Architektura komputerów

sem. zimowy 2014/2015
cz. 1

Andrzej Jędruch

Literatura (1)

- Tanenbaum A.S.: Strukturalna organizacja systemów komputerowych, wyd. Helion
- Null L., Lobur J.: Struktura organizacyjna i architektura systemów komputerowych. Wyd. Helion 2004.
- Lewis D.: Między asemblerem a językiem C, wyd. RM

Literatura (2)

- Chalk B.S.: Organizacja i architektura komputerów. Warszawa WNT 1998
- Stallings W.: Organizacja i architektura systemu komputerowego. Warszawa WNT 2000
- Petzold C.: Kod, wyd. WNT

Literatura (3)

- Schmit L.: Procesory Pentium. Narzędzia optymalizacji. Warszawa wyd. Mikom 1997.
- Biernat J.: Architektura komputerów. Wrocław 2013. Oficyna Wydawnicza Politechniki Wrocławskiej.
- Wróbel E. i in.: Praktyczny kurs asemblera. Wyd. Helion 2004.

Literatura (4)

- Abel P.: Asembler IBM PC programowanie, wyd. RM 2004.

Rozwój konstrukcji komputerów i oprogramowania (1)

- Kalkulatory elektromechaniczne: Bell 1930, Zuse 1941 (arytmetyka zmiennoprzecinkowa)
- 1942 – 1946 pierwsze komputery elektroniczne
- **1945 koncepcje von Neumanna**
- 1948 Opracowanie tranzystora
- 1949 Rozwój oprogramowania: biblioteki podprogramów, asembler
- 1951 Komputer EDVAC (von Neumann) — program przechowywany w pamięci

Rozwój konstrukcji komputerów i oprogramowania (2)

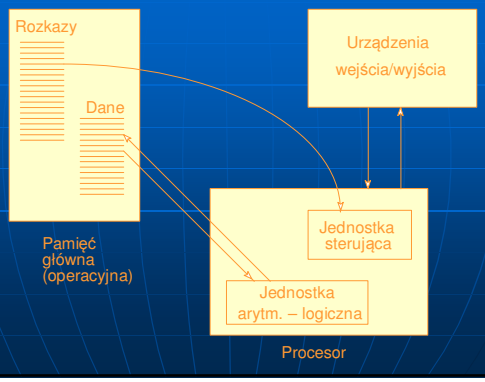
- 1954 język programowania FORTRAN
- 1955 pierwszy komputer tranzystorowy
- 1959 komputer PDP–1
- 1969 System Unix
- lata 70 minikomputery
- 1972 język C
- 1981 komputer IBM PC (16 KB RAM)
- 1990 system Windows 3.0
- 2004 procesory wielordzeniowe
- 2007-2008 procesory Phenom (AMD), Core i7 (Intel)

Rozwój konstrukcji komputerów i oprogramowania (3)

- 1958 komputer XYZ (Zakład Aparatów Matematycznych PAN)
- 1960 komputer ZAM 2
- 1960 język programowania SAKO
- 1964 komputery serii ZAM 21
- 1972 komputery serii ODRA i RIAD (Elwro Wrocław)
- 1975 komputer Momik
- 1985 różne mikrokomputery oparte na procesorze 8080

Komputer ZAM 41 eksploatowany w latach 1969 – 1976 (sala EA 508)

Model komputera wg von Neumanna (1)



Model komputera wg von Neumanna (2)

- Mimo upływu wielu lat prawie wszystkie współczesne komputery ogólnego przeznaczenia stanowią realizację tego modelu.
- Zasadniczą i centralną część każdego komputera stanowi **procesor** — jego własności decydują o pracy całego komputera.
- Komunikacja ze światem zewnętrznym realizowana jest za pomocą urządzeń wejścia/wyjścia.

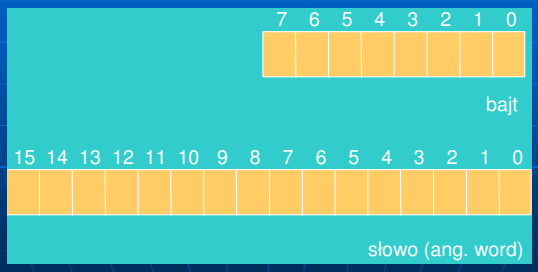
Model komputera wg von Neumanna (3)

- Procesor steruje podstawowymi operacjami komputera, wykonuje operacje arytmetyczne i logiczne, przesyła i odbiera sygnały, adresy i dane z jednego podzespołu komputera do drugiego.
- Procesor pobiera kolejne instrukcje (rozkazy) programu i dane z pamięci głównej (operacyjnej) komputera, przetwarza je i ewentualnie odsyła wyniki do pamięci. W każdej chwili procesor wykonuje dokładnie jedną instrukcję (rozkaz).
- Sposób przechowywania danych i instrukcji jest identyczny — zapisany kod nie pozwala odróżnić instrukcji od danych.

