainicjalizowanych
_  _  _  _  _  _ _  _  _ _  _  _ _  _  _
_  _  _  _  _  _ _  _  _ _  _  _ _  _  _
.code
; rozkazy (instrukcje) programu
_  _  _  _  _  _ _  _  _ _  _  _ _  _  _
_  _  _  _  _  _ _  _  _ _  _  _ _  _  _
END
```

POLITECHNIKA GDAŃSKA

Wiersze źródłowe w asemblerze (1)

- W programie asemblerowym niektóre obszary pamięci opatrywane są nazwami:
  - jeśli nazwa odnosi się do obszaru zawierającego instrukcję (rozkaz) programu, to nazwa taka stanowi *etykietę*,
  - jeśli obszar zawiera zmienną (daną), to nazwa obszaru stanowi *nazwę zmiennej*.
- Nazwę w sensie asemblera tworzy ciąg liter, cyfr i znaków ? (znak zapytania), @ (symbol at), _ (znak podkreślenia), $ (znak dolara); nazwa nie może zaczynać się od cyfry.

POLITECHNIKA GDAŃSKA

Wiersze źródłowe w asemblerze (2)

- Etykietę, wraz ze znakiem **:** (dwukropka), umieszcza się przed rozkazem, np.

powtorz: mov dl, [ecx]

- Taka konstrukcja oznacza, że obszar kilku bajtów pamięci, w których przechowywany jest kod podanego rozkazu **mov dl, [ecx]** ma swoją unikatową nazwę, czyli *etykietę*.
- Opcjonalnie może jeszcze wystąpić komentarz poprzedzony znakiem średnika **;**
- Program w asemblerze pozbawiony komentarzy może być nieczytelny nawet dla autora.

POLITECHNIKA GDAŃSKA

Deklaracje zmiennych w asemblerze (1)

- Zmienne w programie deklaruje się za pomocą dyrektyw:
  - DB – definiowanie bajtu (8 bitów),
  - DW – definiowanie słowa (16 bitów),
  - DD – definiowanie słowa podwójnej długości (32 bity),
  - DQ – definiowanie słowa poczwórnej długości (64 bity)
- Istnieje też kilka podobnych dyrektyw, które pozwalają na bardziej precyzyjny opis typu danych (zob. instrukcję do ćw. laboratoryjnego nr 4).

POLITECHNIKA GDAŃSKA

Deklaracje zmiennych w asemblerze (2)

- Po lewej stronie dyrektywy podaje się nazwę zmiennej, a po prawej wartość początkową, np.

  elem_sygn DD –765407

- W jednym wierszu można podać kilka wartości (litera **H** oznacza liczbę zapisaną w systemie szesnastkowym)

  alfa DW 4567H, 5678H, 6789H

  Powyższy wiersz można interpretować jako deklarację tablicy zawierającej trzy liczby 16-bitowe.

POLITECHNIKA GDAŃSKA

Deklaracje zmiennych w asemblerze (3)

- Przy zapisie liczb szesnastkowych konieczne jest wprowadzenie dodatkowego zera z lewej strony, jeśli pierwszą cyfrą liczby jest A, B,…, F, np. zamiast F3H należy napisać **0F3H**
- Jeśli tablica zawiera identyczne elementy, to stosuje się operator powtarzania **dup**, np. poniższy wiersz stanowi opis tablicy zawierającej 512 elementów o rozmiarze jednego bajtu, przy czym inicjalnie każdemu elementowi zostaje przypisana wartość –1

  wyniki DB 512 dup (–1)

POLITECHNIKA GDAŃSKA

Deklaracje zmiennych w asemblerze (4)

- Liczby dziesiętne całkowite poprzedzone znakiem plus + (lub bez znaku) zamieniane są na liczby w naturalnym kodzie binarnym (czyli zamieniane są na liczby binarne bez znaku).
- Liczby dziesiętne całkowite poprzedzone znakiem minus zamieniane są na liczby w kodzie U2.
- W szczególności z powyższych reguł wynika, że wiersze

```
DB      –128
DB      +128
```

generują takie same kody binarne 10000000

POLITECHNIKA GDAŃSKA

Deklaracje zmiennych w asemblerze (5)

- Znak zapytania (?) w polu argumentu oznacza, że wartość początkowa zmiennej jest nieokreślona, np.

```
linia7a     DQ      ?
znak        DB      ?
```

  Powyższe wiersze stanowią przykłady opisu danych niezainicjalizowanych.

- W polu operandu dyrektywy DB mogą występować także łańcuchy znaków, np.

```
uczelnia    DB   'Politechnika Gdańska', 0DH, 0AH
            DB   'Wydział ETI' , 0DH, 0AH
```

- Nazwę zmiennej przed dyrektywą można pominąć

POLITECHNIKA GDAŃSKA

Deklaracje zmiennych w asemblerze (6)

- Liczby z kropką oddzielającą część ułamkową mogą występować jako operandy dyrektyw DD, DQ, DT – liczby takie są przekształcane na postać zmiennoprzecinkową binarną, np.:

```
c_Eulera  DT          0.577215 ; stała Eulera
__ __ __ __ __ __ __ __ __ __ __ __ __
      fld    c_Eulera  ; załadowanie stałej Eulera
                       ; na stos koprocesora
```

Deklaracje stałych w asemblerze

- Deklaracje stałych umożliwiają przypisanie nazw symbolicznych do określonych wartości, np.

```
num1   EQU      18H  ; wartość max
```

POLITECHNIKA GDAŃSKA

Kodowanie rozkazów (1)

- Typowe rozkazy procesora mają jeden lub dwa argumenty, np. rozkaz przesłania

  mov edi, pomiar

  powoduje wpisanie do rejestru **EDI** zawartości zmiennej **pomiar**
- Operandy rozkazów mają postać wyrażeń różnych typów: są to często rejestry, liczby, ciągi znaków, wyrażenia opisujące położenie lokacji pamięci i wyrażenia adresowe.
- W asemblerze przyjęto, że nazwa zmiennej występująca w polu operandu rozkazu (podobnie jak nazwa rejestru) oznacza, że działanie ma być wykonane na zawartości tej zmiennej.

POLITECHNIKA GDAŃSKA

Kodowanie rozkazów (2)

- W trakcie asemblacji nazwie każdej zmiennej przypisywana jest wartość równa odległości pierwszego bajtu tej zmiennej (liczonej w bajtach) od początku obszaru danych.
- Warto tu podkreślić, że omawiana wartość nie jest wartością zmiennej, ale jest przypisywana nazwie zmiennej i określa położenie zmiennej w pamięci.

POLITECHNIKA GDAŃSKA

Rozmiary operandów (1)

- Jeśli instrukcja wymaga podania dwóch operandów, to prawie zawsze muszą być one jednakowej długości, np.:

```
sub  ecx, edi ; obliczenie ECX ← ECX – EDI
add   eax, dx ; błąd !!! — dodawanie
              ; zawartości rejestru 16-bitowego
              ; do 32-bitowego
```

POLITECHNIKA GDAŃSKA

Rozmiary operandów (2)

- Wyjątkowo, jeśli jeden z operandów znajduje się w pamięci, to możliwa jest doraźna zmiana rozmiaru (typu) zmiennej za pomocą operatora PTR, np.

```
blok_sys        dd   ?     ; zmienna 32-bitowa
— — — — — — — — —
    mov     cx,  word PTR blok_sys
    mov     dx,  word PTR blok_sys+2
```

- W podanym przykładzie do rejestru CX zostanie wpisana młodsza część 32-bitowej zmiennej blok_sys, a rejestru DX starsza część tej zmiennej.
- W przypadku pominięcia operatora PTR sygnalizowany byłby błąd asemblacji wynikający z niejednakowej długości obu operandów.

POLITECHNIKA GDAŃSKA

Operator PTR (1)

- Operator PTR używany jest w wyrażeniach adresowych do ścisłego określania atrybutów symbolu występującego w wyrażeniu adresowym.
- Po prawej stronie operatora występuje wyrażenie adresowe, a po lewej atrybut spośród następujących: byte (8 bitów), word (16 bitów), dword (32 bity), fword (48 bitów), qword (64 bity), tword (80 bitów).
- Operator PTR nie dokonuje konwersji typów — działa on tylko w trakcie asemblacji programu i nie może zastępować rozkazów wykonywanych w trakcie realizacji programu.

POLITECHNIKA GDAŃSKA

Operator PTR (2)

- Z podanych powodów poniższy rozkaz

  mov ebx, dword PTR dh; błąd !

  jest błędny
- Konwersję taką można zrealizować za pomocą dalej opisanych rozkazów MOVSX lub MOVZX

Operator PTR (3)

- Operator PTR stosuje się także w przypadkach, gdy wyrażenie adresowe nie pozwala na jednoznaczne przetłumaczenie rozkazu, np. poniższy rozkaz stanowi tzw. odwołanie anonimowe (nie zawiera nazwy zmiennej)

```
mov     [ebx], 65
```

- Operandy rozkazu nie precyzują czy liczba 65 ma zostać zapisana w pamięci jako liczba 8-bitowa, 16-bitowa czy 32-bitowa — zamiast podanego wyżej rozkazu programista musi wyraźnie określić rozmiar liczby podając jeden z trzech poniższych rozkazów

```
mov     byte    PTR [ebx], 65
mov     word    PTR [ebx], 65
mov     dword   PTR [ebx], 65
```

POLITECHNIKA GDAŃSKA

Rozkazy MOVZX i MOVSX (1)

- Reguła dotycząca jednakowej długości operandów nie obowiązuje w odniesieniu do rozkazów MOVSX i MOVZX
- Rozkazy te powodują przepisanie zawartości 8- lub 16-bitowego rejestru (lub zawartości lokacji pamięci) do rejestru 16- lub 32-bitowego.
- W przypadku rozkazu MOVZX brakujące bity w rejestrze docelowym uzupełniane są zerami, zaś w przypadku MOVSX bity te są wypełniane przez wielokrotnie powielony bit znaku kopiowanego rejestru.

POLITECHNIKA GDAŃSKA

Rozkazy MOVZX i MOVSX (2)

- Przykłady

```
liczba_proc   db    145  ; zmienna 8-bitowa
    —   —   —   —   —   —   —   —   —

        movzx     edx, liczba_proc
        movsx     edx, bh
```

- Rozkaz MOVSX stosuje się do kopiowania liczb ze znakiem — po zwiększeniu ilości bitów liczby, jej wartość pozostaje niezmieniona.

POLITECHNIKA GDAŃSKA

Operator OFFSET

- Położenie zmiennej (lub rozkazu) w pamięci określa się poprzez podanie odległości, liczonej we bajtach, zmiennej od początku obszaru danych — odległość ta nazywana jest *przesunięciem* lub *offsetem* i można ją wyznaczyć za pomocą operatora OFFSET.
- Przykładowo, adres zmiennej moc_czynna, która została zdefiniowana w sekcji danych

  ```
  moc_czynna        dw                800
  ```

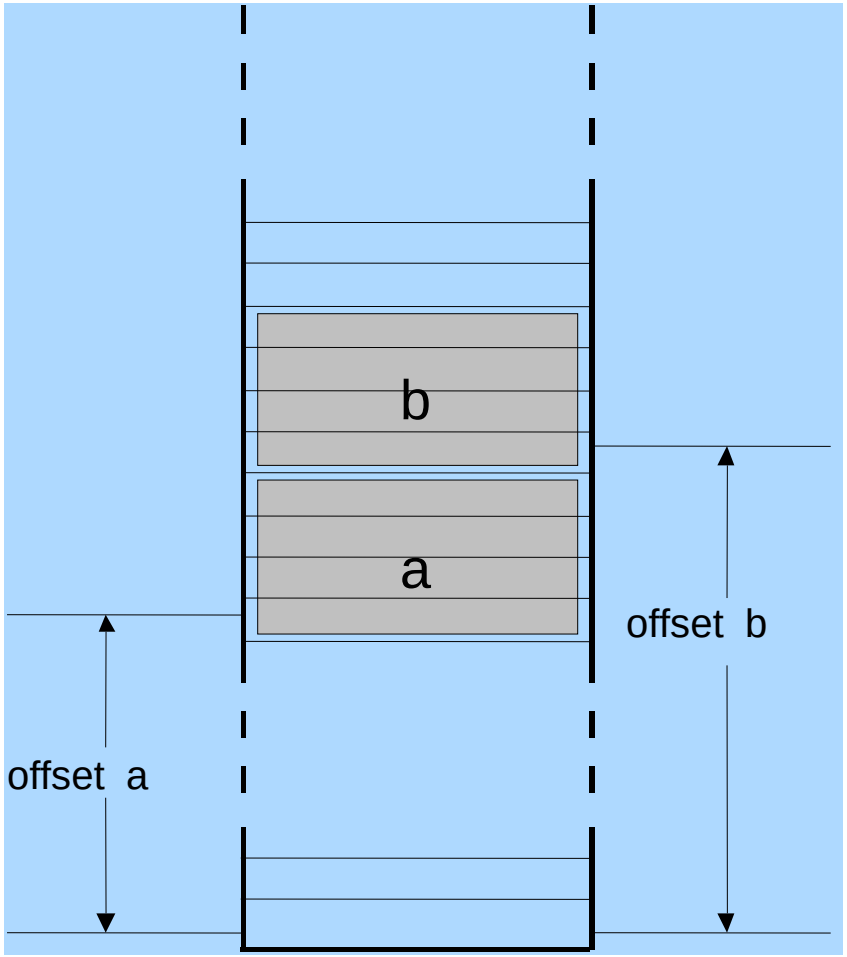
  można wpisać do rejestru EBX za pomocą rozkazu

  ```
  mov   ebx, OFFSET moc_czynna
  ```

POLITECHNIKA GDAŃSKA

Adresowanie zmiennych (1)

- Położenie zmiennych w pamięci określane jest poprzez podanie położenia bajtu o najniższym adresie.
- Adres tego bajtu. nazywany jest także *przesunięciem* lub *offsetem* zmiennej.



POLITECHNIKA GDAŃSKA

Rozkaz LEA (1)

- Położenie pewnej lokacji pamięci względem początku obszaru danych można także wyznaczyć za pomocą rozkazu LEA, który wyznacza adres efektywny rozkazu, czyli położenie lokacji pamięci (względem początku danych), na której zostanie wykonana operacja.
- Podany poprzednio przykład można przedstawić w innej postaci

```
lea     ebx, moc_czynna
```

POLITECHNIKA GDAŃSKA

Rozkaz LEA (2)

- Rozkaz LEA wyznacza adres efektywny w trakcie wykonywania programu, podczas gdy wartość operatora OFFSET obliczana jest w trakcie translacji (asemblacji) programu.
- Wartość offsetu wyznaczana w trakcie asemblacji może ulec zmianie podczas konsolidacji i ładowania programu do pamięci; dokonują tego konsolidator i system operacyjny
- Należy zwrócić uwagę, że rozkaz LEA nigdy nie odczytuje zawartości lokacji pamięci.

POLITECHNIKA GDAŃSKA

Rozkaz LEA (3)

- Rozkaz LEA używany jest także w obliczeniach nie zawsze związanych z wyznaczeniem adresu zmiennej, np. można go zastosować do obliczenia sumy zawartości rejestrów czy mnożenia.
- Takie konstrukcje trzeba jednak stosować ostrożnie, ponieważ w przypadku rozkazu LEA ewentualny nadmiar nie jest sygnalizowany.

POLITECHNIKA GDAŃSKA

Rozkaz LEA (4)

- Obliczenia sumy zawartości rejestrów EBX i ECX:

```
lea     eax, [ebx + ecx]
```

zamiast tradycyjnego rozwiązania

```
mov     eax, ebx
add     eax, ecx
```

- Mnożenie liczby w EAX przez 5

```
lea     eax, [eax + eax*4]
```

zamiast tradycyjnego rozwiązania

```
mov     ebx, 5
mul     ebx
```

POLITECHNIKA GDAŃSKA

Rozkazy sterujące (1)

- Wykonywanie rozkazów pobieranych z pamięci w naturalnej kolejności nie pozwala zmieniać sposobu obliczeń w zależności o wartości uzyskiwanych wyników pośrednich.
- Potrzebne są więc specjalne rozkazy (instrukcje), które w zależności od własności uzyskanego wyniku (np. czy jest ujemny) zmienią porządek wykonywania rozkazów.
- Zmiana porządku realizowana jest poprzez zwiększenie lub zmniejszenie zawartości wskaźnika instrukcji (rejestru EIP).

POLITECHNIKA GDAŃSKA

Rozkazy sterujące (2)

- Do tego celu używane są rozkazy sterujące, nazywane zazwyczaj *rozkazami skoku*, których zadaniem jest sprawdzenie pewnego warunku, np. czy znacznik ZF w rejestrze stanu procesora zawiera wartość 1 i odpowiednie pokierowanie dalszym wykonywaniem programu.
- Jeśli warunek testowany przez rozkaz skoku jest spełniony, to procesor zmienia naturalny porządek wykonywania rozkazów, jeśli warunek jest nie spełniony, to procesor wykonuje dalej rozkazy w niezmienionym porządku (po kolei, tak jak są umieszczone w pamięci głównej).

POLITECHNIKA GDAŃSKA

Rozkazy sterujące (3)

- Istnieje wiele rozkazów sterujących (skoków) — dla każdego rozkazu sterującego zdefiniowany jest pewien charakterystyczny warunek.
- Niektóre rozkazy sterujące testują pojedyncze znaczniki w rejestrze stanu procesora, inne obliczają wartości wyrażeń logicznych, w których występują zawartości kilku znaczników.
- Każdy rozkaz sterujący ma przypisany skrót literowy (mnemonik), który składa się z litery **J** (skrót od ang. *jump* – skok) i dalszych liter skojarzonych z nazwą testowanego znacznika (np. **JZ** — skrót od *jump if zero*) lub typem operacji porównania.

Rozkazy używane do porównywania liczb bez znaku

Mnemonik		Interpretacja (zastosowanie)	Testowany warunek
typowy	symbole równo-ważne		
JA		skocz, gdy większy	CF = 0 i ZF = 0
	JNBE	skocz, gdy niemniejszy i nierówny	
JAE		skocz, gdy większy lub równy	CF = 0
	JNB	skocz, gdy niemniejszy	
	JNC	skocz, gdy nie było przeniesienia (nadmiaru)	
JB		skocz, gdy mniejszy	CF = 1
	JNAE	skocz, gdy niewiększy i nierówny	
	JC	skocz, gdy było przeniesienie (nadmiar)	
JBE		skocz, gdy mniejszy lub równy	CF = 1 lub ZF = 1
	JNA	skocz, gdy niewiększy	

Rozkazy sterujące (4)

- W języku maszynowym rozkaz skoku zajmuje zazwyczaj dwa bajty (w niektórych przypadkach asembler lub kompilator musi jednak zastosować rozkazy 6-bajtowe): pierwszy bajt opisuje czynność wykonywaną przez rozkaz, drugi bajt (pole adresowe) zawiera liczbę, która określa zakres skoku.

jz	01110100	pole adresowe	warunek spełniony gdy ZF = 1
jnz	01110101	pole adresowe	warunek spełniony gdy ZF = 0
ja	01110111	pole adresowe	warunek spełniony gdy (CF = 0) i (ZF = 0)

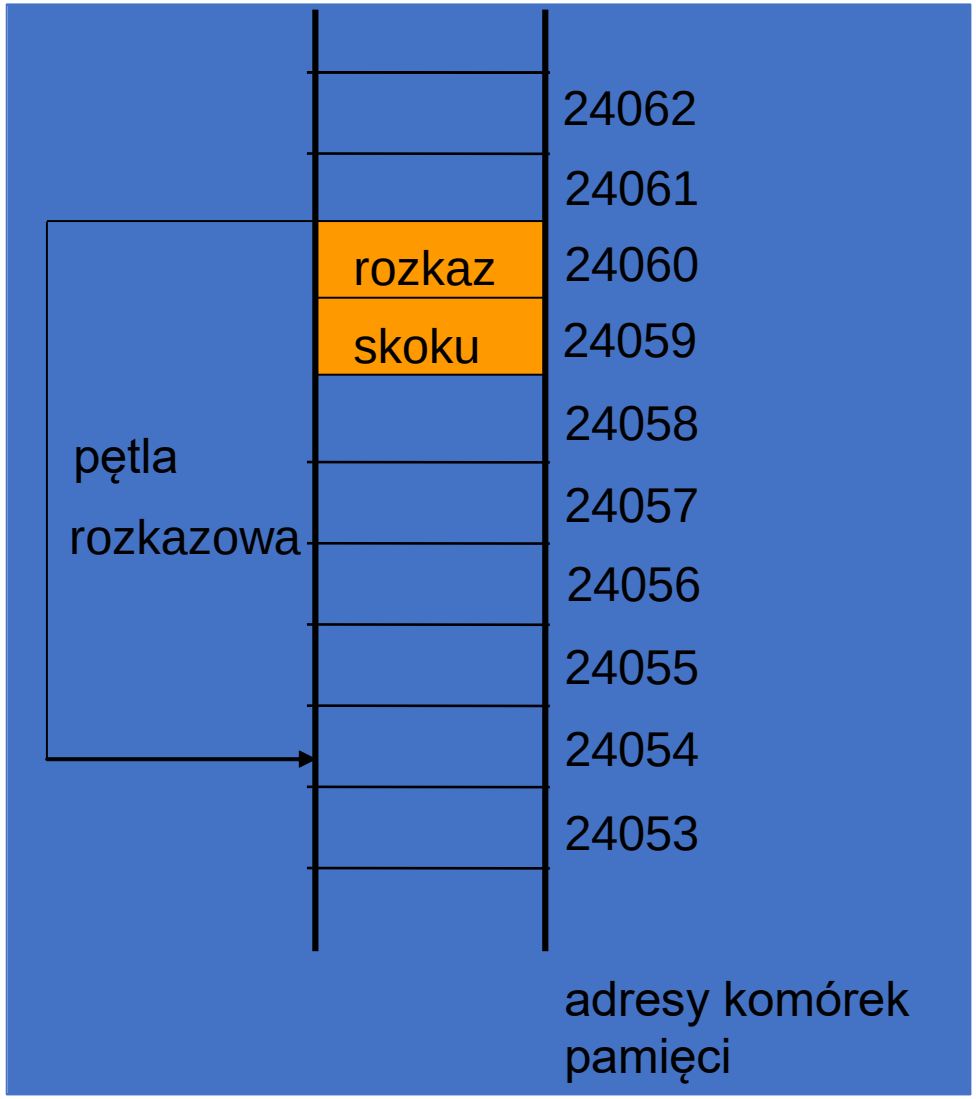
POLITECHNIKA GDAŃSKA

Rozkazy sterujące (5)

- Jeśli testowany warunek jest spełniony, to liczba (dodatnia lub ujemna) umieszczona w polu adresowym jest dodawana do wskaźnika instrukcji EIP, ponadto wskaźnik instrukcji EIP jest zwiększany o liczbę bajtów zajmowanych przez sam rozkaz skoku (zwykle o 2).
- Jeśli warunek nie jest spełniony, to wskaźnik instrukcji EIP jest zwiększany o liczbę bajtów zajmowanych przez rozkaz skoku (zwykle o 2) — pole adresowe jest ignorowane.

Pętle rozkazowe

- Zmniejszenie wskaźnika instrukcji EIP powoduje, że procesor rozpocznie ponownie wykonywanie rozkazów, które były wcześniej wykonywane — powstaje więc pętla rozkazowa.



POLITECHNIKA GDAŃSKA

Rozkazy sterujące a rejestr EIP

- 1. gdy testowany warunek jest spełniony:

  EIP ← EIP + <*liczba bajtów aktualnie wykonywanego rozkazu*>
  + <*zawartość pola adresowego rozkazu*>

- 2. gdy testowany warunek nie jest spełniony

  EIP ← EIP + <*liczba bajtów aktualnie wykonywanego rozkazu*>

POLITECHNIKA GDAŃSKA

Rozkazy sterujące bezwarunkowe (1)

- Rozkazy sterujące, zwane *bezwarunkowymi*, służą do zmiany porządku wykonywania instrukcji (nie wykonują one żadnego sprawdzenia, przyjmują, że testowany warunek jest zawsze spełniony).

- W architekturze x86 rozkazy tej grupy oznaczane są mnemonikiem **JMP**.

POLITECHNIKA GDAŃSKA

Rozkazy sterujące bezwarunkowe (2)

- W architekturze x86 dostępne są dwie odmiany rozkazów sterujących (skoków) bezwarunkowych:
  - *skoki bezpośrednie*, jeśli wartość wpisywana (albo dodawana) do rejestru EIP podana jest w polu adresowym rozkazu;
  - *skoki pośrednie*, jeśli wartość podana w polu adresowym wskazuje rejestr lub lokację pamięci, w której znajduje się nowa zawartość EIP — mechanizm ten pozwala m.in. na wykonanie skoku do lokacji pamięci, której adres zostanie obliczony dopiero w trakcie wykonywania programu.

POLITECHNIKA GDAŃSKA

Rozkazy sterujące bezwarunkowe (3)

- Przykład skoku bezpośredniego

```
sygnalizuj:
-  -  -  -  -  -  -  -  -  -  -  -  -  -
        jmp  sygnalizuj   ; skok bezpośredni
```

- Przykłady skoków pośrednich

1)

```
        jmp      ebx
```

2)

```
wybor     DD   OFFSET  kontynuacja
-  -  -  -  -  -  -  -  -  -  -  -  -
     jmp  dword  PTR  wybor
-  -  -  -  -  -  -  -  -  -  -  -  -
kontynuacja:
```

POLITECHNIKA GDAŃSKA

Tryby adresowania (1)

- Adres komórki pamięci, w której umieszczony jest rozkaz do wykonania określa wskaźnik instrukcji (nazywany także licznikiem rozkazów). Natomiast położenie argumentu, na którym zostanie wykonana operacja, może być określone w różny sposób, zależnie od zastosowanego w rozkazie trybu adresowania (ang. *addressing mode*).
- Tryb adresowania określa sposób, w jaki sposób wyznaczane jest położenie argumentu lub argumentów biorących udział w operacji. Argumenty mogą znajdować się w rejestrach lub w pamięci głównej (operacyjnej) komputera.

POLITECHNIKA GDAŃSKA

Tryby adresowania (2)

- Adres lokacji pamięci zawierającej argument, na którym zostanie wykonana operacja nosi nazwę *adresu efektywnego*.

- Jeśli operacja wykonywana jest na argumencie zajmującym 2, 4 lub więcej bajtów, to podaje się adres bajtu o najniższym adresie.

POLITECHNIKA GDAŃSKA

Tryby adresowania (3)

- W podanych dalej opisach trybów adresowania stosowane są oznaczenia symboliczne, odnoszące się do procesorów różnych typów.
- Oznaczenia te odbiegają od stosowanych w asemblerze dla procesorów x86, w szczególności:

  Rejestry oznaczono symbolami R1, R2, R3, . . .

  Zawartości rejestrów oznaczono przez Regs[R1], Regs[R2], Regs[R3], . . .

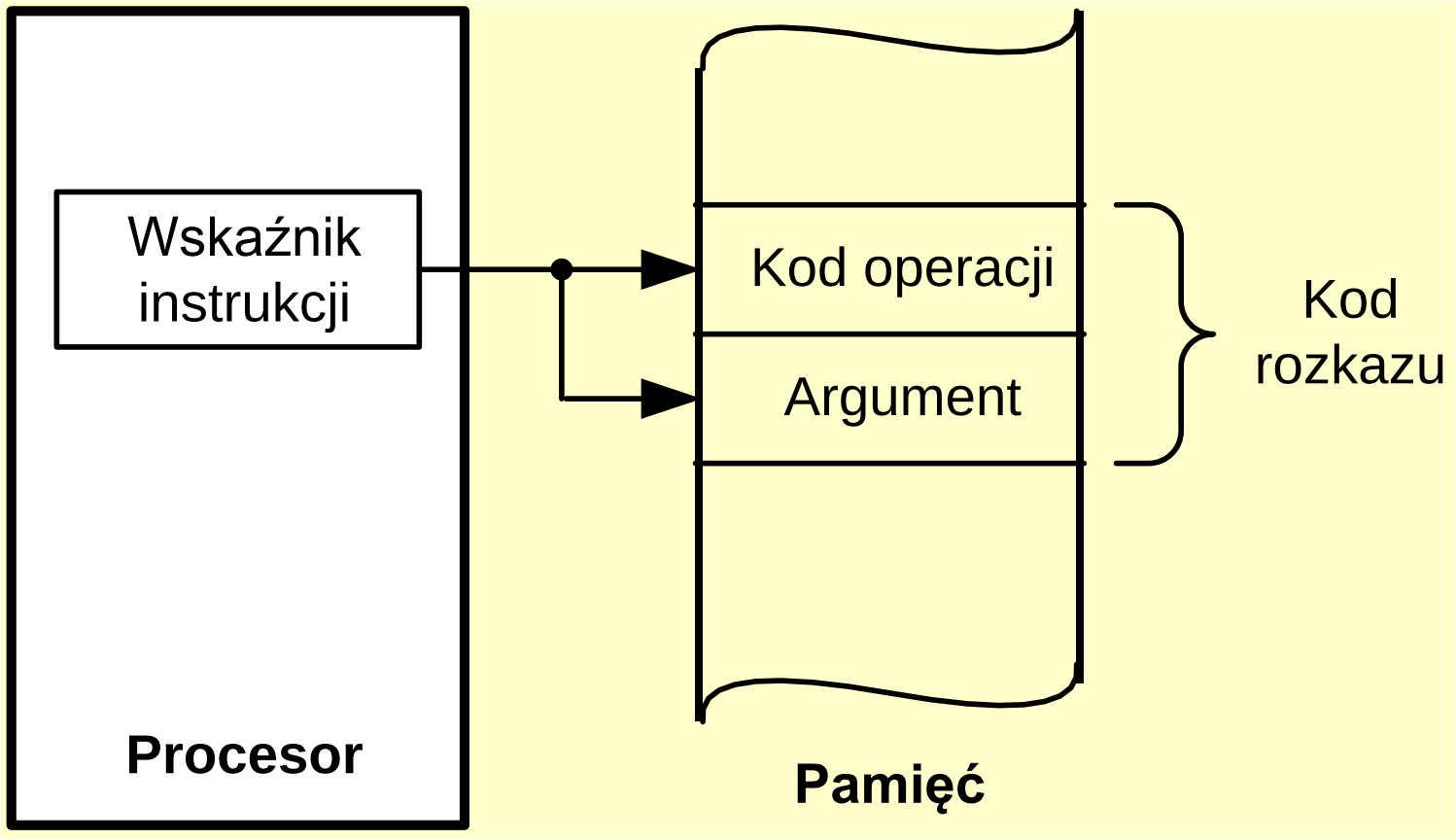
  Zawartości lokacji pamięci oznaczono przez Mem [], gdzie wartość podana w nawiasach kwadratowych stanowi adres lokacji.

POLITECHNIKA GDAŃSKA

Adresowanie natychmiastowe (1)

- ***Adresowanie natychmiastowe*** jest najprostszym sposobem adresowania — w tym przypadku argument stanowi fragment kodu rozkazu i jest umieszczony bezpośrednio za kodem operacji.
- Tak więc w trakcie pobierania rozkazu z pamięci jednocześnie pobierany jest *argument natychmiastowy* stanowiący fragment całego rozkazu.

Adresowanie natychmiastowe (2)



Adresowanie natychmiastowe (3)

- Przykład: dodawanie liczby 3 do rejestru

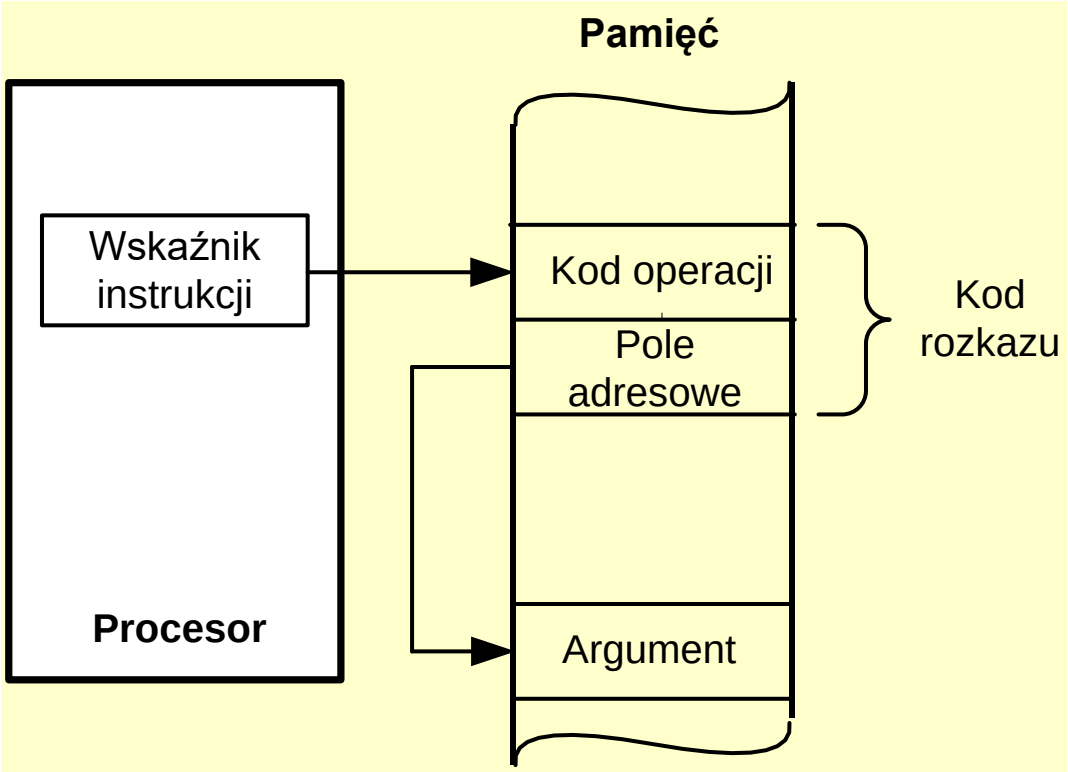
  zapis symboliczny i jego postać w asemblerze:

  ```
  Regs[R4] ← Regs[R4] + 3
  add  R4, #3
  ```

  podobny zapis w asemblerze Intel:

  ```
  add  ESI, 3
  ```

Adresowanie bezpośrednie (1)



- W przypadku ***adresowania bezpośredniego***, w rozkazie występuje kilkubajtowe (zazwyczaj 4-bajtowe) pole adresowe, w którym umieszczany jest adres argumentu.

Adresowanie bezpośrednie (2)

- Przykład: dodawanie zawartości lokacji pamięci o adresie 5432 do rejestru

zapis symboliczny i jego postać w asemblerze:

```
Regs[R1] ← Regs[R1] + Mem[5432]
add  R1, (5432)
```

podobny zapis w asemblerze Intel:

```
add     BX, ds:[5432]
```

Adresowanie bezpośrednie (3)

- Inny przykład adresowania bezpośredniego (asembler Intel)