Elementy elektroniczne współczesnych komputerów

- Do budowy współczesnych komputerów używane są elementy elektroniczne — inne rodzaje elementów (np. mechaniczne) są znacznie wolniejsze (o kilka rzędów).
- Ponieważ elementy elektroniczne pracują pewnie i stabilnie jako elementy dwustanowe, informacje przechowywane i przetwarzane przez komputer mają postać ciągów zerojedynkowych.

Bity, bajty, słowa, ... (1)



Bity, bajty, słowa, ... (2)

- Tworzone są zespoły bajtów:
  - 16-bitowe *słowa*,
  - 32-bitowe *podwójne słowa*,
  - 64-bitowe *poczwórne słowa*,
  - w miarę potrzeby tworzy się także większe zespoły bajtów;
- Producenci procesorów ustalają konwencję numeracji bitów w bajtach i słowach — numeracja przyjęta m. in. w procesorach firmy Intel pokazana jest na rysunku.

Pamięć główna (operacyjna) (1)

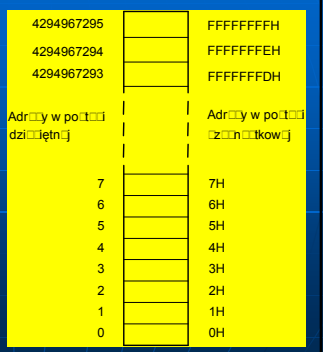
- Pamięć główna (operacyjna) komputera składa z dużej liczby komórek (np. kilku miliardów), a każda komórka utworzona jest z pewnej liczby bitów.
- Gdy komórkę pamięci tworzy 8 bitów, to mówimy, że *pamięć ma organizację bajtową* — taka organizacja jest typowa dla większości współczesnych komputerów.
- Poszczególne komórki mogą zawierać dane, na których wykonywane są obliczenia, jak również mogą zawierać rozkazy (instrukcje) dla procesora.

Pamięć główna (operacyjna) (2)

- Poszczególne bajty (komórki) pamięci są ponumerowane od 0 — numer komórki pamięci nazywany jest jej *adresem fizycznym*.
- Adres fizyczny przekazywany jest przez procesor (lub inne urządzenie) do podzespołów pamięci w celu wskazania położenia bajtu, który ma zostać odczytany lub zapisany.
- Zbiór wszystkich adresów fizycznych nazywa się *fizyczną przestrzenią adresową*.

Pamięć główna (operacyjna) (3)

- W wielu procesorach adresy fizyczne są 32-bitowe, co określa od razu maksymalny rozmiar zainstalowanej pamięci: 2^{32} = 4 294 967 296 bajtów (4 GB)



Pamięć główna (operacyjna) (4)

- Do adresowania pamięci niezbędna jest określona liczba linii adresowych skojarzonych z bitami rejestru adresowego.
- Aktualnie wytwarzane procesory 64-bitowe pozwalają na dostęp do 1 TB pamięci.
- Wersje 64-bitowe systemu MS Windows adresują 16 TB pamięci.

Rozmiar pamięci	Liczba bitów potrzebnych do adresowania
4 GB	32
64 GB	36
1 TB (1 terabajt)	40
16 TB	44
256 TB	48
4 petabajty	52

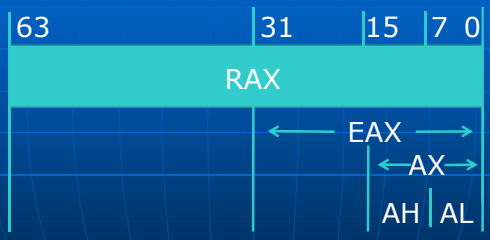
Rejestry ogólnego przeznaczenia (1)

- W trakcie wykonywania obliczeń często wyniki pewnych operacji stają się danymi dla kolejnych operacji — w takim przypadku nie warto odsyłać wyników do pamięci operacyjnej, a lepiej przechować te wyniki w komórkach pamięci wewnątrz procesora — komórki te mają zazwyczaj postać rejestrów, które oznaczane są symbolami literowymi.
- W komputerach występuje także odrębny rodzaj pamięci określany jako **pamięć podręczna** (ang. cache memory) — temat ten będzie omawiany później.

Rejestry ogólnego przeznaczenia (2)

- We wczesnych wersjach procesorów rodziny 8086 firmy Intel używano rejestrów 8-bitowych (np. AL, AH, BL, ...) i 16-bitowych (np. AX, BX, CX, ...).
- Później rozszerzono te rejestry do 32 bitów nadając im nazwy EAX, EBX,..
- Z kolei, w ostatnich latach rejestry 32-bitowe rozszerzono do 64 bitów nadając im nazwy RAX, RBX, ...
- Na kolejnych dwóch rysunkach przedstawiono współzależności rejestrów 8-, 16-, 32- i 64-bitowych na przykładzie rejestrów RAX i RBX.