```
maska    dd   5252FFFFH
_  _  _  _  _  _  _  _  _  _
xor      ebx, maska
```

POLITECHNIKA GDAŃSKA

Adresowanie rejestrowe (1)

- Odmianą adresowania bezpośredniego jest *adresowanie rejestrowe* gdy argumenty operacji są zawarte w rejestrach procesora. Wtedy w rozkazie adres argumentu przyjmuje postać identyfikatora (numeru) rejestru.
- Ponieważ w celu pobrania argumentów nie trzeba (dodatkowo) sięgać do pamięci, rozkazy z adresowaniem rejestrowym wykonują się szybciej.

Adresowanie rejestrowe (2)

- Przykład: dodawanie zawartości rejestrów

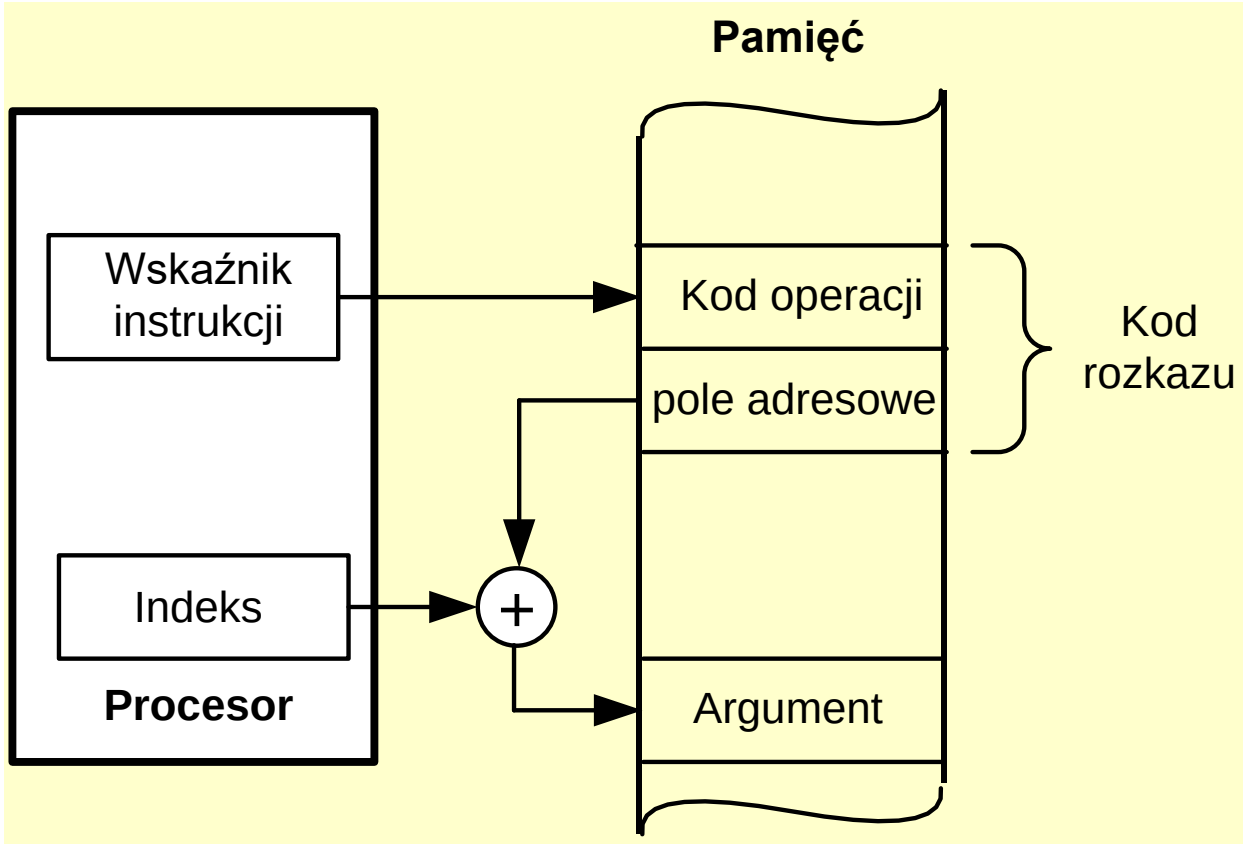
zapis symboliczny i jego postać w asemblerze:

```
Regs[R4] ← Regs[R4] + Regs[R3]
add  R4, R3
```

podobny zapis w asemblerze Intel:

```
add  DH, CL
```

Adresowanie indeksowe (1)



- W przypadku *adresowania indeksowego* adres argumentu określony jest przez sumę zawartości pola adresowego rozkazu i rejestru indeksowego.

Adresowanie indeksowe (2)

- Przykład: dodawanie do rejestru R4 zawartości lokacji pamięci — adres tej lokacji określa zawartość rejestru R1 powiększona o 360

zapis symboliczny i jego postać w asemblerze:

```
Regs[R4] ← Regs[R4] + Mem[360 + Regs[R1]]
add   R4, 360(R1)
```

podobny zapis w asemblerze Intel:

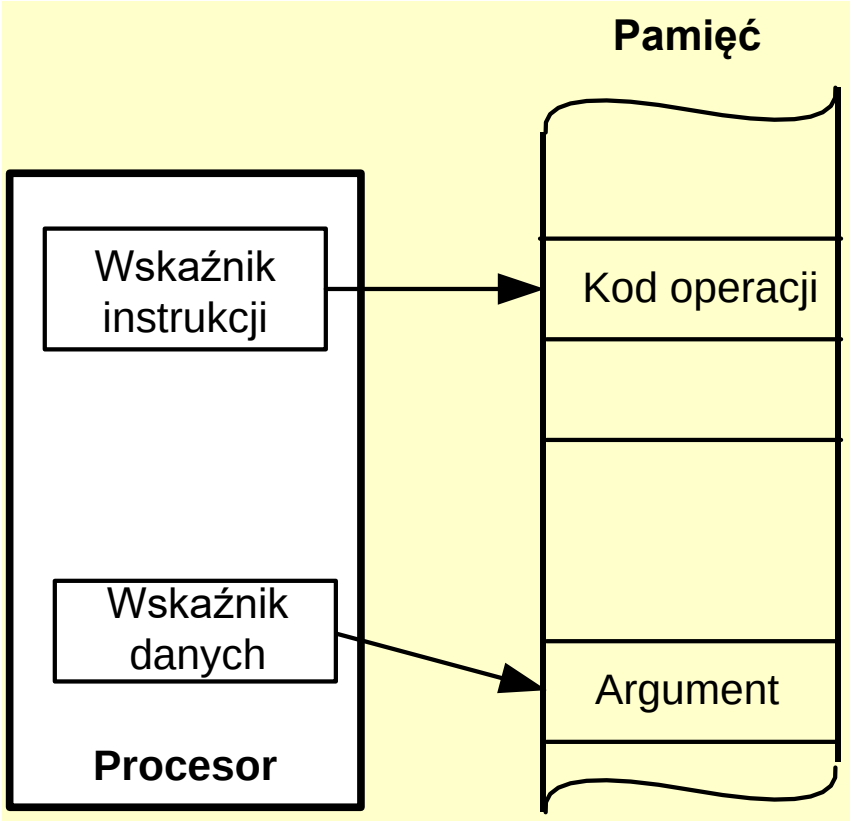
```
add   EAX, [EBX + 360]
```

Adresowanie indeksowe (3)

- W architekturze x86 rolę rejestru indeksowego może pełnić dowolny 32-bitowy rejestr ogólnego przeznaczenia: EAX, EBX, . . .

Adresowanie indeksowe (4)

- W przypadku szczególnym pole adresowe może być pominięte, co oznacza, że adres argumentu określony jest wyłącznie przez zawartość rejestru indeksowego. W literaturze tego rodzaju adresowanie określane jest jako *adresowanie za pomocą wskaźników*.



Adresowanie indeksowe (5)

- Przykład: dodawanie do rejestru R4 zawartości lokacji pamięci wskazanej przez zawartość rejestru R1

zapis symboliczny i jego postać w asemblerze:

```
Regs[R4] ← Regs[R4] + Mem[Regs[R1]]
add  R4, (R1)
```

podobny zapis w asemblerze Intel:

```
add  EAX, [ESI]
```

POLITECHNIKA GDAŃSKA

Adresowanie bazowo-indeksowe (1)

- Bardziej rozbudowaną wersją adresowania indeksowego jest *adresowanie bazowo-indeksowe ze skalowaniem* (dostępne m.in. w procesorach rodziny *x86*). W tym przypadku adres argumentu określony jest przez sumę poniższych składników:
  - zawartości rejestru bazowego,
  - zawartości rejestru indeksowego pomnożonego przez współczynnik skali,
  - pola adresowego rozkazu.

POLITECHNIKA GDAŃSKA

Adresowanie bazowo-indeksowe (2)

- Przykład: dodawanie do rejestru R1 zawartości lokacji pamięci — adres tej lokacji określa suma zawartość rejestru R2, zawartości rejestru R3 pomnożonej przez 4 oraz liczby 240

  zapis symboliczny i jego postać w asemblerze:

```
Regs[R1] ← Regs[R1] +
     Mem[240 + Regs[R2] + Regs[R3]*4]

add      R1,240(R2)[R3*4]
```

  podobny zapis w asemblerze Intel:

```
add      EAX, [ESI + EDI*4 + 240]
```

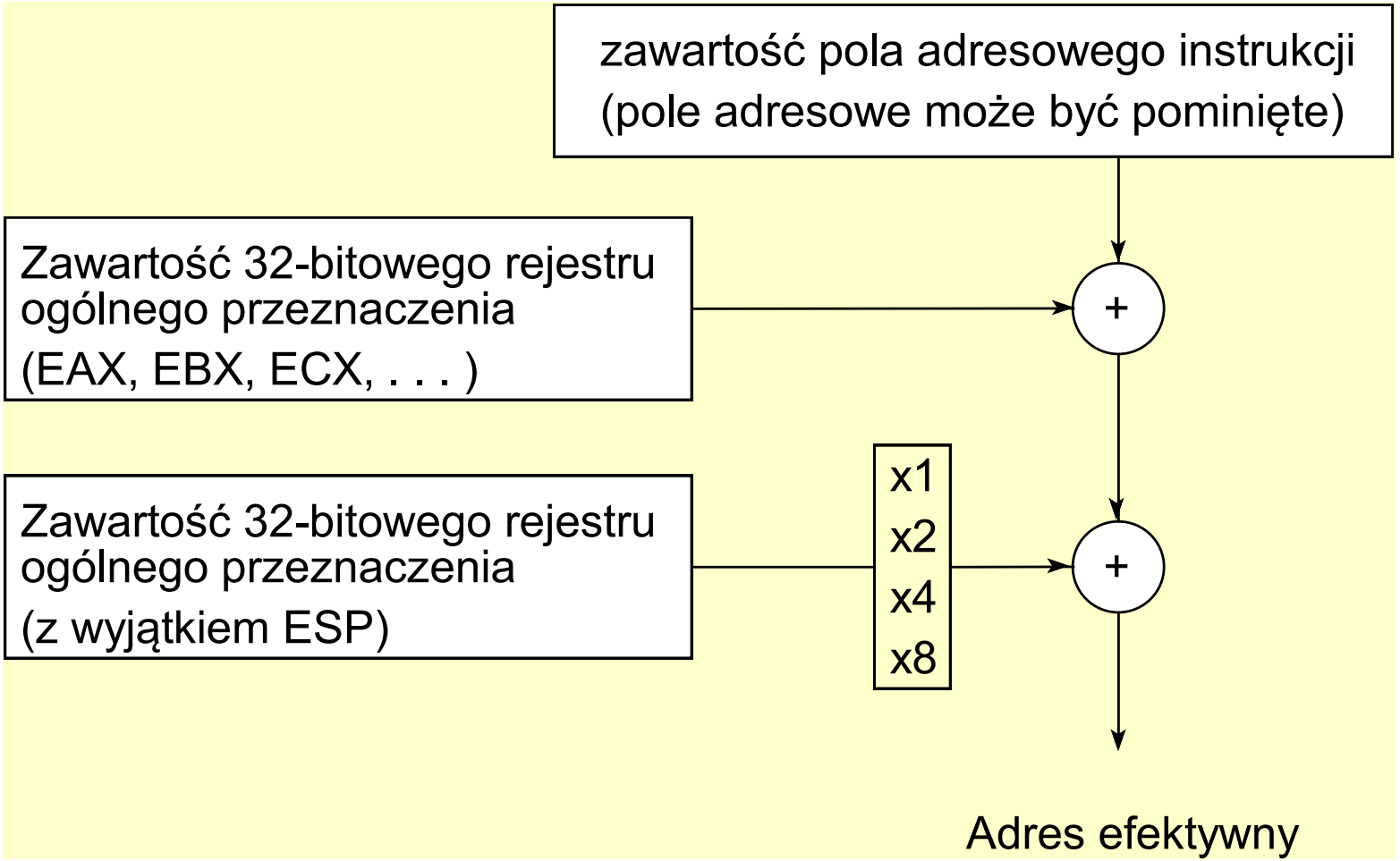
POLITECHNIKA GDAŃSKA

Adresowanie bazowo-indeksowe (3)

- Sposób obliczania adresu argumentu przy zastosowaniu adresowania bazowo-indeksowego, stosowany w procesorach x86, pokazany jest na rysunku.
- Zawartość rejestru indeksowego może być opcjonalnie mnożona przez 1, 2, 4 lub 8 (tzw. współczynnik skali).

POLITECHNIKA GDAŃSKA

Adresowanie bazowo-indeksowe (4)



POLITECHNIKA GDAŃSKA

Adresowanie bazowo-indeksowe (5)

- Przykład:

```
add     ah, [esi + 4*edi + 12]
```

- Pokazany na rysunku schemat adresowania bazowo-indeksowego w procesorach x86 obejmuje także wcześniej wymienione przypadki szczególne.
- Jeśli w zapisie rozkazu nie określono rejestru bazowego ani indeksowego, to adres efektywny jest równy zawartości pola adresowego rozkazu.

POLITECHNIKA GDAŃSKA

Adresowanie bazowo-indeksowe (6)

- Opcjonalnie można wskazać tylko rejestr bazowy albo rejestr indeksowy z wymaganym współczynnikiem skali; jeśli w zapisie rozkazu podano rejestr bazowy lub indeksowy, to pole adresowe rozkazu może być pominięte.

Operacje na tablicach

- Posługując się adresowaniem bazowo-indeksowym (lub tylko indeksowym) można łatwo budować pętle rozkazowe, w których ten sam rozkaz wykonuje działania na kolejnych elementach tablicy.
- W pętlach tego typu zwykle rejestr bazowy wskazuje położenie tablicy w pamięci, a rejestr indeksowy wskazuje położenie elementu tablicy względem jej początku. Może też być uwzględniona wartość podana w polu adresowym rozkazu.

POLITECHNIKA GDAŃSKA

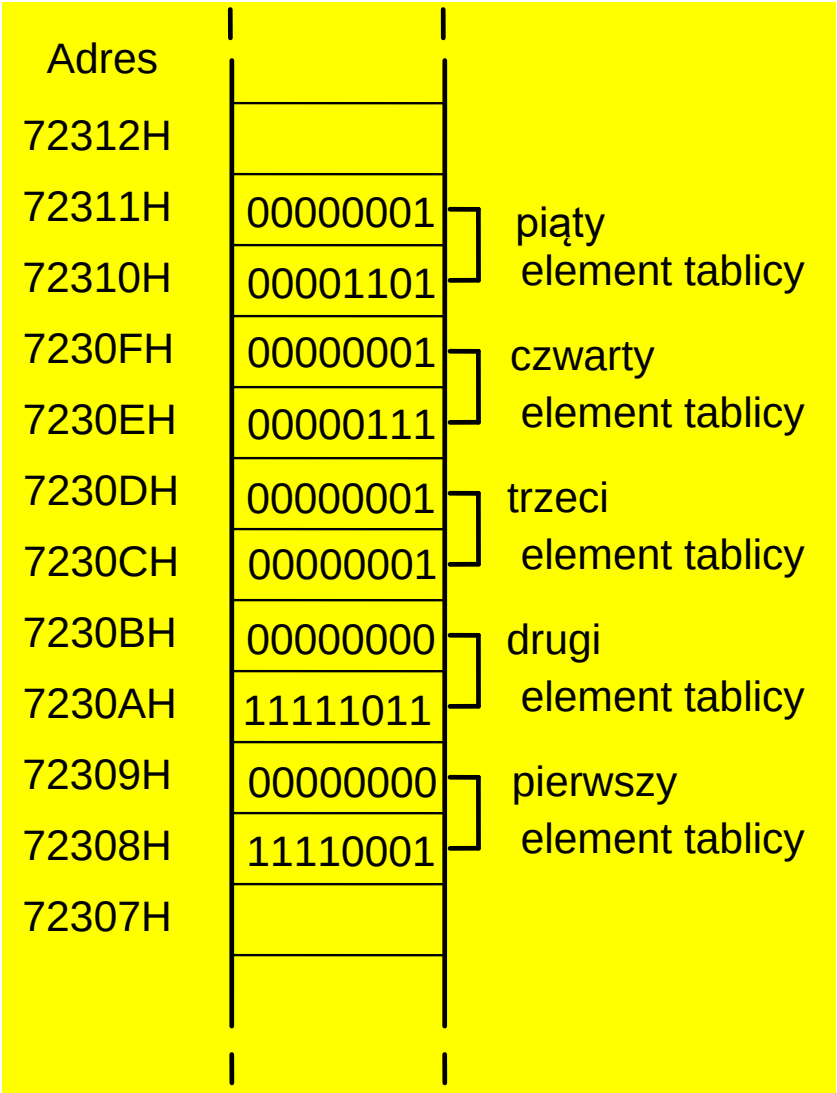
Przykład sumowania elementów tablicy (1)

- Powracamy do omawianego wcześniej przykładu sumowania liczb w tablicy. W podanym dalej rozwiązaniu używana jest pętla rozkazowa.
- W pamięci głównej (operacyjnej) komputera, począwszy od adresu 72308H, znajduje się tablica zawierająca pięć liczb 16-bitowych całkowitych bez znaku — tablica ta stanowi część obszaru danych programu.

POLITECHNIKA GDAŃSKA

Przykład sumowania elementów tablicy (2)

- Litera H występująca po cyfrach liczby oznacza, że wartość liczby została podana w kodzie szesnastkowym (heksadecymalnym).



POLITECHNIKA GDAŃSKA

Przykład sumowania elementów tablicy (3)

- Obliczenie sumy wymaga więc wykonania w pętli 4 operacji dodawania. Początkowa zawartość licznika obiegów pętli wpisywana jest do rejestru ECX.
- Sterowanie pętlą wykonywane jest za pomocą rozkazu **loop**, który opisany jest dalej.
- Dodatkowo zakładamy, że kolejne sumy uzyskiwane w trakcie dodawania dadzą się przedstawić w postaci liczb 16-bitowych (nie wystąpi przepełnienie — nadmiar).

POLITECHNIKA GDAŃSKA

Przykład sumowania elementów tablicy (4)

- W podanym przykładzie istotną rolę odgrywa rozkaz dodawania:

```
add     ax, ds:[7230AH][ebx]
```

- Adres [7230AH] wskazuje położenie drugiego elementu tablicy (adres pierwszego elementu wynosi 72308H).
- Rejestr EBX pełni funkcje rejestru indeksowego.
- W każdym obiegu pętli zawartość rejestru EBX jest zwiększana o 2, tak by wskazać kolejny element tablicy.

POLITECHNIKA GDAŃSKA

Przykład sumowania elementów tablicy (5)

```
   mov     ecx, 4           ; licznik obiegów pętli
   mov     ax, ds:[72308H]  ; pocz. wartość sumy
   mov     ebx, 0           ; pocz .zawartość
                            ; rejestru indeksowego
ptl_suma:

   ; dodanie kolejnego elementu tablicy
   add     ax, ds:[7230AH][ebx]

   add     ebx, 2     ; zwiększenie indeksu
   loop    ptl_suma   ; sterowanie pętlą
```

POLITECHNIKA GDAŃSKA

Rozkaz loop (1)

- Rozkaz **loop** umieszczony na końcu pętli rozkazowej stanowi odmianę rozkazu skoku warunkowego:
  - Najpierw rozkaz zmniejsza zawartość rejestru ECX o 1 (w przypadku szczególnym, jeśli zawartość ECX wynosi 0, to po zmniejszeniu w ECX znajdować się będzie liczba -1, czyli FFFFFFFFH).
  - Następnie, jeśli po odejmowaniu rejestr ECX zawiera liczbę różną od zera, to warunek jest spełniony i następuje skok na początek pętli.
  - Jeśli po odejmowaniu ECX zawiera 0, to warunek nie jest spełniony i następuje przejście do następnego rozkazu.

POLITECHNIKA GDAŃSKA

Rozkaz loop (2)

- W przypadku, gdy warunek jest spełniony, skok na początek pętli polega w istocie na dodaniu do rejestru EIP liczby (w tym przypadku ujemnej) zawartej w polu adresowym rozkazu **loop**.
- Odpowiednia zawartość pola adresowego rozkazu **loop** jest wyznaczana przez asembler lub kompilator („ręczne” obliczenie wymaga znajomości liczby bajtów zajmowanych przez poszczególne rozkazy wchodzące w skład pętli).

loop

01110100	pole adresowe

ECX ← ECX – 1
następnie: warunek spełniony,
gdy ECX ≠ 0

POLITECHNIKA GDAŃSKA

Rozkaz loop (3)

- W niektórych sytuacjach rozkaz **loop** można zastąpić parą rozkazów:

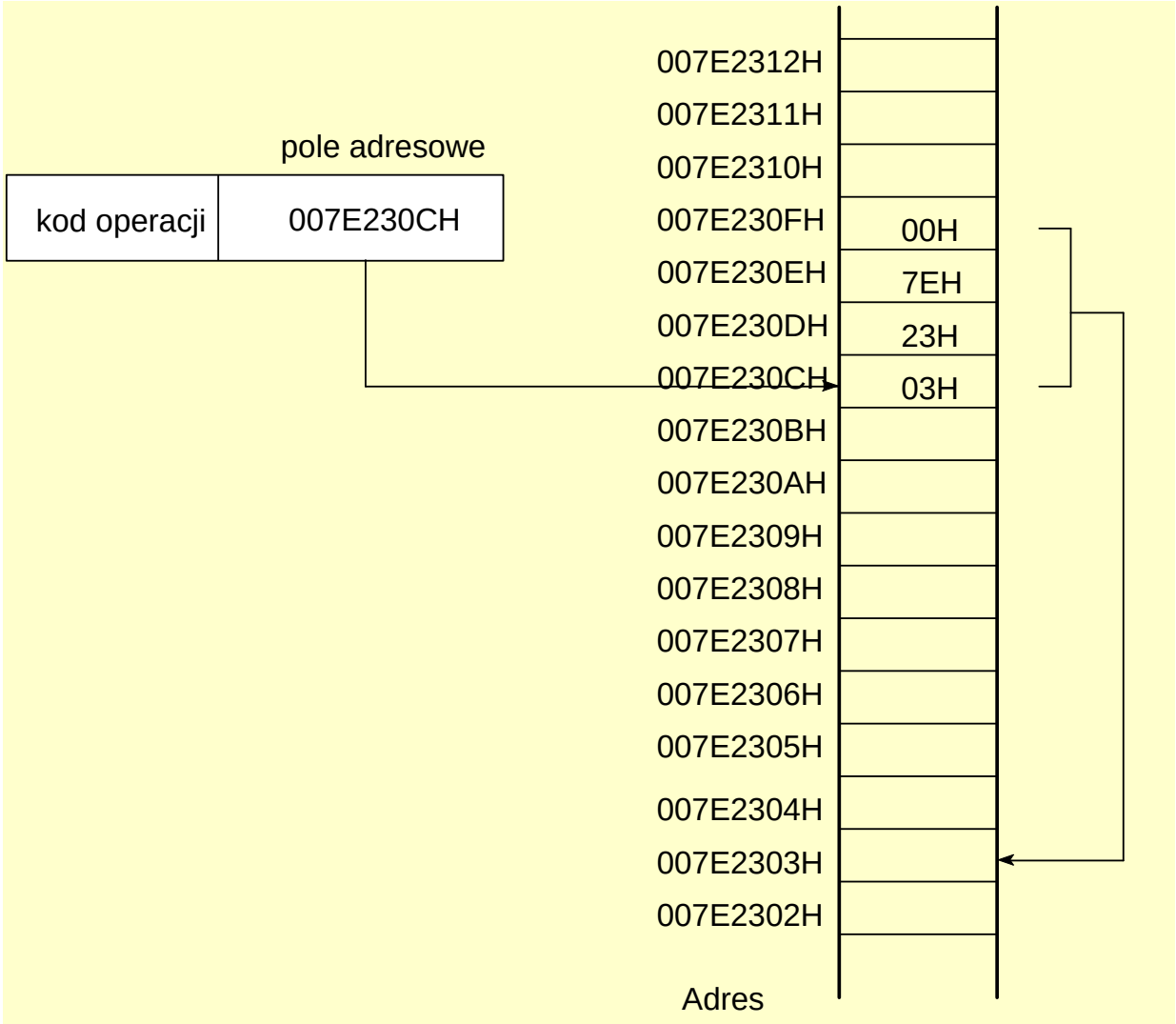
```
dec  ecx
jnz  jakaś_etykieta
```

- Rozkaz **loop** jest powszechnie stosowany do organizacji pętli rozkazowych.

POLITECHNIKA GDAŃSKA

Adresowanie pośrednie (1)

- W tym przypadku adres podany w polu adresowym wskazuje lokację pamięci (zwykle 4-bajtową), w której zawarty jest adres argumentu.



Adresowanie pośrednie (2)

- Przykład: dodawanie do rejestru R4 zawartości lokacji pamięci, której adres zawarty jest lokacji pamięci wskazanej przez rejestr R3

zapis symboliczny i jego postać w asemblerze:

```
Regs[R4] ← Regs[R4] + Mem[Mem[Regs[R3]]]
add  R4,@(R3)
```

w procesorach x86 adresowanie pośrednie dostępne jest tylko dla rozkazów sterujących (skoków)

POLITECHNIKA GDAŃSKA

Adresowanie z autoinkrementacją (1)

- Ten rodzaj adresowania (niedostępny w procesorach x86) stanowi rozszerzenie adresowania indeksowego: dodatkowo, po wykonaniu właściwej operacji automatycznie zwiększana jest zawartość rejestru indeksowego.

POLITECHNIKA GDAŃSKA

Adresowanie z autoinkrementacją (2)

- Przykład: dodawanie do rejestru R1 zawartości lokacji pamięci, której adres podany jest w rejestrze R2. Po wykonaniu dodawania zawartość rejestru R2 jest zwiększana o stałą *d* (zwykle 2, 4 lub 8).

zapis symboliczny i jego postać w asemblerze:

```
Regs[R1] ← Regs[R1] + Mem[Regs[R2]]
Regs[R2] ← Regs[R2] + d

add  R1,(R2)+
```

POLITECHNIKA GDAŃSKA

Adresowanie z autodekrementacją (1)

- Ten rodzaj adresowania (niedostępny w procesorach x86) stanowi rozszerzenie adresowania indeksowego: dodatkowo, przed wykonaniem właściwej operacji automatycznie zmniejszana jest zawartość rejestru indeksowego.

POLITECHNIKA GDAŃSKA

Adresowanie z autodekrementacją (2)

- Przykład: dodawanie do rejestru R1 zawartości lokacji pamięci, której adres podany jest w rejestrze R2. Przed wykonaniem dodawania zawartość rejestru R2 jest zmniejszana o stałą *d* (zwykle 2, 4 lub 8).

zapis symboliczny i jego postać w asemblerze:

```
Regs[R2] ← Regs[R2] – d
Regs[R1] ← Regs[R1] + Mem[Regs[R2]]