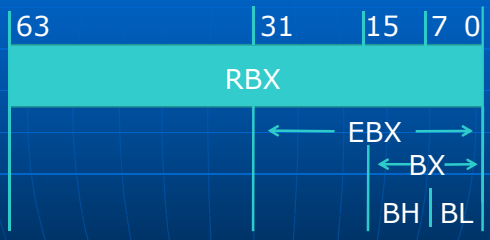
Rejestry ogólnego przeznaczenia (3)



Rejestry ogólnego przeznaczenia (4)

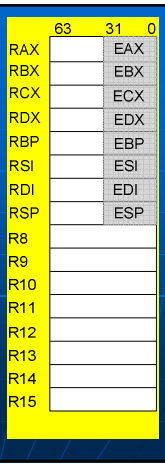
- Zauważmy, że rejestr AL stanowi 8 najmłodszych bitów rejestru RAX, a rejestr AX stanowi 16 najmłodszych bitów rejestru RAX.
- Zbliżoną lub identyczną strukturę mają pozostałe rejestry (RCX, RDX, ...)

Rejestry ogólnego przeznaczenia (5)



Rejestry ogólnego przeznaczenia (6)

Rejestry R8, R9, …, R15 dostępne są tylko w trybie 64-bitowym.



Reprezentacja danych w pamięci komputera (1)

- Współczesne komputery przetwarzają dane reprezentujące wartości liczbowe, teksty, dane opisujące dźwięki i obrazy i wiele innych.
- Dane o charakterze nieliczbowym (np. znaki, sygnały) muszą być zapisane (zakodowane) w postaci liczb — w rezultacie wszelkie dane przechowywane i przetwarzane w komputerze mają postać ciągów złożonych z zer i jedynek.

Reprezentacja danych w pamięci komputera (2)

- Dane liczbowe przedstawiane są w różnych formatach, ale z punktu widzenia działania procesora szczególne znaczenie mają liczby całkowite — opisy techniczne wielu rozkazów procesora odnoszą się bowiem do operacji wykonywanych na liczbach całkowitych kodowanych w naturalnym kodzie binarnym (liczby bez znaku) albo do operacji wykonywanych na liczbach całkowitych binarnych ze znakiem.

Reprezentacja danych w pamięci komputera (3)

- Nie oznacza to, że rozkazy procesora nie mogą wykonywać działań na liczbach ułamkowych — programista może odpowiednio dostosować algorytm obliczeń w taki sposób, ażeby działania na ułamkach zostały zastąpione przez działania na liczbach całkowitych

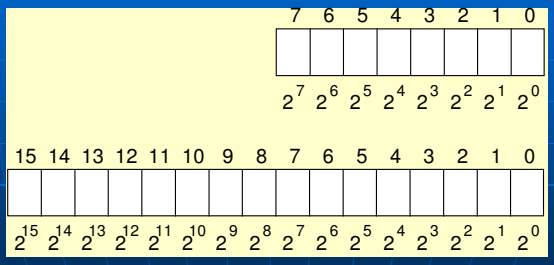
Reprezentacja danych w pamięci komputera (4)

- Ponadto procesor, i stowarzyszony z nim koprocesor arytmetyczny, posiadają zdolność wykonywania działań na liczbach w formacie zmiennoprzecinkowym (zmiennopozycyjnym); liczby te kodowane są w specyficzny sposób — zagadnienia związane z liczbami zmiennoprzecinkowymi omawiane będą w dalszej części wykładu.

Kodowanie liczb całkowitych

- W wielu współczesnych procesorach wyróżnia się wiele formatów liczb — na razie rozpatrzymy tylko liczby całkowite bez znaku, kodowane w naturalnym kodzie binarnym.
- Numeracja bitów i przyporządkowanie wag dla liczb 8- i 16-bitowych stosowana w procesorach Intel i AMD pokazana jest na rysunku.

Kodowanie liczb całkowitych bez znaku (1)



Kodowanie liczb całkowitych bez znaku (2)

- Wartość liczby binarnej bez znaku (*m* oznacza liczbę bitów rejestru lub komórki pamięci) określa wyrażenie

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

gdzie x_i oznacza wartość i-tego bitu liczby.

Przykłady kodowania liczb bez znaku

- liczba 253 w różnych formatach:
  8-bitowa: 1111 1101
  16-bitowa: 0000 0000 1111 1101
  32-bitowa:
  0000 0000 0000 0000 0000 0000 1111 1101
- liczba 2 007 360 447 w postaci 32-bitowej liczby binarnej:
  0111 0111 1010 0101 1110 0011 1011 1111

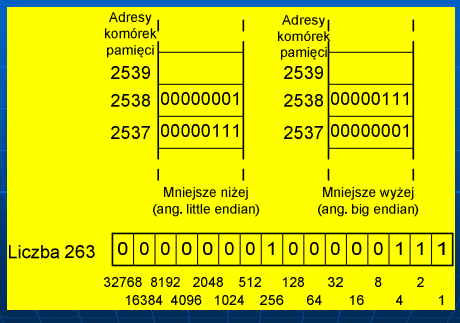
Zakresy liczb całkowitych bez znaku

- liczby 8-bitowe: <0, 255>
- liczby 16-bitowe <0, 65535>
- liczby 32-bitowe <0, 4 294 967 295>
- liczby 64-bitowe <0, 18 446 744 073 709 551 615>
  (*osiemnaście trylionów*
  czterysta czterdzieści sześć biliardów
  siedemset czterdzieści cztery biliony
  siedemdziesiąt trzy miliardy
  siedemset dziewięć milionów
  pięćset pięćdziesiąt jeden tysięcy
  sześćset piętnaście)
- 1 trylion = 10^{18}

Przechowywanie liczb w pamięci komputera (1)

- Liczby występujące w programach często przekraczają 255 i muszą być zapisywane na dwóch, czterech lub na większej liczbie bajtów.
- W systemach komputerowych przyjęto dwa podstawowe schematy rozmieszczenia poszczególnych bajtów liczby w pamięci:
  - *mniejsze niżej* (ang. little endian)
  - *mniejsze wyżej* (ang. big endian)

Przechowywanie liczb ... (2)

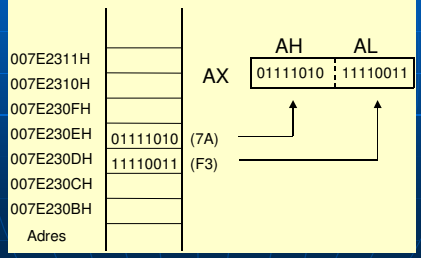


Przechowywanie liczb … (3)

- Format *little endian* stosowany jest m.in. w procesorach rodziny x86/64 (AMD/Intel), VAX, Alpha.
- Format *big endian* stosowany jest m.in. w procesorach Motorola 680x0, SunSPARC i większości procesorów klasy RISC.
- Procesor PowerPC udostępnia oba tryby pracy — w rejestrze MSR (Machine Status Register) wprowadzono dwa bity, z których pierwszy określa stosowaną kolejność bajtów dla procesora działającego w trybie systemu operacyjnego (ang. kernel mode), drugi bit określa aktualną kolejność dla zwykłego programu.

Przesyłanie liczb z pamięci do rejestru (i odwrotnie)

Przykład przesłania 16-bitowej liczby 7AF3H (=31475) z pamięci głównej do rejestru AX (architektura IA–32)



Rozpoznawanie formatu mniejsze niżej/wyżej (1)

- Identyfikację schematu reprezentacji liczb stosowanego w danym komputerze można przeprowadzić za pomocą niżej podanego fragmentu programu w języku C.
- W podanym przykładzie założono, że wartości typu int są 32-bitowe.

Rozpoznawanie formatu mniejsze niżej/wyżej (2)

```
unsigned int liczba = 0x12345678;

unsigned char * wsk = (unsigned char *) & liczba;

if (wsk[0] == 0x12)
   printf ("\nFormat mniejsze wyżej (big endian)");
else
   printf ("\nFormat mniejsze niżej (little endian)");
```

Wyrównywanie danych w pamięci głównej (operacyjnej)

- Dane liczbowe przechowywane w pamięci komputera mają długość 1, 2, 4, … bajtów. W przypadku danych o rozmiarze 2, 4, … bajtów pożądane jest by znajdowały się one w pamięci pod adresem podzielnym przez ich długość liczoną w bajtach — takie ulokowanie danych pozwala na uzyskanie najszybszego dostępu do nich.
- Jeśli adres danej jest podzielny przez jej długość (np. liczba 4-bajtowa została zapisana w pamięci pod adresem 456), to mówimy, że stosowane jest ***wyrównanie naturalne***.

Kodowanie znaków (1)

- Współczesne komputery posiadają zdolność przechowywania i przetwarzania danych tekstowych — niezbędne jest więc ustalenie sposobów kodowania znaków używanych w tekstach w postaci odpowiednich ciągów zer i jedynek.
- Podobny problem kodowania pojawił się kilkadziesiąt lat wcześniej w komunikacji telegraficznej (dalekopisowej) — opracowano wówczas różne schematy kodowania znaków w postaci ciągów zer i jedynek.

Kodowanie znaków (2)

- W tej sytuacji w systemach komputerowych przyjęto kod opracowany w pierwszej połowie XX w. dla urządzeń dalekopisowych — około roku 1968 w USA ustalił się sposób kodowania znaków znany jako kod ASCII (ang. American Standard Code for Information Interchange).
- Kod ten obejmuje małe i wielkie litery alfabetu łacińskiego, cyfry, znaki przestankowe i sterujące (np. nowa linia)

Kod ASCII (1)

- Znaki w kodzie ASCII zapisywane są na 8 bitach, ale znaki *podstawowego kodu ASCII* (alfabet łaciński, znaki przestankowe i sterujące) wykorzystują tylko kody o wartościach 0 ÷ 127, spośród dopuszczalnego przedziału 0 ÷ 255.