add  R1, –(R2)
```

POLITECHNIKA GDAŃSKA

Obliczanie adresu efektywnego (1)

- W procesorach zgodnych z architekturą x86 adres efektywny obliczany jest modulo 2^{32}, tj. po obliczeniu sumy zawartości pola adresowego rozkazu i zawartości rejestrów indeksowych bierze się pod uwagę 32 najmniej znaczące bity uzyskanego wyniku.
- Podana reguła pozwala w szczególności na uzyskiwanie adresów efektywnych mniejszych od zawartości pola adresowego rozkazu — ilustruje to przykład:

POLITECHNIKA GDAŃSKA

Obliczanie adresu efektywnego (2)

rozkaz mov ...., [ebx] + 000003A9H

. . .	. . .	A9	03	00	00

rejestr EBX | FFFFFFFCH

0 0 0 0 0 3 A 9
+ F F F F F F F C

1 0 0 0 0 0 3 A 5

adres efektywny

Architektury 32- i 64-bitowe (1)

- W ciągu ostatnich 30 lat nastąpiły bardzo znaczne zmiany w konstrukcji procesorów zgodnych z architekturą x86, ale zmiany te miały charakter łagodny, nie powodując istotnych trudności dla użytkowników komputerów.
- W szczególności wprowadzono pośrednie tryby pracy, np. tryb V86 symulujący pracę procesora 8086 w procesorach nowszych typów.
- Aktualnie, przejście z architektury 32-bitowej na 64-bitową odbywa się także w sposób niezauważalny dla użytkowników komputerów.

POLITECHNIKA GDAŃSKA

Architektury 32- i 64-bitowe (2)

- Ok. roku 2000 firma Intel opracowała nową architekturę IA–64 (procesor Itanium), całkowicie odrębną od architektury x86 (Intel 32).
- Architektura IA–64 nie rozpowszechniła się, natomiast aprobatę uzyskała architektura 64-bitowa opracowana przez firmę AMD znana jako AMD64 (procesor Opteron, 2003).
- Architektura AMD64 stanowi 64-bitowe rozwinięcie powszechnie używanej architektury x86.

Architektury 32- i 64-bitowe (3)

- Kierując się podobnymi przesłankami firma Intel zaprojektowała architekturę IA–32e/EM64T — listy rozkazów procesorów zgodnych z architekturą AMD64 i Intel 64 są prawie identyczne.
- Po wprowadzeniu procesorów o architekturze 64–bitowej firma Intel przyjęła oznaczenia Intel 32 zamiast IA–32 i Intel 64 zamiast IA–32e/EM64T.
- Warto zwrócić uwagę, że oznaczenie IA–64 (także Intel Itanium) dotyczy nowoczesnej architektury procesorów, aczkolwiek nie używanych w komputerach PC.

POLITECHNIKA GDAŃSKA

Tryb rzeczywisty i tryb chroniony w procesorach x86 (1)

- Procesory rodziny x86 mogą pracować w kilku trybach, z których najważniejsze znaczenie mają:
  - *tryb rzeczywisty* (ang. real mode), w którym procesor zachowuje się podobnie do swojego poprzednika 8086/88;
  - *tryb chroniony* (ang. protected mode), w którym procesor stosuje złożone mechanizmy adresowania i ochrony pamięci, wspomaga implementację pamięci wirtualnej i wielozadaniowości, blokuje niektóre operacje dla zwykłych programów użytkowych.

POLITECHNIKA GDAŃSKA

Tryb rzeczywisty i tryb chroniony w procesorach x86 (2)

- Bezpośrednio po włączeniu (lub zresetowaniu) komputera procesor pracuje w trybie rzeczywistym, dopiero uruchomiony system operacyjny (Windows, Linux) powoduje przełączenie (w sposób programowy) do trybu chronionego.
- Tryb chroniony stanowi obecnie podstawowy tryb pracy procesora, w tym trybie pracują systemy operacyjne i wykonywane są aplikacje.

POLITECHNIKA GDAŃSKA

Tryb rzeczywisty i tryb chroniony w procesorach x86 (3)

- Wyjątkowo, spotyka się programy opracowane w latach 80. i 90. ubiegłego stulecia, które były przewidziane do wykonywania w trybie rzeczywistym. Aktualnie eksploatowane systemy operacyjne zazwyczaj nie pozwalają na wykonywanie programów tej klasy. Ewentualnie można je uruchomić za pomocą maszyny wirtualnej (np. DOSBox).
- Opisane dalej mechanizmy adresowania w trybie rzeczywistym aktualnie używane w bardzo wąskim zakresie (np. bezpośrednio po włączeniu komputera). Jednak ich znajomość pozwala lepiej zrozumieć rozwój architektury procesorów.

POLITECHNIKA GDAŃSKA

Adresowanie pamięci w trybie rzeczywistym (1)

- W trybie rzeczywistym procesor odwołuje się do pamięci głównej (operacyjnej), której rozmiar ograniczony jest do 1MB, co wymaga stosowania 20-bitowych linii adresowych.
- W trakcie wykonywania rozkazów, które odwołują się do komórek pamięci procesor wyznacza każdorazowo adres fizyczny komórki pamięci.

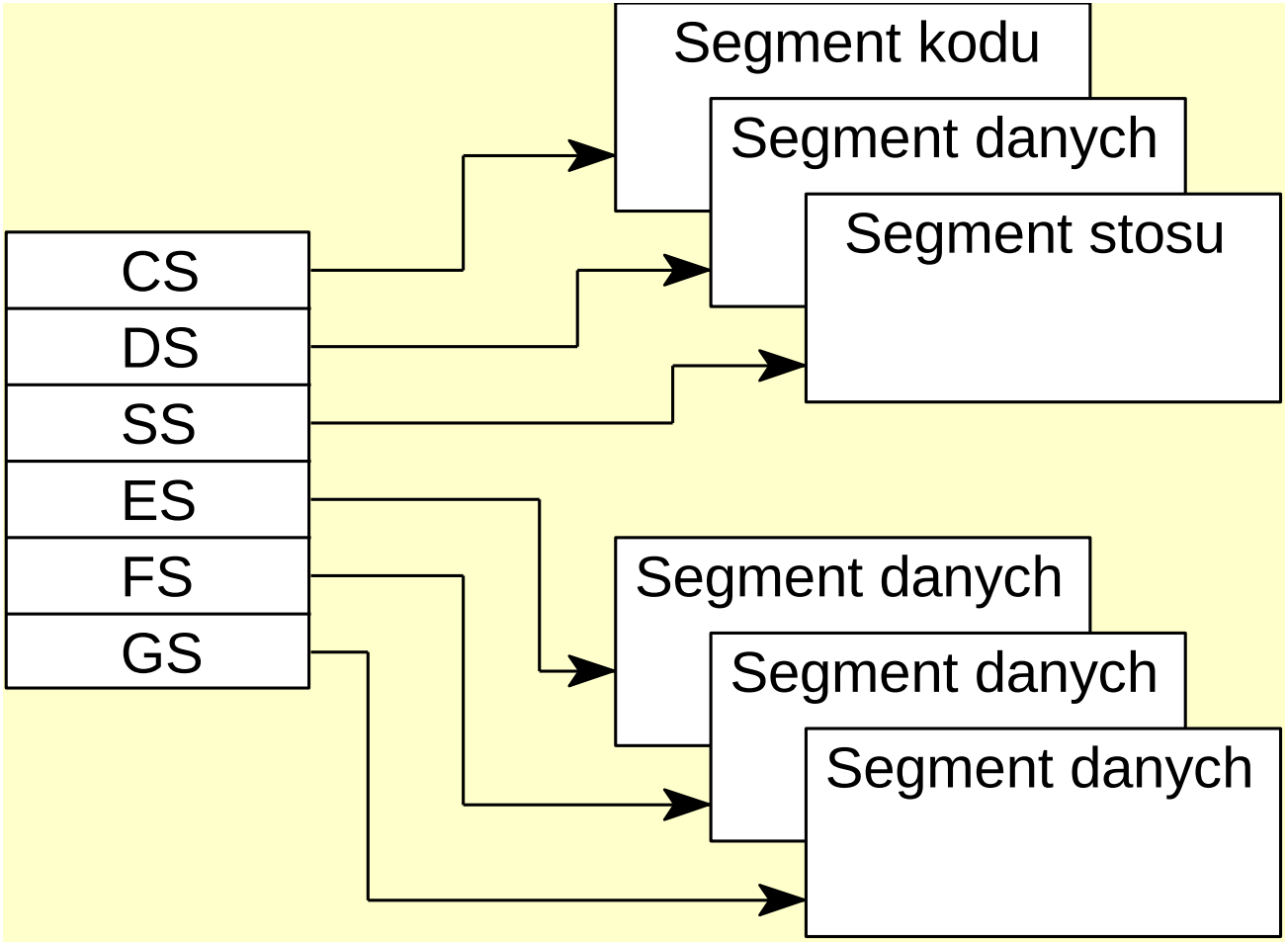
POLITECHNIKA GDAŃSKA

Adresowanie pamięci w trybie rzeczywistym (2)

- W procesie wyznaczania adresu istotną rolę odgrywają 16-bitowe rejestry segmentowe: CS, DS, ES, SS (i wprowadzone później FS, GS), w szczególności:
  - Rejestr CS (ang. code segment) wskazuje położenie w pamięci obszaru rozkazowego programu,
  - Rejestr DS (ang. data segment) wskazuje położenie obszaru danych programu,
  - Rejestr SS (ang. segment stack) wskazuje położenie stosu.

POLITECHNIKA GDAŃSKA

Adresowanie pamięci w trybie rzeczywistym (3)

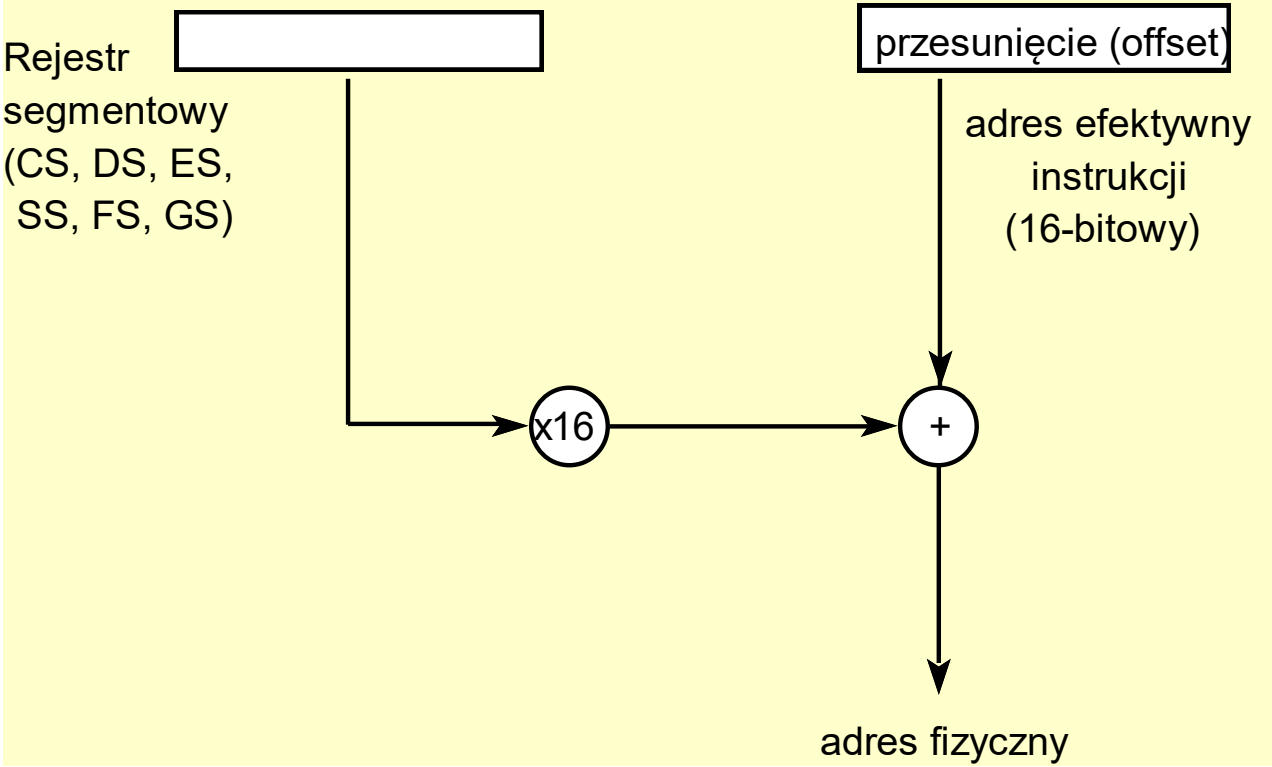


POLITECHNIKA GDAŃSKA

Adresowanie pamięci w trybie rzeczywistym (4)

- W trybie rzeczywistym:

adres fizyczny = zawartość rejestru segmentowego * 16 + przesunięcie



POLITECHNIKA GDAŃSKA

Adresowanie pamięci w trybie rzeczywistym (5)

- W trybie rzeczywistym adres lokacji pamięci wyrażany jest zazwyczaj w postaci dwóch liczb:

segment : offset

- Parę tą często nazywa się w literaturze adresem logicznym.
- Przykładowo, 32-bitowy programowy licznik czasu umieszczony jest w lokacji pamięci o adresie logicznym 40H : 6CH, tzn. w lokacji pamięci o adresie fizycznym 40H * 16 + 6CH = 46CH.

POLITECHNIKA GDAŃSKA

Adresowanie pamięci w trybie chronionym

- Sposób obliczania adresu fizycznego w trybie chronionym jest znacznie bardziej skomplikowany: zawartości rejestrów segmentowych traktowane są jako indeksy do tablic systemowych, w których zawarte są adresy podlegające jeszcze dalszym przekształceniom w ramach mechanizmu stronicowania.
- We współczesnych systemach operacyjnych obserwuje się stosowanie trybów adresowania, które marginalizują rolę rejestrów segmentowych (CS, DS, …).

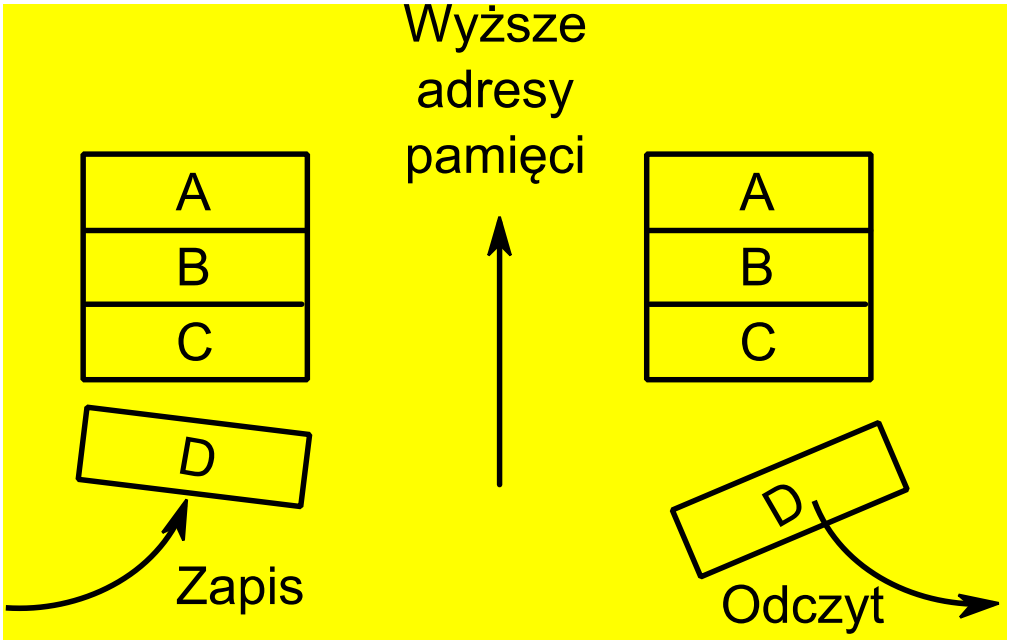
POLITECHNIKA GDAŃSKA

Organizacja stosu (1)

- W ujęciu abstrakcyjnym stos stanowi liniową strukturę danych o nieograniczonej pojemności.
- Stos dostępny jest do zapisywania i odczytywania danych tylko z jednego końca, nazywanego *wierzchołkiem stosu*.
- Stos klasyfikowany jest jako struktura danych typu LIFO (ang. Last In, First Out) — ostatni na wejściu, pierwszy na wyjściu.
- Wprowadzenie nowej pozycji danych na stos nazywane jest *zapisem* lub *załadowaniem* (ang. pushing), a odwrotnością tej operacji jest *odczyt* lub *zdjęcie* (ang. popping).

Organizacja stosu (2)

- W typowych komputerach stos umieszczony jest w pamięci operacyjnej, przy czym kolejne dane zapisywane na stosie lokowane są w komórkach pamięci o coraz niższych adresach – „stos rośnie w kierunku malejących adresów”.



POLITECHNIKA GDAŃSKA

Wierzchołek stosu (1)

- W operacjach wykonywanych na stosie szczególne znaczenie ma ostatnio zapisana dana, stanowiąca *wierzchołek stosu.*
- Położenie wierzchołka stosu w pamięci komputera wskazuje rejestr nazywany **wskaźnikiem stosu** (ang. stack pointer).
- W architekturze x86 rolę wskaźnika stosu pełni 32-bitowy rejestr ESP, natomiast w architekturze x86-64 — rejestr RSP (64-bitowy).
- Zatem rejestr ESP (lub RSP) wskazuje adres komórki pamięci, w której znajduje się dana ostatnio zapisana na stosie.

Wierzchołek stosu (2)

- W klasycznym ujęciu, ze stosu może być odczytywana dana znajdująca się tylko na jego wierzchołku, co oznacza, że dostęp do danej znajdującej wewnątrz stosu wymaga uprzedniego usunięcia innych danych.
- Jak pokażemy dalej współczesne procesory udostępniają mechanizmy adresowania, które pozwalają na dostęp do danych znajdujących się wewnątrz stosu bez konieczności usuwania innych danych. Stos w komputerze nie może być więc traktowany jako idealne odzwierciedlenie struktury danych LIFO (ang. Last In, First Out).

Operacje push i pop (1)

- Procesor realizuje dwie podstawowe operacje stosu:

  push — zapisywanie danych na stosie,

  pop — odczytywanie danych ze stosu.

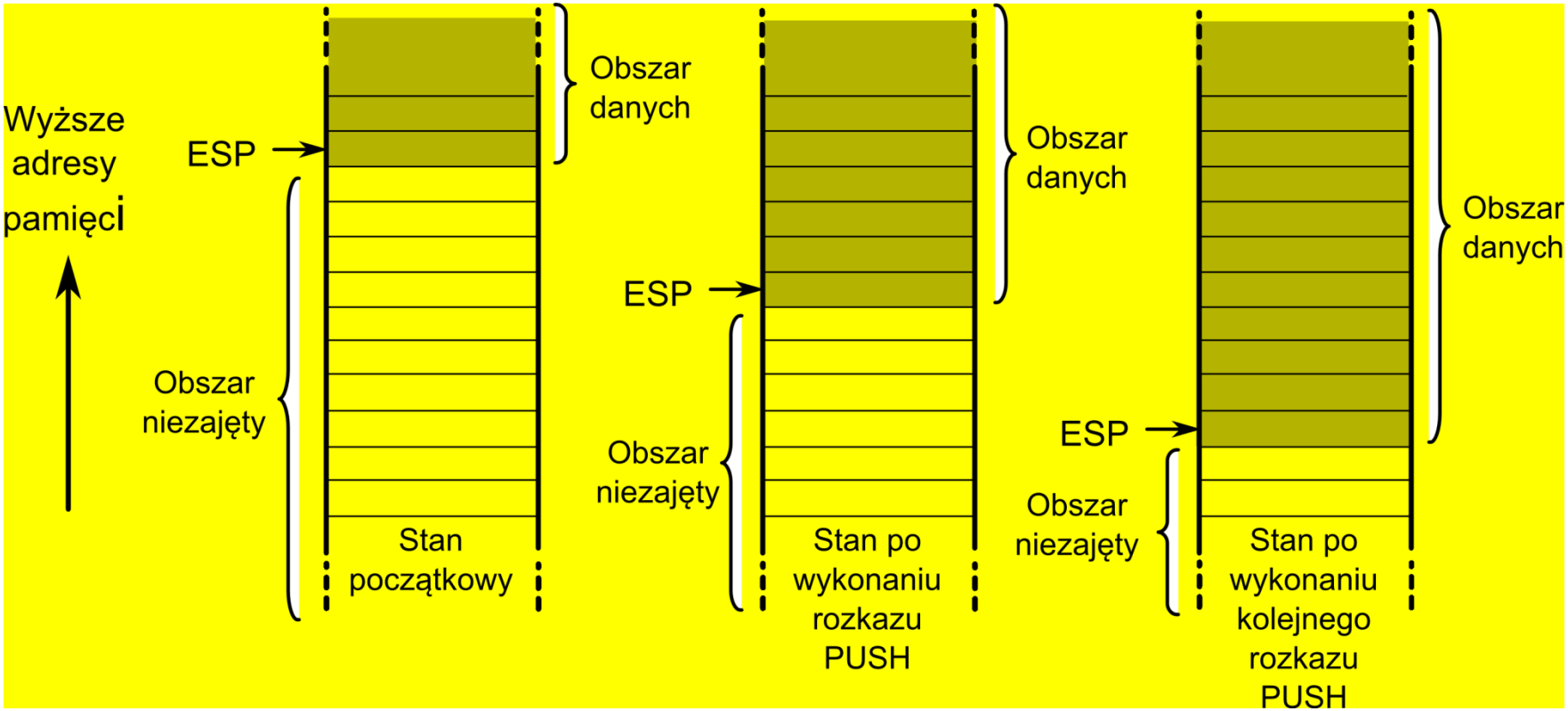
- W architekturze x86 (w trybie 32-bitowym) przed zapisaniem nowej danej stosie procesor zmniejsza rejestr ESP o 4 („stos rośnie w kierunku malejących adresów”), analogicznie przy odczytywaniu zwiększa ESP o 4 po odczytaniu danej.

Operacje push i pop (2)

- Analogicznie, w trybie 64-bitowym przed zapisaniem nowej danej stosie procesor zmniejsza rejestr RSP o 8, a przy odczytywaniu zwiększa RSP o 8 po odczytaniu danej.
- W trybie 32-bitowym wymaga się by zawartość rejestru ESP była podzielna przez 4, czyli zawartość rejestru ESP musi wskazywać lokację pamięci o adresie podzielnym przez 4.
- W trybie 64-bitowym zawartość rejestru RSP musi być podzielna przez 8, a w niektórych przypadkach przez 16 (dalsze szczegóły podane są w instrukcji do ćwiczenia laboratoryjnego nr 4).

POLITECHNIKA GDAŃSKA

Operacje push i pop (3) (przykład dla trybu 32-bitowego)



Operacje push i pop (4)

- Rozkazy push i pop mają jeden operand, którym najczęściej jest 32-bitowy rejestr procesora (albo 64-bitowy w architekturze 64-bitowej).
- Przykładowo, rozkaz

```
push   ecx
```

  powoduje zapisanie na stosie zawartości rejestru ECX.
- Rozkaz

```
pop   edi
```

  powoduje usunięcie danej na wierzchołku stosu i wpisanie jej do rejestru EDI.

Operacje push i pop (5)

- Po usunięciu danej z wierzchołka nowym wierzchołkiem staje się dana, która uprzednio znajdowała się pod wierzchołkiem. Zmiana ta jest wynikiem zwiększenia zawartości wskaźnika stosu ESP o 4 (w trybie 32-bitowym).
- Warto zwrócić uwagę, że pobranie danej ze stosu za pomocą rozkazu **pop** nie oznacza usunięcia tej danej z pamięci – pozostaje ona nadal w obszarze niezajętym stosu (zob. rys.). Dana ta nie należy jednak do obszaru danych stosu i nie może być używana do obliczeń – w każdej chwili może być nadpisana w wyniku wykonania instrukcji **push** lub wskutek wystąpienia przerwania sprzętowego.

Operacje push i pop (6)

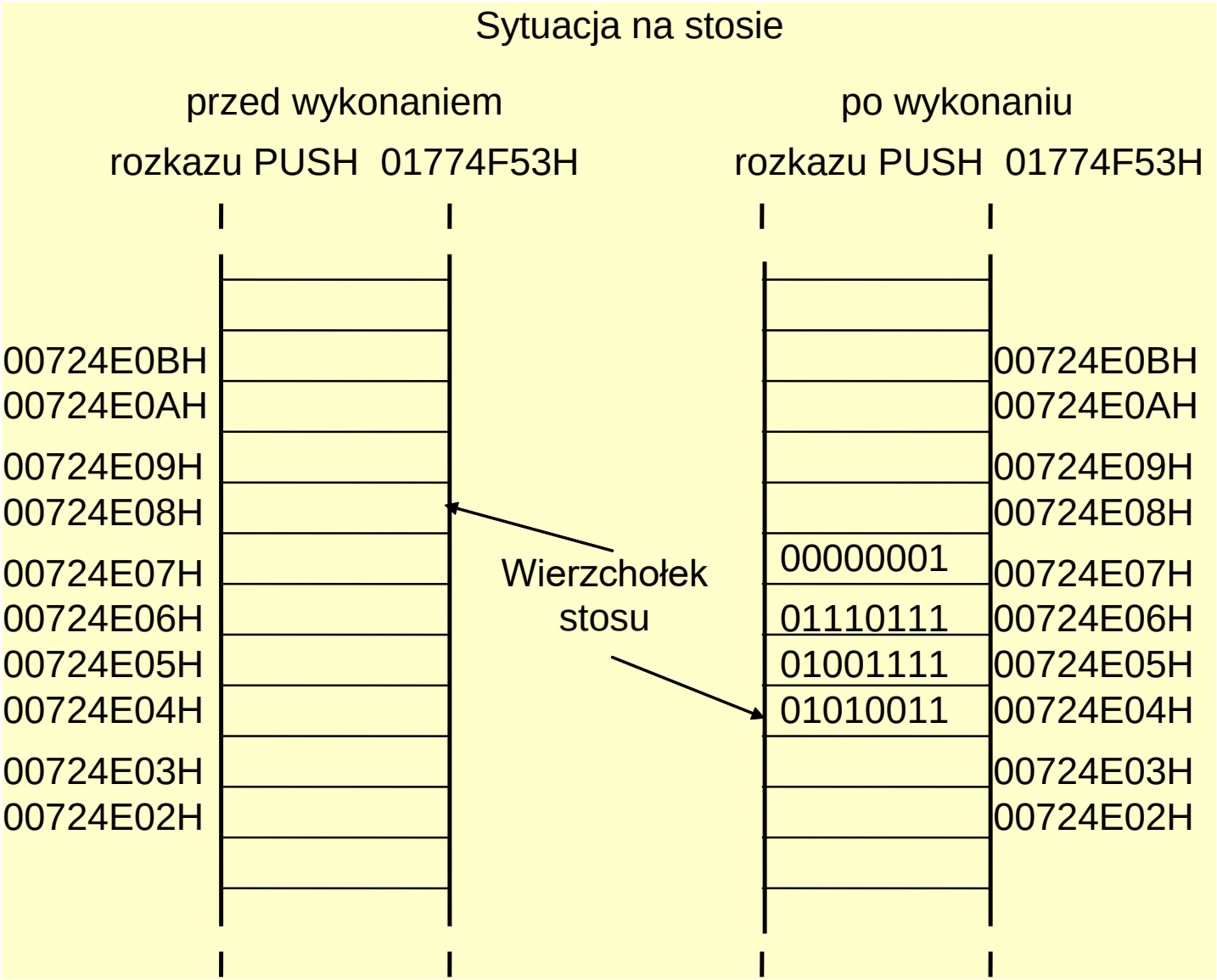
- Operandem rozkazów push i pop może być także zmienna znajdująca się w pamięci, np.