A	0100 0001	41H
B	0100 0010	42H
C	0100 0011	43H
D	0100 0100	44H
E	0100 0101	45H
F	0100 0110	46H
—	—	—
Y	0101 1001	59H
Z	0101 1010	5AH

a	0110 0001	61H
b	0110 0010	62H
c	0110 0011	63H
d	0110 0100	64H
e	0110 0101	65H
f	0110 0110	66H
—	—	—
y	0111 1001	79H
z	0111 1010	7AH

Kod ASCII (2)

0	0011 0000	30H
1	0011 0001	31H
2	0011 0010	32H
3	0011 0011	33H
—	—	—
8	0011 1000	38H
9	0011 1001	39H

!	0010 0001	21H
"	0010 0010	22H
#	0010 0011	23H
$	0010 0100	24H
—	—	—
{	0111 1011	7BH
|	0111 1100	7CH

Kod ASCII (3)

- Na bazie *podstawowego kodu ASCII* (kody o wartościach 0 ÷ 127) zaprojektowano wiele *kodów rozszerzonych*, w których wykorzystano także kody o wartościach z przedziału 128 ÷ 255
- W kodach rozszerzonych pierwsze 128 pozycji jest identyczne, jak w podstawowym kodzie ASCII, a następne 128 pozycji zawiera znaki alfabetów narodowych, symbole matematyczne, itp.

Kod ASCII (4)

- Istnieje wiele kodów rozszerzonych ASCII; w Polsce najbardziej znane są:
  - Windows 1250 (Microsoft CP 1250)
  - ISO 8859–2
  - Latin 2
  - Mazovia (wyszedł z użycia)

Kod ASCII (5)

Standard kodowania	Znak			
	a	ą	A	Ą
kody znaków podano w zapisie szesnastkowym				
Latin 2	61	A5	41	A4
Windows 1250	61	B9	41	A5
ISO 8859-2	61	B1	41	A1
Mazovia	61	86	41	8F
Unicode: format UTF – 16 (mniejsze niżej / little endian)	61 00	05 01	41 00	04 01
Unicode: format UTF – 16 (mniejsze wyżej / big endian)	00 61	01 05	00 41	01 04
Unicode: format UTF–8	61	C4 85	41	C4 84

Unicode (1)

- Kodowanie znaków na 8 bitach okazało się niewystarczające do przechowywania znaków narodowych krajów europejskich, i tym bardziej niewystarczające dla alfabetów krajów dalekiego wschodu — pojawiła się konieczność kodowania na większej liczbie bitów.
- Prace nad wprowadzeniem uniwersalnego zestawu podjęto na początku lat 90. ubiegłego stulecia — prace te podjęła zarówno Międzynarodowa Organizacja Normalizacyjna ISO, jak i konsorcjum zrzeszające producentów oprogramowania.

Unicode (2)

- Wynikiem prac obu instytucji było zdefiniowanie dwóch zestawów znaków:
  - UCS — Universal Character Set (ISO 10646).
  - Unicode (konsorcjum producentów).
- Znaki obu standardów są identyczne, ale obie instytucje wydają odrębne dokumenty, a także występują inne, drobne różnice.
- W dalszej części wykładu omawiany *uniwersalny zestaw znaków* określać będziemy terminem **Unikod** (lub Unicode).

Unicode (3)

- W systemie Unicode każdemu znakowi przypisana jest wartość liczbowa określanej jako ***punkt kodowy*** (ang. code point), przy czym dodatkowo każdemu znakowi przyporządkowana jest także nazwa, nie jest natomiast określony kształt drukowanego znaku. Przykładowo:
  - wielka litera **A** ma przypisany kod liczbowy, który zapisywany jest w postaci 0041 (szesnastkowo), a oficjalna nazwa brzmi „LATIN CAPITAL LETTER A”.
  - litera **ą** ma przypisany kod 0105, a oficjalna nazwa brzmi „LATIN SMALL LETTER A WITH OGONEK”.

Unicode (4)

- Wartości punktów kodowych dla znaków z podstawowego kodu ASCII (małe i wielkie litery alfabetu łacińskiego, cyfry, znaki przestankowe) są identyczne z wartościami odpowiednich kodów ASCII, aczkolwiek zazwyczaj zapisywane są w postaci liczb 16-bitowych, np. kod litery A w zapisie szesnastkowym ma postać:

ASCII 61

Unikod 0061

Unicode (5)

- Punkty kodowe Unikodu zapisywane są w postaci liczb złożonych z 4, 5 lub 6 cyfr w zapisie szesnastkowym.
- Dość często spotykany jest zapis, w którym wartość liczbowa poprzedzona jest znakami U+, co należy traktować jako informację, że wartość podana jest w zapisie szesnastkowym.
- Kody kilku początkowych liter alfabetu polskiego zawiera tabela (wartości podane są w kodzie szesnastkowym).

Znak	Kod
a	0061
A	0041
ą	0105
Ą	0104
b	0062
B	0042
c	0063
C	0043
ć	0107
Ć	0106

Reprezentacja punktów kodowych Unikodu w pamięci komputera (1)

- Przypuszcza się, że liczba różnych znaków, które używane są na świecie, wynosi ponad milion — wynika stąd konieczność przyjęcia sposobu kodowania tych znaków wykorzystujących co najmniej 21 bitów.
- Kierując się tym oszacowaniem, w standardzie Unikod udostępniono 1 114 112 punktów kodowych (w przedziale od 0 do 1 114 111). Punkty te tworzą *przestrzeń kodową* Unikodu.

Reprezentacja punktów kodowych Unikodu w pamięci komputera (2)

- Aktualnie, w najnowszej wersji standardu (6.0) zdefiniowano 109 384 znaków.
- Większość powszechnie używanych znaków jest przyporządkowana punktom kodowym o wartościach nie przekraczających 65 535 — zbiór ten, obejmujący kody od 0 do 65535, oznaczany jest skrótem BMP (ang. Basic Multilingual Plane).

Reprezentacja punktów kodowych Unikodu w pamięci komputera (3)

- Jeśli zamierzamy umieścić wartość punktu kodowego w pamięci komputera lub w pliku, to trzeba ustalić odpowiedni sposób kodowania dla używanego środowiska. Ponieważ niektóre wartości zajmują 21 bitów, wskazane byłoby zarezerwowanie na każdą wartość słowa 32-bitowego.
- Jeśli jednak uwzględnić podaną wyżej informację, że większość używanych znaków należy do zbioru BMP, gdzie punkty kodowe można zapisać na 16 bitach, to używanie słów 32-bitowych będzie powodować rozwlekłość kodowania.

Reprezentacja punktów kodowych Unikodu w pamięci komputera (4)

- Problem staje się jeszcze bardziej wyraźny, jeśli uwzględnić, że w zwykłych tekstach dominują litery alfabetu łacińskiego, które w kodzie ASCII zajmują 7 bitów.
- Z podanych powodów wprowadzono bardziej efektywne sposoby przechowywania znaków Unikodu w pamięci komputera — najczęściej używane są formaty UTF–8 i UTF–16 (Unicode Transformation Format).

UTF–8 (1)

- Przy zastosowaniu kodowania UTF–8 znaki podstawowe (znaki kodu ASCII z zakresu <1,127>) kodowane są jako 1-bajtowe, a inne znaki jako 2-bajtowe lub dłuższe.
- Przy kodowaniu UTF–8 obowiązują następujące reguły (wartości podane są w kodzie szesnastkowym):
  - Znaki Unikodu o kodach 0000 do 007F są kodowane jako pojedyncze bajty o wartościach z przedziału 00 do 7F. Oznacza to, że pliki zawierające wyłącznie znaki z podstawowego zestawu ASCII mają taką samą postać zarówno w kodzie ASCII jak i w UTF–8.

UTF–8 (2)

- Wszystkie znaki Unikodu o kodach większych od 007F są kodowane jako sekwencja kilku bajtów, z których każdy ma ustawiony najstarszy bit na 1. Pierwszy bajt w sekwencji kilku bajtów jest zawsze liczbą z przedziału C0 do FD i określa ile bajtów następuje po nim. Wszystkie pozostałe bajty zawierają liczby z przedziału 80 do BF.

UTF–8 (3)

- Podana niżej tablica określa sposób kodowania UTF–8 dla różnych wartości kodów znaków. Bity oznaczone xxx..xx zawierają reprezentację binarną kodu znaku.

Zakresy kodów		Liczba kodo-wanych bitów	Reprezentacja w postaci UTF-8
od	do		
0000	007F	7	0*xxxxxxx*
0080	07FF	11	110*xxxxx* 10*xxxxxx*
0800	FFFF	16	1110*xxxx* 10*xxxxxx* 10*xxxxxx*

UTF–8 (4)

- Podana tablica zawiera kody UTF–8 kilku początkowych liter alfabetu języka polskiego (wartości podano w postaci liczb szenastkowych):

Znak	Kod UTF-8
a	61
A	41
ą	C4 85
Ą	C4 84
b	62
B	42
c	63
C	43
ć	C4 87
Ć	C4 86

Przykład kodowania UTF–8

- Litera **ą** w Unikodzie ma przypisaną wartość:
  $(0105)_{16} = (0000\ 0\mathbf{0001\ 0000\ 0101})_2$
- Ponieważ wartość ta należy do przedziału <0080,07FF>, więc reprezentacja UTF–8 zostanie wyznaczona wg schematu:
  110*xxxxx* 10*xxxxxx*
- Zauważmy, że wszystkie liczby binarne z przedziału <0080,07FF> dają się zapisać na 11 bitach (w szczególności 07FF = 111 1111 1111) — zatem z podanej liczby $(0105)_{16}$ bierzemy tylko 11 najmłodszych bitów, które wstawiamy w miejsce symboli *xxx..xx* i otrzymujemy:
  110 00100 10 000101 (= C4 85)