```
wynik       dd   2345
_   _   _   _   _   _
push        wynik
```

- Ponadto operandem rozkazu push może być wartość liczbowa, np.

```
push        01774F53H
```

Przykład wykonania operacji push



POLITECHNIKA GDAŃSKA

Przechowywanie wyników pośrednich (1)

- W praktyce programowania występują wielokrotnie sytuacje, w których konieczne jest tymczasowe przechowanie zawartości rejestru — zazwyczaj rejestrów ogólnego przeznaczenia jest zbyt mało by przechowywać w nich wszystkie wyniki pośrednie występujące w trakcie obliczeń.
- Wyniki pośrednie uzyskiwane w trakcie obliczeń można przechowywać w zwykłym obszarze danych programu — operacja ta (rozkaz MOV) wymaga podania dwóch argumentów: zapisywanej wartości i adresu komórki pamięci, w której ta wartość ma zostać zapisana.

POLITECHNIKA GDAŃSKA

Przechowywanie wyników pośrednich (2)

- Taka technika jest dość niepraktyczna, ponieważ przechowanie potrzebne jest tylko przez krótki odcinek czasu, natomiast lokacja pamięci musi być rezerwowana na cały czas wykonywania programu.
- Zapisywanie wyników pośrednich na stosie jest wygodniejsze: podaje się wyłącznie wartość, która ma być zapisana, przy czym nie potrzeba podawać adresu — zapisywana wartość zostaje umieszczona na wierzchołku stosu.
- Ponadto po usunięciu danej ze stosu, w zwolnionym obszarze pamięci mogą być zapisane inne dane.

POLITECHNIKA GDAŃSKA

Typowe zastosowania stosu

- Przechowywanie wyników pośrednich,
- obliczanie wartości wyrażeń arytmetycznych,
- przechowywanie adresu powrotu podprogramu,
- przechowywanie zmiennych lokowanych dynamicznie,
- przekazywanie parametrów do podprogramu.

POLITECHNIKA GDAŃSKA

Formaty danych zapisywanych na stosie

- W architekturze 32-bitowej na stosie mogą być zapisywane wyłącznie wartości 32-bitowe (4 bajty), analogicznie w architekturze 64-bitowej na stosie mogą być zapisywane wartości 64-bitowe (8 bajtów).
- Jeśli konieczne jest zapisanie danej kodowanej na mniejszej liczbie bitów, to należy zapisać tę daną w postaci rozszerzonej, np. do 32 bitów, a po odczycie zignorować starsze bity.

POLITECHNIKA GDAŃSKA

Równoważenie liczby operacji zapisu i odczytu na stosie

- W praktyce programowania należy zwracać uwagę na równoważenie liczby operacji zapisu na stos (PUSH) i odczytu ze stosu (POP).
- W bardziej rozbudowanych programach występują sytuacje nadzwyczajne: użytkownik wprowadza czasami błędne dane, wskutek czego pętle rozkazowe mogą kończyć po wykonaniu mniejszej liczby obiegów niż planowano, niektóre fragmenty mogą być pominięte, co w rezultacie może powodować niezrównoważenie stosu — wynikające stąd błędy mogą być trudne do wykrycia.

POLITECHNIKA GDAŃSKA

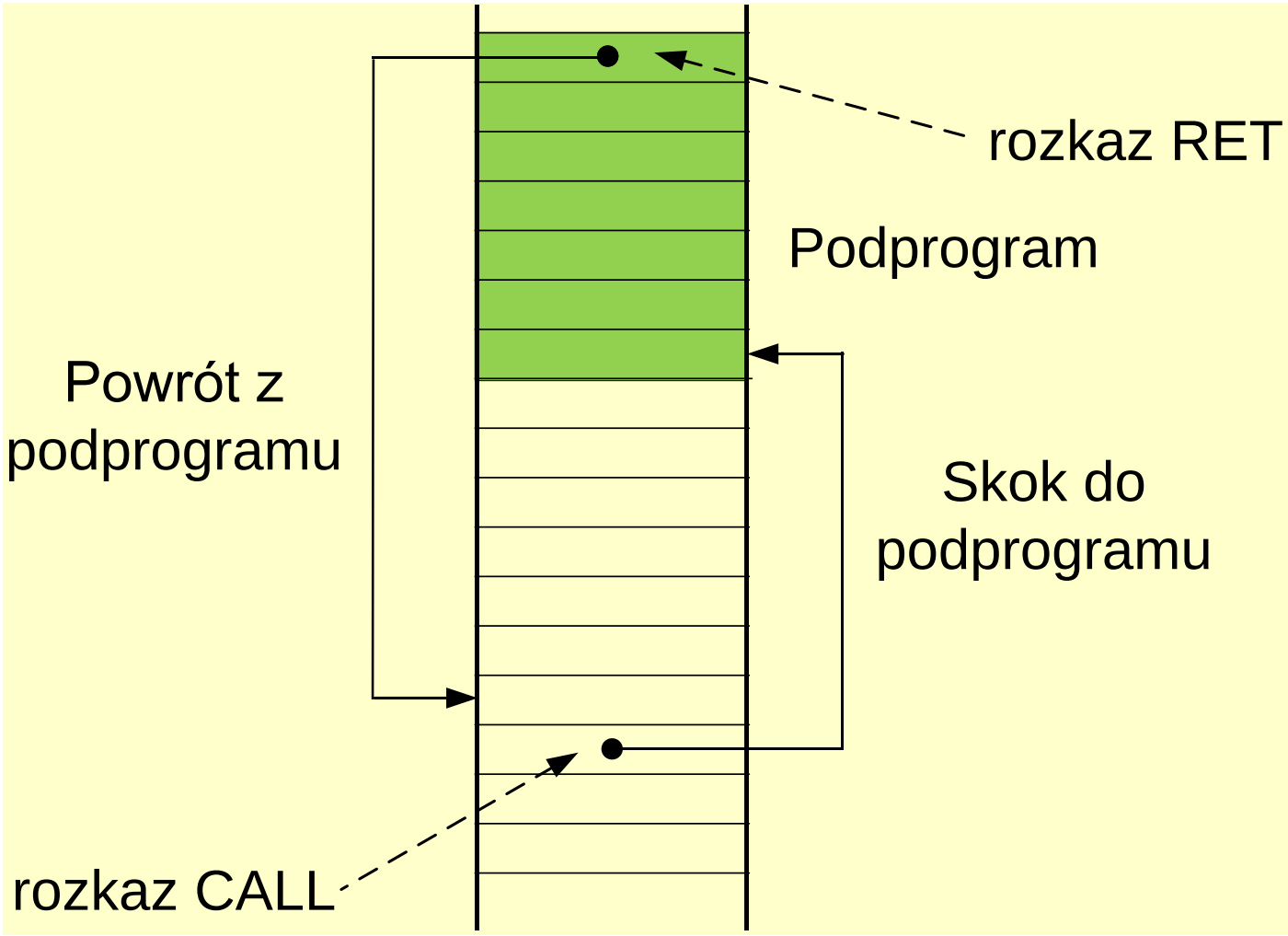
Organizacja stosu w innych architekturach

- W architekturze x86 przyjęto, że wszystkie elementy stosu przechowywane są w pamięci głównej (operacyjnej).
- W innych architekturach spotyka się rozwiązania, w których wierzchołek stosu wraz z kilkoma elementami przylegającymi umieszczony jest w zarezerwowanych rejestrach procesora — przyśpiesza to znacznie odczyt danych ze stosu.
- Niekiedy tworzone są dwa stosy, z których jeden przechowuje dane, a drugi adresy (np. ślad tworzony w chwili wywołania podprogramu).
- Czasami występuje odrębny moduł pamięci wyłącznie dla stosu.

Podprogramy (1)

- Podprogramy, w innych językach programowania nazywane także procedurami lub funkcjami, stanowią wygodny sposób kodowania wielokrotnie powtarzających się fragmentów programu.
- Na poziomie rozkazów procesora wywołanie podprogramu polega na wykonaniu skoku bezwarunkowego, przy czym dodatkowo zapamiętuje się **ślad**, czyli położenie w pamięci kolejnego rozkazu, który powinien zostać wykonany po zakończeniu podprogramu.

Podprogramy (2)



POLITECHNIKA GDAŃSKA

Rozkazy CALL i RET (1)

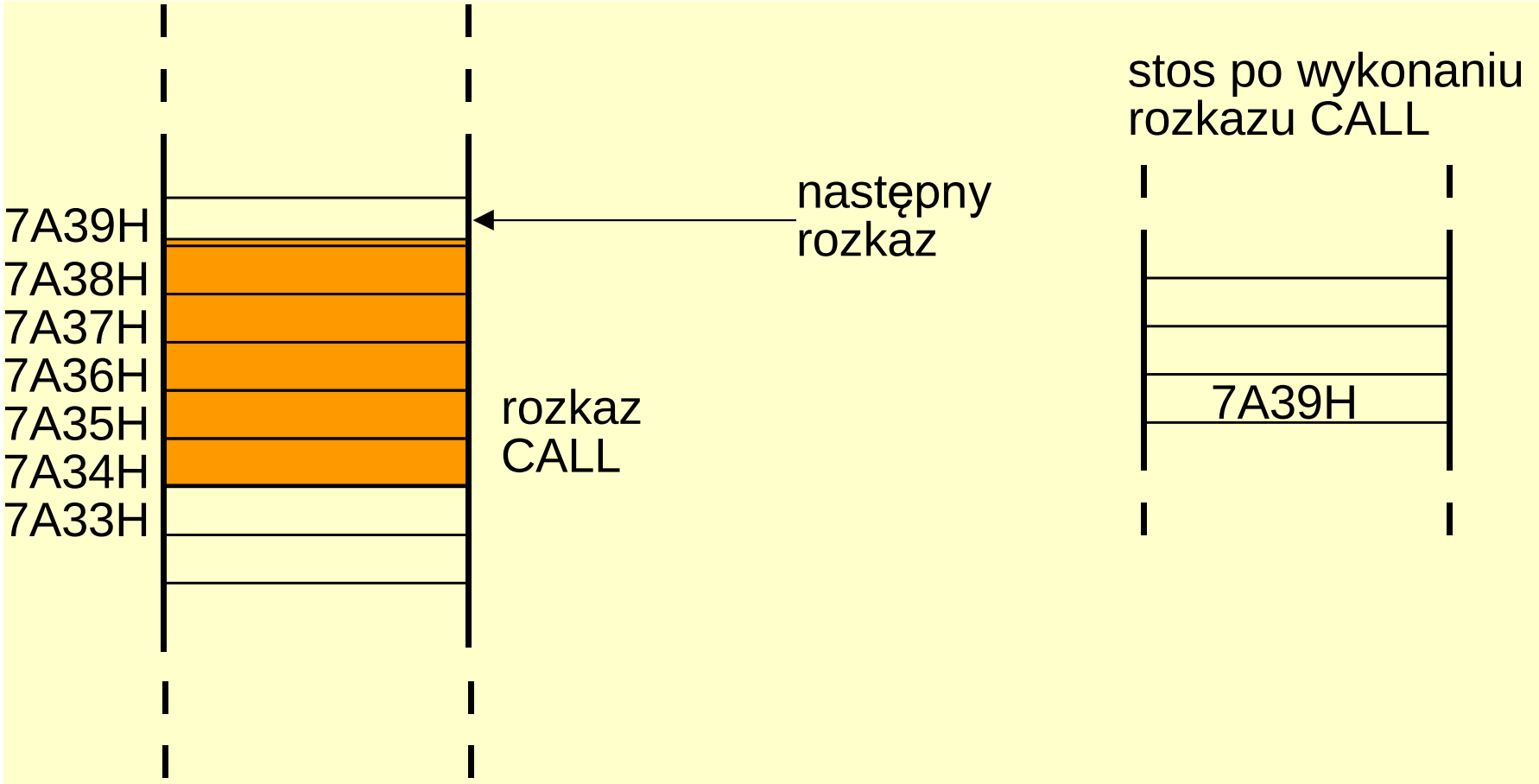
- W procesorach zgodnych z architekturą x86 adres powrotu zapisuje się na stosie.
- Spotyka się inne typy procesorów (zwłaszcza klasy RISC), w których ślad zapisywany jest w rejestrach.
- W procesorach x86 wywołanie podprogramu realizuje rozkaz **CALL** stanowiący połączenie skoku bezwarunkowego z operacją zapamiętania śladu na stosie.
- Na końcu podprogramu umieszcza się rozkaz **RET**, który przekazuje sterowanie do programu głównego.

POLITECHNIKA GDAŃSKA

Rozkazy CALL i RET (2)

- W ujęciu technicznym rozkaz RET odczytuje liczbę z wierzchołka stosu i wpisuje ją do wskaźnika instrukcji EIP.
- Podobnie jak w przypadku skoków bezwarunkowych, dostępne są dwie odmiany rozkazu CALL:
  - wykonujące skok do podprogramu bezpośredni,
  - wykonujące skok do podprogramu pośredni.
- Rozkaz CALL typu pośredniego używany jest m.in. przez kompilatory języka C do implementacji wywoływania funkcji przez wskaźnik.

Rozkazy CALL i RET (3)



POLITECHNIKA GDAŃSKA

Przykład podprogramu w asemblerze (1)

- Podany podprogram **kwadrat** oblicza wartość wyrażenia

$$y = x^2 + 1$$

gdzie x jest liczbą całkowitą bez znaku.

Zakładamy, że wartość **x** została wpisana do rejestru ESI przed rozpoczęciem wykonywania podprogramu, a wynik obliczenia dostępny będzie w rejestrze EDI. Przyjmujemy też, że obliczona wartość **y** da się przedstawić w postaci liczby 32-bitowej bez znaku (w trakcie obliczeń nie wystąpi nadmiar).

Przykład … (2)

```
kwadrat  PROC
; kopiowanie zawartości rejestru ESI do rejestru EAX
         mov  eax, esi
; mnożenie zawartości rejestru EAX przez zaw.
  rejestru ESI –
; wynik mnożenia  (64-bitowy) wpisywany jest
; do rejestru EDX:EAX
         mul  esi
; kopiowanie zawartości rejestru EAX do rejestru EDI
; (bierzemy tylko młodszą część iloczynu)
         mov  edi, eax
; dodanie 1 do wyniku mnożenia
         add  edi, 1
         ret  ; powrót z podprogramu
kwadrat  ENDP
```

Przykład … (3)

- Przykładowe wywołanie podprogramu

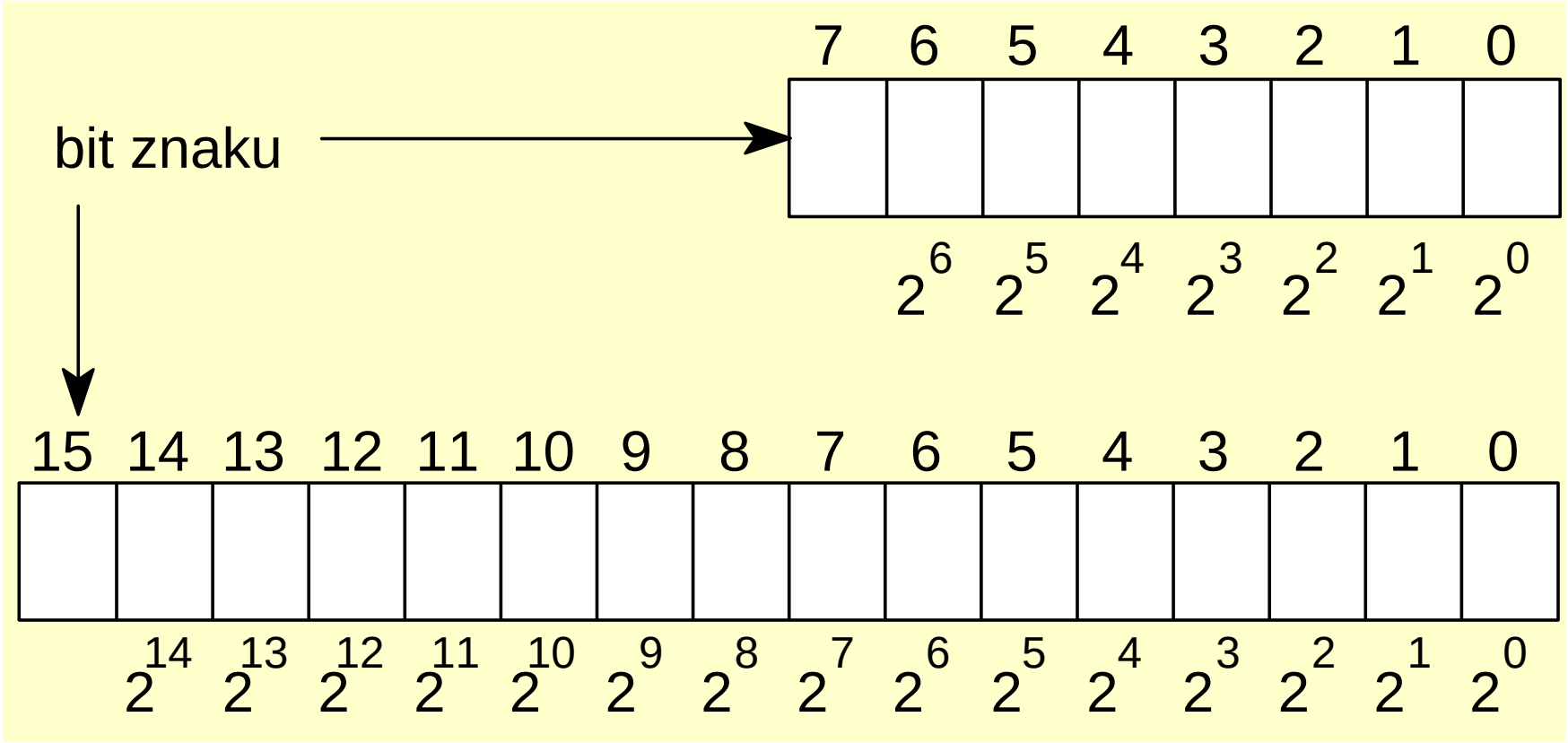
```
mov     esi, 7
call    kwadrat
```

Kodowanie liczb całkowitych

- W wielu współczesnych procesorach, w tym w procesorach zgodnych z architekturą x86, wyróżnia się:
  - liczby całkowite bez znaku, kodowane w naturalnym kodzie binarnym — liczby te omawiane były wcześniej;
  - liczby całkowite ze znakiem kodowane w kodzie U2;
  - liczby całkowite ze znakiem kodowane w kodzie znak–moduł.

POLITECHNIKA GDAŃSKA

Kodowanie liczb całkowitych ze znakiem



POLITECHNIKA GDAŃSKA

Kodowanie w systemie znak–moduł (1)

- W tym systemie kodowania skrajny lewy bit określa znak liczby, a pozostałe bity określają wartość bezwzględną liczby (moduł).
- Ten rodzaj kodowania stosowany jest nadal w arytmetyce zmiennoprzecinkowej.
- W operacjach stałoprzecinkowych kodowanie w systemie znak–moduł jest rzadko stosowane.

POLITECHNIKA GDAŃSKA

Kodowanie w systemie znak–moduł (2)

- Wartość liczby binarnej kodowanej w systemie *znak—moduł* określa poniższe wyrażenie (gdzie x_i oznacza wartość i-tego bitu liczby, m oznacza liczbę bitów rejestru lub komórki pamięci, zaś s stanowi wartość bitu znaku)

$$w = (-1)^s \cdot \sum_{i=0}^{m-2} x_i \cdot 2^i$$

POLITECHNIKA GDAŃSKA

Kodowanie w systemie U2 (1)

- Kodowanie liczb w systemie U2 jest obecnie powszechnie stosowane w wielu systemach komputerowych.
- Taki rodzaj kodowania upraszcza i przyśpiesza wykonywanie operacji arytmetycznych przez procesor.

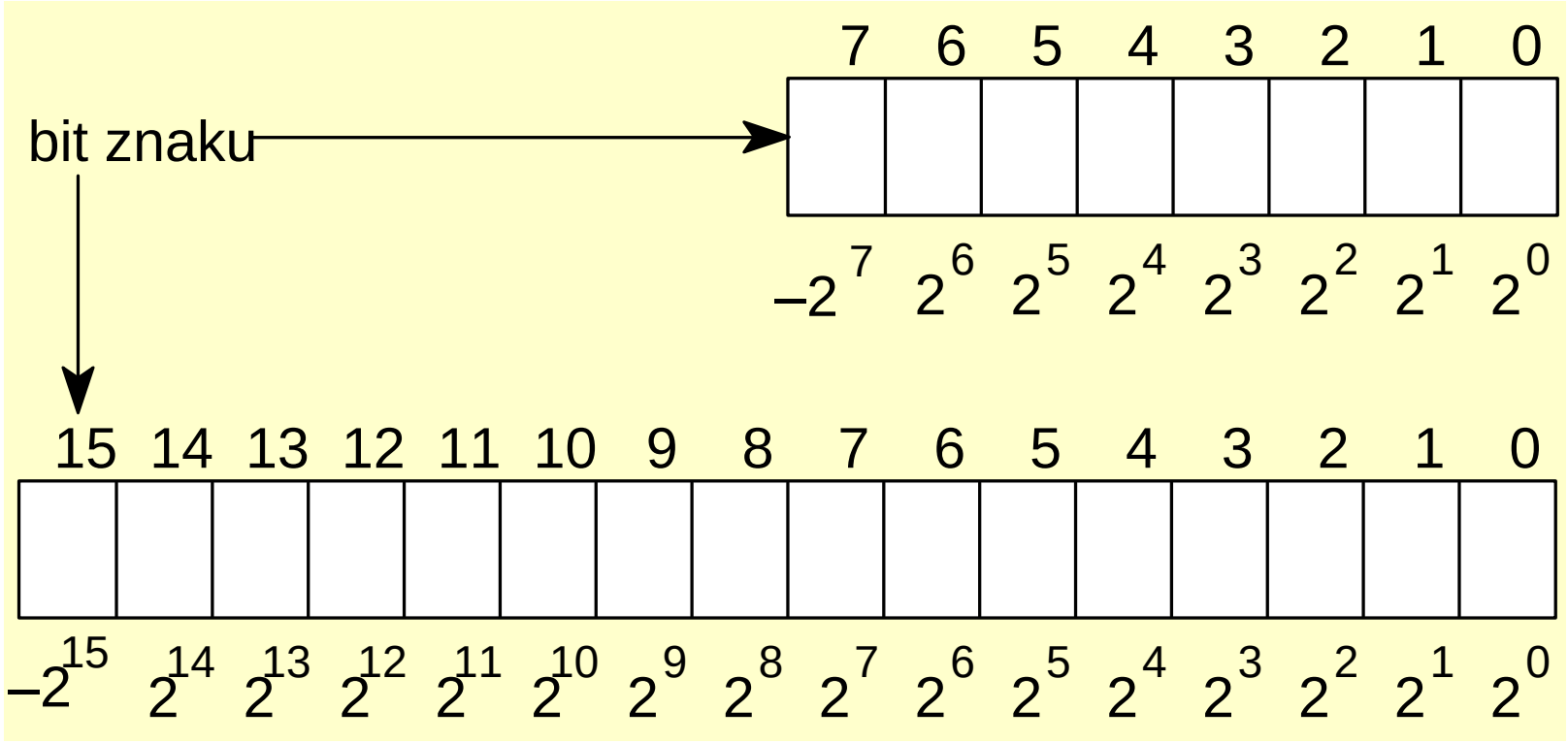
POLITECHNIKA GDAŃSKA

Kodowanie w systemie U2 (2)

- Zakresy liczb kodowanych w systemie U2:

liczby 8-bitowe: <–128, +127>
liczby 16-bitowe <–32768, +32767>
liczby 32-bitowe <–2 147 483 648, +2 147 483 647>
liczby 64-bitowe
<–9 223 372 036 854 775 808,
+9 223 372 036 854 775 807>

Kodowanie w systemie U2 (3)



Kodowanie w systemie U2 (4)

- Przykładowo, reprezentacja liczby –1 w kodzie U2 ma postać:

  8-bitowa: 1111 1111

  16-bitowa: 1111 1111 1111 1111

  32-bitowa: 1111 1111 1111 1111 1111 1111 1111 1111

- Wartość liczby binarnej kodowanej w systemie U2 określa poniższe wyrażenie (*m* oznacza liczbę bitów rejestru lub komórki pamięci)

$$w = -x_{m-1} \cdot 2^{m-1} + \sum_{i=0}^{m-2} x_i \cdot 2^i$$

Kodowanie w systemie U2 (5)

- Przykład: reprezentacja liczby –3 w kodzie U2

  8-bitowa: 1111 1101

  16-bitowa: 1111 1111 1111 1101

  32-bitowa: 1111 1111 1111 1111 1111 1111 1111 1101

- Przykład: liczba –2147483648 w postaci liczby binarnej 32-bitowej w kodzie U2

  1000 0000 0000 0000 0000 0000 0000 0000

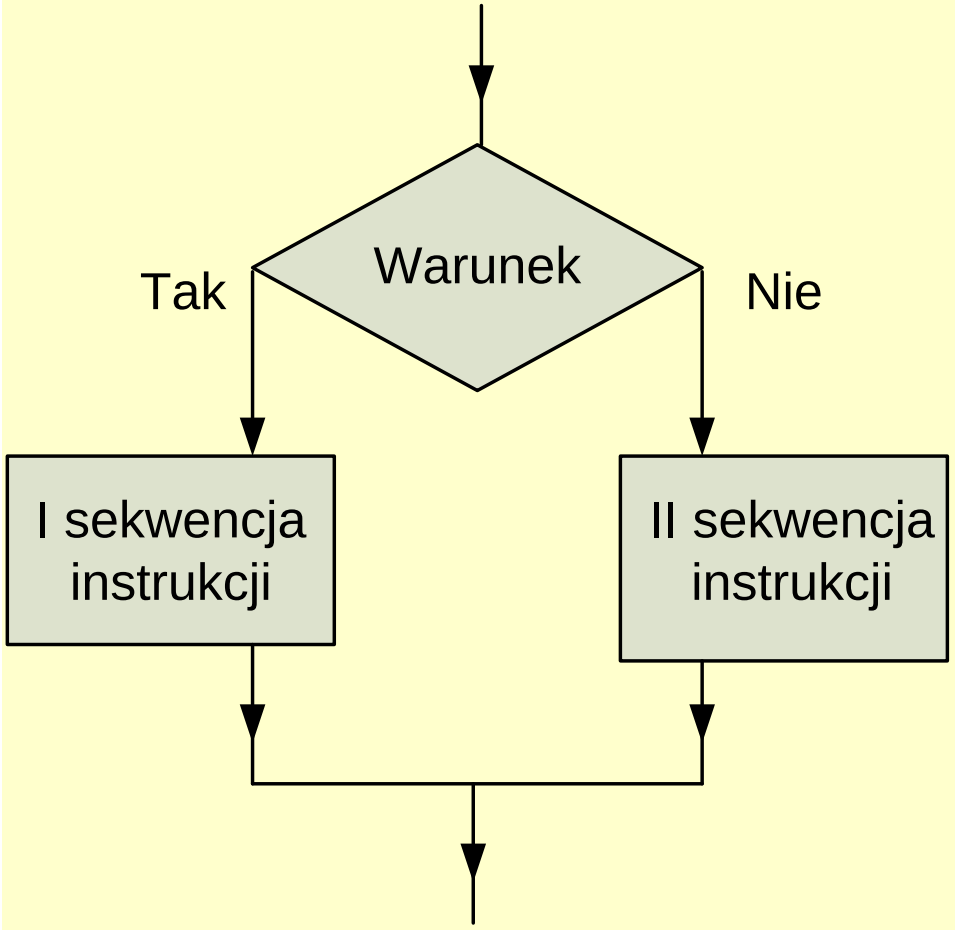
POLITECHNIKA GDAŃSKA

Implementacja struktur decyzyjnych (1)

- Prawie wszystkie algorytmy realizowane w komputerach zawierają struktury decyzyjne, które określają sposób dalszego wykonywania programu w zależności od wartości uzyskanych wyników pośrednich — struktury te mają postać rozwidleń i pętli.
- W tego rodzaju strukturach podstawowe znaczenie mają operacje porównania — trzeba zbadać która z dwóch wartości jest większa albo czy wartości są równe.

POLITECHNIKA GDAŃSKA

Implementacja struktur decyzyjnych (2)



- W językach wysokiego poziomu struktury decyzyjne realizowane są za pomocą instrukcji

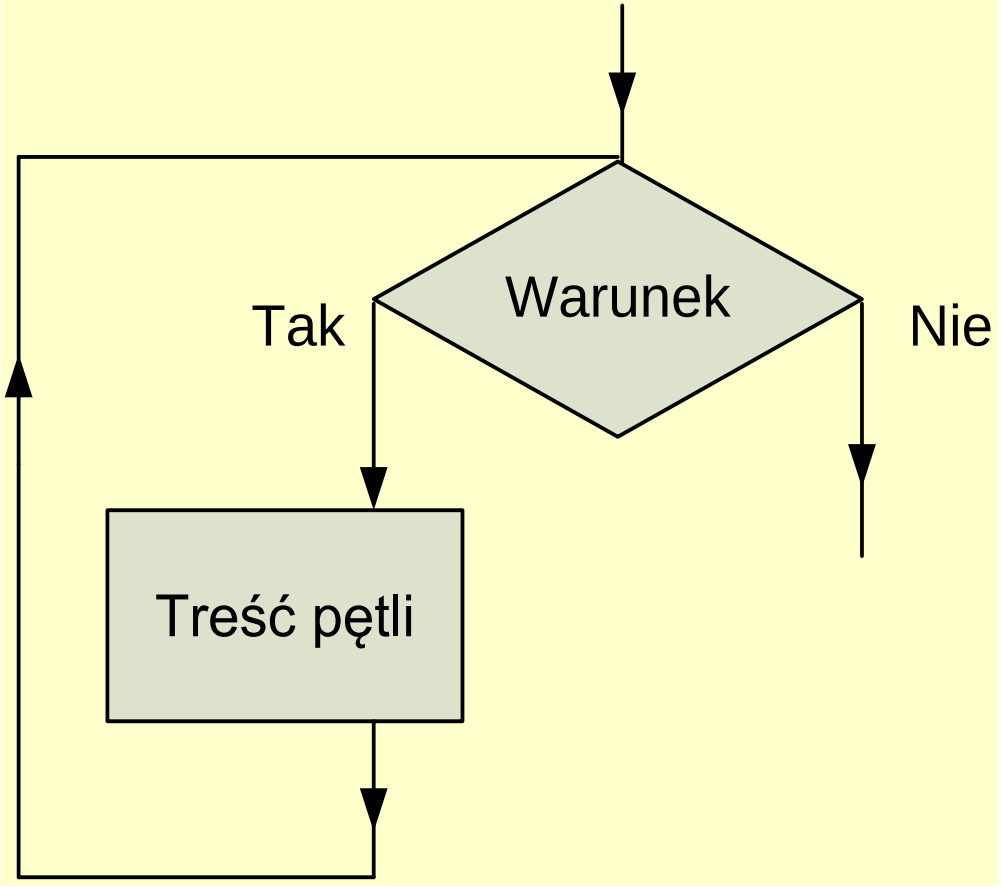
  if … then … else

POLITECHNIKA GDAŃSKA

Implementacja struktur decyzyjnych (3)

Z kolei pętle realizowane są za pomocą instrukcji

while (zob. rys. obok), **for, repeat … until**



POLITECHNIKA GDAŃSKA

Implementacja struktur decyzyjnych (4)

- W architekturze x86 do porównania używa się dwóch kolejnych rozkazów: pierwszy z nich wykonuje odejmowanie obu porównywanych wartości, drugi jest rozkazem skoku, który testuje własności wyniku odejmowania i ewentualnie przekazuje sterowanie („skacze”) do innego fragmentu programu.
- Zatem w zależności od wyniku porównania rozkazy programu mogą być wykonywane dalej w porządku naturalnym, albo procesor może ominąć pewną liczbę rozkazów i przejść (przeskoczyć) do wykonywania rozkazów znajdujących się w odległym miejscu pamięci.

POLITECHNIKA GDAŃSKA

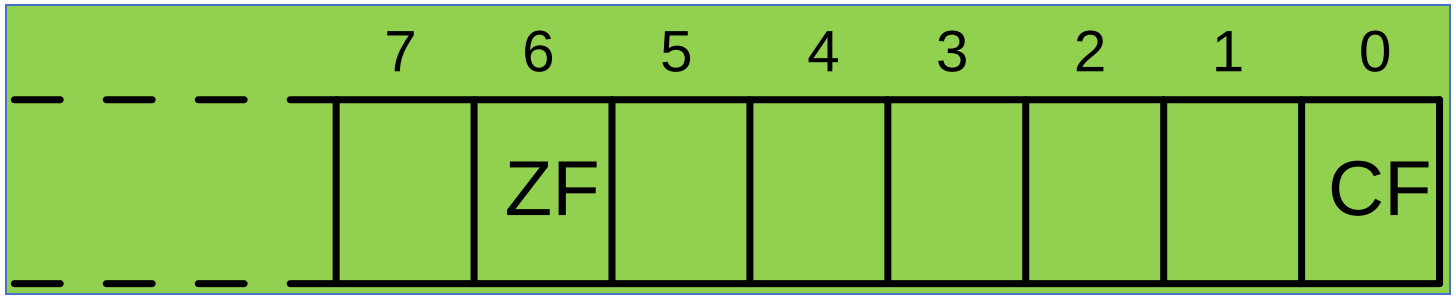
Porównywanie zawartości rejestrów i komórek pamięci (1)

- W architekturze x86 operacje porównania realizowane są poprzez odejmowanie porównywanych wartości i testowanie zawartości znaczników: CF, ZF, OF, SF.
- W operacjach porównywania liczb bez znaku istotne znaczenie mają (omawiane wcześniej) znaczniki CF i ZF:

POLITECHNIKA GDAŃSKA

Porównywanie zawartości rejestrów i komórek pamięci (2)

- CF znacznik przeniesienia, ustawiany w stan 1 w przypadku wystąpienia przeniesienia (przy dodawaniu) lub pożyczki (przy odejmowaniu)
- ZF znacznik zera, ustawiany w stan 1, jeśli wynik operacji arytmetycznej lub logicznej jest równy 0 — w przeciwnym razie znacznik ustawiany jest w stan 0



POLITECHNIKA GDAŃSKA

Porównywanie zawartości rejestrów i komórek pamięci (3)

- Procesor wykonał odejmowanie dwóch liczb bez znaku znajdujących się w rejestrach EBX i ECX (rozkaz sub ebx, ecx) – jakie wartości zostaną wpisane do znaczników CF i ZF?
  1. (EBX) > (ECX) ZF = 0 CF = 0
  2. (EBX) = (ECX) ZF = 1 CF = 0
  3. (EBX) < (ECX) ZF = 0 CF = 1

- Na podstawie stanu znaczników ZF i CF można określić, która z porównywanych liczb jest większa, albo czy liczby są równe.

Rozkazy SUB i CMP

- Zamiast rozkazu **SUB** używa się zazwyczaj rozkazu **CMP**, który również wykonuje odejmowanie, ustawia znaczniki, ale nigdzie nie wpisuje wyniku odejmowania — ułatwia to kodowanie programu, ponieważ po wykonaniu rozkazu CMP oba operandy pozostają niezmienione (rozkaz SUB wpisuje do pierwszego operandu wynik odejmowania).

POLITECHNIKA GDAŃSKA

Porównywanie liczb ze znakiem i bez znaku (1)

- Do testowania stanu znaczników używa się omawianych wcześniej rozkazów sterujących (skoków) warunkowych (np. ja, jnz, ...), przy czym do porównywania liczb ze znakiem (w kodzie U2) i liczb bez znaku stosuje się odrębne grupy rozkazów sterujących. Rozkazy te pokazane na następnych slajdach.
- Mnemoniki omawianych rozkazów zaczynają się od litery J (ang. jump), a kolejne litery mogą być następujące:

POLITECHNIKA GDAŃSKA

Porównywanie liczb ze znakiem i bez znaku (2)

- W rozkazach używanych przy porównywaniu liczb ze znakiem (w kodzie U2) i liczb bez znaku
  - E equal (równy)
  - N not (nie)
- W rozkazach używanych przy porównywaniu liczb bez znaku
  - A above (powyżej),
  - B below (poniżej)
- W rozkazach używanych przy porównywaniu liczb ze znakiem (w kodzie U2)
  - G greater (większy)
  - L less (mniejszy)

Rozkazy używane do porównywania liczb ze znakiem (w kodzie U2)

Mnemonik		Interpretacja (zastosowanie)	Testowany warunek
typowy	symbole równo-ważne		
JG		skocz, gdy większy	ZF = 0 i SF = OF
	JNLE	skocz, gdy niemniejszy i nierówny	
JGE		skocz, gdy większy lub równy	SF = OF
	JNL	skocz, gdy niemniejszy	
JL		skocz, gdy mniejszy	SF ≠ OF
	JNGE	skocz, gdy niewiększy i nierówny	
JLE		skocz, gdy mniejszy lub równy	ZF = 1 lub SF ≠ OF
	JNG	skocz, gdy niewiększy	
JO		skocz, gdy wystąpił nadmiar	OF = 1
JNO		skocz, gdy nie wystąpił nadmiar	OF = 0

Rozkazy używane do porównywania liczb bez znaku i ze znakiem (w kodzie U2)

Mnemonik		Interpretacja (zastosowanie)	Testowany warunek
typowy	symbole równo-ważne		
JE		skocz, gdy równy	ZF = 1
	JZ	skocz, gdy wynik = 0	
JNE		skocz, gdy nierówny	ZF = 0
	JNZ	skocz, gdy wynik ≠ 0	

Inne rozkazy używane do testowania stanu znaczników

JP	JPE	skocz, gdy liczba bitów parzysta	PF = 1
JNP	JPO	skocz, gdy liczba bitów nieparzysta	PF = 0
JS		skocz, gdy wynik ujemny	SF = 1
JNS		skocz, gdy wynik nieujemny	SF = 0

POLITECHNIKA GDAŃSKA

Przykład: wyszukiwanie liczby największej w tablicy liczb (bez znaku)