UTF–16

- Kodowanie UTF–16 przeznaczone jest do reprezentacji znaków Unikodu w środowiskach lub kontekstach ukierunkowanych na słowa 16-bitowe.
- W przypadku znaków z grupy BMP (kody od 0H do FFFFH), kod UTF–16 jest identyczny z wartością punktu kodowego.
- Dla znaków z przedziału od 10000H do 10FFFFH (nie należących do BMP) stosuje się dwa słowa 16-bitowe (pomijamy tu szczegóły kodowania).

Procesory powszechnego użytku (1)

- We współczesnych komputerach osobistych stosuje się różne typy procesorów, z których większość wywodzi się z procesora 8086 firmy Intel (r. 1978).
- Procesor 8086 (lub jego odmiana 8088) został zastosowany w komputerze osobistym IBM (rok 1981); później, mniej więcej co 4 lata, pojawiały się jego coraz bardziej rozbudowane wersje: 80286, 80386, 486, różne wersje Pentium, …, Intel Core i7, … .
- Obok firmy Intel, procesory zgodne programowo z ww. produkuje także firma AMD.

Procesory powszechnego użytku (2)

- Dla wygody opisu omawiane procesory (firmy Intel i AMD) oznaczane są w literaturze jako procesory zgodne z architekturą **x86**, przy czym do oznaczania wersji 64-bitowych używa się symbolu **x86-64**. W odniesieniu do procesorów firmy Intel używa się też oznaczenia *Intel 32* dla procesorów 32-bitowych i *Intel 64* dla procesorów 64-bitowych.
- Charakterystyczną cechą tych procesorów jest kompatybilność wsteczna, co oznacza że każdy nowy model procesora realizuje funkcje swoich poprzedników.

Procesory powszechnego użytku (3)

- M.in. programy dla komputera IBM PC opracowane na początku lat osiemdziesiątych mogą być wykonywane także w komputerze wyposażonym w procesor Intel Core i5, jednak systemy operacyjne nie przewidują takiej możliwości — pozostaje wykonywanie programu za pomocą maszyny wirtualnej, np. DOSBox.

Lista rozkazów procesora (1)

- Procesor składa się z wielu różnych podzespołów wykonawczych, które wykonują określone działania (np. sumowanie liczb) — podzespoły te reprezentowane są przez jednostkę arytmetyczno–logiczną (ang. arithmetic logic unit).
- Podzespoły wykonawcze podejmują działania wskutek sygnałów otrzymywanych z jednostki sterującej.

Lista rozkazów procesora (2)

- Dla każdego typu procesora konstruktorzy ustalają pewien podstawowy zbiór operacji — każdej operacji przypisuje się ustalony kod w postaci ciągu zero-jedynkowego.
- Do zbioru operacji podstawowych należą zazwyczaj cztery działania arytmetyczne, operacje logiczne na bitach (negacja, suma logiczna, iloczyn logiczny), operacje przesyłania, operacje porównywania i wiele innych.

Lista rozkazów procesora (3)

- Operacje zdefiniowane w zbiorze podstawowym nazywane są ***rozkazami*** lub ***instrukcjami*** procesora; każdy rozkaz ma przypisany ustalony kod zero-jedynkowy, który ma postać jednego lub kilku bajtów o ustalonej zawartości.
- Podstawowy zbiór operacji procesora jest zwykle nazywany ***listą rozkazów procesora***.
- Ze względu na to, że posługiwanie się w procesie kodowania programu wartościami zero-jedynkowymi byłoby bardzo kłopotliwe, wprowadzono skróty literowe (tzw. mnemoniki) dla poszczególnych rozkazów (instrukcji) procesora.

Lista rozkazów procesora (4)

- Przekazanie procesorowi x86 rozkazu (instrukcji) w formie bajtu **01000010** spowoduje zwiększenie o 1 liczby umieszczonej w rejestrze EDX, natomiast przekazanie bajtu **01001010** spowoduje zmniejszenie tej liczby o 1.
- Podane tu operacje zapisuje się zazwyczaj w postaci symbolicznej, używając skrótów literowych opisujących funkcje rozkazu (mnemoników) i nazw rejestrów:
  - kod 01000010 zastępuje się przez
    INC EDX (ang. increment zwiększenie)
  - kod 01001010 zastępuje się przez
    DEC EDX (ang. decrement zmniejszenie)

Lista rozkazów procesora (5)