```
; adres pierwszego elementu tablicy został
; wcześniej wpisany do rejestru EBX
; liczba elementów tablicy została wcześniej
; wpisana do rejestru ECX
      dec   ecx          ; ECX ← ECX-1
      mov   eax, [ebx]
ptla: add   ebx, 4       ; EBX ← EBX+4
      cmp   eax, [ebx]
      jae   skocz        ; skocz gdy > lub =
      mov   eax, [ebx]
skocz:    sub   ecx, 1; ECX ← ECX-1
      jnz   ptla         ; skocz, gdy ECX ≠ 0
```

POLITECHNIKA GDAŃSKA

Rozkazy wyznaczania wartości logicznych (1)

- Podane wcześniej typy rozkazów sterujących warunkowych pozwalają na zmianę porządku wykonywania rozkazów programu w zależności od wyniku porównania.
- Czasami jednak przed wykonaniem skoku trzeba wykonać dodatkowe operacje — w takich przypadkach, za pomocą wybranego rozkazu z grupy **SET**... można zapisać wynik porównania w rejestrze 8-bitowym lub zmiennej.

POLITECHNIKA GDAŃSKA

Rozkazy wyznaczania wartości logicznych (2)

- Rozkazy grupy **SET**... mogą być też stosowane do kodowania programów, w których nie używa się rozkazów skoku (rozkazy skoku zmniejszają wydajność procesora — zagadnienia te omawiane są w drugiej części wykładu).
- Sekwencje rozkazów podane w lewej i w prawej kolumnie są funkcjonalnie równoważne.

```
cmp     ecx, edx
mov     al, 1
jae     zgadza_sie
dec     al
zgadza_sie:
```

```
cmp     ecx, edx
setae   al
```

POLITECHNIKA GDAŃSKA

Tworzenie kodu wykonywalnego programu (1)

- Przekształcenie kodu źródłowego w asemblerze na ciągi zero-jedynkowe zrozumiałe przez procesor realizowane jest w kilku etapach:

1. *asemblacja* polega na przekształceniu wierszy źródłowych programu na ciągi zerojedynkowe, jednak z pozostawieniem pewnej elastyczności umożliwiającej późniejsze dołączenie podprogramów bibliotecznych i innych modułów programowych; kod wygenerowany przez asembler jest określany jako kod w *języku pośrednim* (w systemie Windows jest zapisywany w pliku z rozszerzeniem .OBJ)

POLITECHNIKA GDAŃSKA

Tworzenie kodu wykonywalnego programu (2)

2. *konsolidacja* (lub linkowanie) polega na scaleniu różnych modułów programu, w tym kodu wytworzonego podczas asemblacji i podprogramów bibliotecznych do postaci pojedynczego programu, który zostaje zapisany w pliku (w systemie Windows z rozszerzeniem .EXE)
3. *ładowanie* stanowi ostatnią fazę translacji programu: system operacyjny wpisuje program do pamięci głównej (operacyjnej) i dokonuje niewielkich zmian kodu w celu dostosowania programu do aktualnego położenia w pamięci.

POLITECHNIKA GDAŃSKA

Tworzenie kodu wykonywalnego programu (3)

- Odpowiednikiem asemblacji w przypadku języków wysokiego poziomu jest kompilacja — kompilatory używane w systemie Windows generują również kod w języku pośrednim (plik .OBJ).

Asemblacja programów (1)

- Asemblacja realizowana jest dwuprzebiegowo: w każdym przebiegu czytany jest cały plik źródłowy (ściśle: moduł) od początku do końca.
- W pierwszym przebiegu asembler stara się wyznaczyć ilości bajtów zajmowane przez poszczególne rozkazy i dane; jednocześnie asembler rejestruje w *słowniku symboli* wszystkie pojawiające się definicje symboli (zmiennych i etykiet).
- W drugim przebiegu asembler tworzy kompletną wersję tłumaczonego programu określając adresy wszystkich rozkazów w oparciu o informacje zawarte w słowniku symboli.

Asemblacja programów (2)

- W procesie asemblacji programów istotną rolę odgrywa rejestr programowy (tj. definiowany przez asembler), zwany *licznikiem lokacji*.
- Licznik lokacji określa adres komórki pamięci operacyjnej, do której zostanie przesłany aktualnie tłumaczony rozkaz lub dana.
- Po załadowaniu rozkazu lub danej, licznik lokacji zostaje zwiększony o ilość bajtów zajmowanych przez ten rozkaz lub daną.
- Przed rozpoczęciem tłumaczenia pierwszego wiersza licznik lokacji zawiera 0.

Asemblacja programów (3)

- Jeśli asembler napotka wiersz zawierający definicję symbolu, to rejestruje go w słowniku symboli, jednocześnie przypisując temu symbolowi wartość równą aktualnej zawartości licznika lokacji; ponadto zapisywane są także atrybuty symbolu, jak np. byte, word, itp;

POLITECHNIKA GDAŃSKA

Przykład asemblacji fragmentu programu (1)

```
gamma   dw      ?
beta    db      ?
alfa    dd      74567H, 885678H, 789H
        dd      0A15FF3H, 89ABH
```

— — — — — — —

```
        mov     ebx, 12
; odjęcie liczby 74567H od EAX
        sub     eax, alfa
; wpisanie liczby 885678H do EDX
        mov     edx, alfa+4
; dodanie liczby 89ABH do ESI
        add     esi, alfa [ebx]+4
```

POLITECHNIKA GDAŃSKA

Przykład asemblacji fragmentu programu (2)

- Po wczytaniu segmentu danych (w którym zdefiniowano zmienne: gamma, beta, alfa) słownik symboli zawiera następujące pozycje

alfa	Dword	0003	_DATA
beta	Byte	0002	_DATA
gamma	Word	0000	_DATA

POLITECHNIKA GDAŃSKA

Przykład asemblacji fragmentu programu (3)

- Tłumaczenie rozkazu mov ebx, 12 nie wymaga zaglądania do słownika symboli — wystarczy tylko zamienić mnemonik mov na odpowiedni kod binarny, następnie określić kod binarny rejestru ebx oraz liczbę 12 przedstawić w postaci 32-bitowej liczby binarnej, co prowadzi do podanego niżej rozkazu 5-bajtowego

mov ebx, 12

BB	0C	00	00	00

POLITECHNIKA GDAŃSKA

Przykład asemblacji fragmentu programu (4)

- Tłumaczenie na postać binarną następnych rozkazów jest bardziej skomplikowane: w celu wyznaczenia pola adresowego asembler musi każdorazowo odszukiwać w słowniku symboli wartość przypisaną nazwie alfa — wartość licznika lokacji dla obszaru danych w chwili wystąpienia definicji zmiennej alfa wynosiła 3, zatem wartość przypisana nazwie alfa wynosi 3.

POLITECHNIKA GDAŃSKA

Przykład asemblacji fragmentu programu (5)

- W rezultacie asemblacja rozkazu sub eax, alfa prowadzi do uzyskania niżej pokazanego kodu 6-bajtowego.
- Należy zwrócić uwagę, że liczba 3 nie jest tu wartością zmiennej alfa, lecz jej adresem względem początku obszaru danych.

sub eax, alfa

2B	05	03	00	00	00

POLITECHNIKA GDAŃSKA

Przykład asemblacji fragmentu programu (6)

- Dwa następne rozkazy zostaną przetłumaczone na poniższą postać binarną.
- Zauważmy, że oba rozkazy mają identyczne pole adresowe (cztery ostatnie bajty).

mov edx, alfa + 4

8B	15	07	00	00	00

add esi, alfa [ebx] + 4

03	B3	07	00	00	00

POLITECHNIKA GDAŃSKA

Operacje na liczniku lokacji (1)

- Licznikiem lokacji można się także posługiwać w programie źródłowym – aktualna zawartość licznika lokacji jest reprezentowana przez symbol $
- Symbol $ może stanowić operand w wyrażeniach języka asembler, reprezentujący bieżącą lokację wewnątrz aktualnie tłumaczonego kodu.

Operacje na liczniku lokacji (2)

- *Przykład*: podany niżej dwubajtowy rozkaz jmp powoduje przejście do następnego rozkazu. Obok podano postać rozkazu po asemblacji.

  jmp $+2 (kod maszynowy: EBH 00H)

- Czasami tego rodzaju rozkazy umieszcza się w programie w celu wprowadzenia dodatkowego opóźnienia.

Operacje na liczniku lokacji (3)

- **Przykład:** wykorzystując symbol $ można łatwo wyznaczyć liczbę znaków łańcucha, np.:

```
blad_par  db     'Podano błędne parametry'
rozmiar = $  —  blad_par
- - - - - - - - - - - - - - - - - - - - - - - - - - - -
        mov        ecx, rozmiar
```

Operacje na liczniku lokacji (4)

- Wartość wyrażenia $ — blad_par obliczana jest w trakcie asemblacji programu, a nie w czasie jego wykonywania (co jest charakterystyczne dla języków wysokiego poziomu) — z tego powodu wyrażenia tego rodzaju nazywane są *wyrażeniami arytmetycznymi czasu translacji*.
- Także zmienna rozmiar jest *zmienną czasu translacji*, co oznacza, że dla zmiennej tej nie jest rezerwowany żaden obszar w programie wynikowym.

Operacje na liczniku lokacji (5)

- Innymi słowy zmienna rozmiar funkcjonuje jedynie w czasie translacji (asemblacji) programu.
- Zauważmy, że trakcie asemblacji instrukcji mov zmienna czasu translacji rozmiar traktowana jest jako liczba, a nie jako nazwa zmiennej (zdefiniowanej za pomocą dyrektywy DW).

POLITECHNIKA GDAŃSKA

Dyrektywa ORG

- Dyrektywa ORG umożliwia wpisanie do licznika lokacji potrzebnej wartości, np.: **ORG 100H**
- W polu operandu dyrektywy ORG można podać wyrażenie arytmetyczne czasu translacji, np. dyrektywa **ORG $+7** powoduje zwiększenie aktualnej zawartości licznika lokacji o 7.

POLITECHNIKA GDAŃSKA

Sprawozdanie z przebiegu translacji programu (1)

- W trakcie asemblacji tworzony jest plik z rozszerzeniem .LST zawierający obszerne sprawozdanie z jej przebiegu (tzw. listing asemblacji).
- Analogicznie w trakcie konsolidacji tworzony jest plik z rozszerzeniem .MAP

POLITECHNIKA GDAŃSKA

Sprawozdanie z przebiegu translacji programu (2)

- W pliku .LST podawane są kolejne wiersze programu źródłowego oraz wygenerowany na ich podstawie kod półskompilowany (ang. object code); podane są także nazwy i wartości wszystkich etykiet, zmiennych i symboli stosowanych w programie źródłowym; sygnalizowane są też ewentualne błędy.
- Poniżej podano przykładowy fragment tego pliku obejmujący trzy rozkazy programu.

POLITECHNIKA GDAŃSKA

Sprawozdanie z przebiegu translacji programu (3)

```
000000A6    8A 03      nowy:    mov al, [ebx]
000000A8    43                  inc  ebx
000000A9    3C 0A               cmp  al,10
```

- W końcowej części sprawozdania podany jest słownik symboli używanych w programie.

POLITECHNIKA GDAŃSKA

Kod asemblerowy w wersji AT&T (1)

- Omawiany wcześniej asembler został zaprojektowany przez firmę Intel dla wytwarzanych przez nią procesorów.
- Dostępne są także asemblery używające innych składni języka, spośród których najbardziej znana jest składnia AT&T stosowana w Linuksie.

POLITECHNIKA GDAŃSKA

Kod asemblerowy w wersji AT&T (2)

- Zasadnicze różnice między tymi asemblerami są następujące:
  - nazwy rejestrów poprzedzone są znakiem %
  - przesłania zapisywane są w postaci *skąd, dokąd*
  - prawie każdy rozkaz posiada jawnie zdefiniowany rozmiar operandów, określony przez ostatnią literę instrukcji, np.:

```
movl  (%ebx), %eax          (AT&T)
mov   eax, dword ptr [ebx]  (Intel)
```

POLITECHNIKA GDAŃSKA

Kod asemblerowy w wersji AT&T (3)

- Wartości bezpośrednie są poprzedzone znakiem $, np.

```
movl   $1, %ebx        (AT&T)
mov    ebx, 1          (Intel)
```

- Liczby w zapisie szesnastkowym poprzedzone są znakami 0x (tak jak w C)
- Segmenty programu są deklarowane poprzez nazwy, np. .text, .bss
- Część rozkazów procesora posiada odmienne mnemoniki, np. cbw (Intel) → cbtw (AT&T)
- Komentarze poprzedzone są znakiem #

POLITECHNIKA GDAŃSKA

Kod asemblerowy w wersji AT&T (4)

- Wyrażenia indeksowe stosują dość specyficzną notację:

```
movl   (%ebx), %eax
                        mov   eax, [ebx]
movl   3(%ebx), %eax
                        mov   eax, [ebx+3]
mov    (%ebx, %ecx), %eax
                        mov   eax, [ebx + ecx]
mov     (%ebx, %ecx, 2), %eax
                        mov   eax, [ebx + ecx*2]
```

POLITECHNIKA
GDAŃSKA

HISTORIA MĄDROŚCIĄ
PRZYSZŁOŚĆ WYZWANIEM