- Często polecenia przekazywane procesorowi składają się z kilku bajtów, np. bajty **10000000 11000111 00100101** traktowane są przez procesor jako polecenie dodania liczby 37 do liczby znajdującej się w rejestrze BH.
- Powyższa operacja zapisana w postaci symbolicznej ma postać:
  ADD BH, 37 (ang. addition dodawanie)
- Mnemoniki, nazwy rejestrów czy liczby dziesiętne są niezrozumiałe dla procesora i przed wprowadzeniem programu do pamięci muszą być zamienione na odpowiadające im kody zero-jedynkowe — programy dokonujące takiej konwersji nazywane są *asemblerami*

Wykonywanie programu w języku maszynowym przez procesor (1)

- W podanym ujęciu algorytm obliczeń przedstawiany jest za pomocą operacji ze zbioru podstawowego.
- Algorytm zakodowany jest w postaci sekwencji ciągów zero-jedynkowych zdefiniowanych w podstawowym zbiorze operacji procesora — tak zakodowany algorytm nazywać będziemy *programem w języku maszynowym.*

Wykonywanie programu w języku maszynowym przez procesor (2)

- Program w języku maszynowym przechowywany jest w pamięci głównej (operacyjnej) komputera.
- Wykonywanie programu polega na przesyłaniu kolejnych ciągów zero-jedynkowych z pamięci głównej do układu sterowania procesora.
- Zadaniem układu sterowania, po odczytaniu takiego ciągu, jest wygenerowanie odpowiedniej sekwencji sygnałów kierowanych do poszczególnych podzespołów wykonawczych , tak by w rezultacie wykonać wymaganą operację (np. dodawanie).

Język maszynowy a asembler

- Język maszynowy jest kłopotliwy w użyciu nawet dla specjalistów; znacznie wygodniejszy jest spokrewniony z nim język asemblera, który będzie omawiany dalej.
- W asemblerze kody bajtowe rozkazów zapisuje się w postaci skrótów literowych (mnemoników), np. ADD, DIV, MOV, LOOP, JMP, itd.; dostępnych jest wiele innych udogodnień, jak na przykład możliwość zapisu liczb w postaci dziesiętnej lub szesnastkowej.

Cykl rozkazowy (1)

- Wstępnie założymy, że sekwencja instrukcji tworzących program (w postaci ciągów zerojedynkowych) ułożona jest pamięci komputera w porządku naturalnym, tj. następna instrukcja umieszczona jest w pamięci bezpośrednio za poprzednią.
- W dalszej części wykładu pokażemy, że założenie to nie dotyczy struktur decyzyjnych (czyli instrukcji typu if .. then .. else i różnych typów pętli).

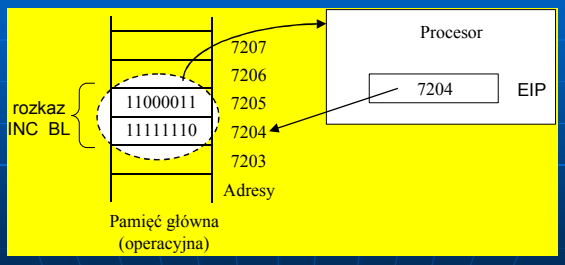
Cykl rozkazowy (2)

- Tak więc proces pobierania kolejnych instrukcji z pamięci operacyjnej i ich wykonywania musi być precyzyjnie zorganizowany, tak by natychmiast po wykonaniu kolejnej instrukcji procesor pobierał z pamięci następną.
- Aby pobrać tę instrukcję, procesor musi oczywiście znać jej położenie w pamięci głównej (operacyjnej) — informacje o położeniu kolejnej instrukcji są umieszczone w specjalnym rejestrze, nazywanym ***wskaźnikiem instrukcji.***

Cykl rozkazowy (2)

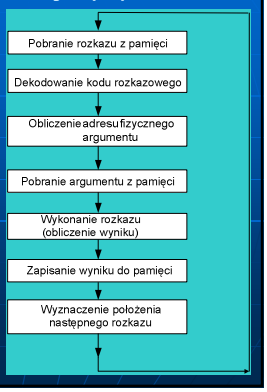
- W architekturze x86 używany jest 32-bitowy wskaźnik instrukcji oznaczony jest symbolem **EIP** (ang. extended instruction pointer).
- W architekturze x86-64 używany jest 64-bitowy rejestr RIP; w innych architekturach omawiany rejestr nazywany jest także *licznikiem rozkazów* lub *licznikiem programu* (PC – ang. program counter).

Cykl rozkazowy (3)



Cykl rozkazowy (4)

- Czynności wykonywane przez procesor w trakcie pobierania i wykonywania poszczególnych rozkazów powtarzane są cyklicznie, a cały proces nosi nazwę *cyklu rozkazowego.*
- Cykl rozkazowy jest podstawą działania wszystkich komputerów.



Rozkazy niesterujące (1)

- Podstawowe operacje przetwarzania: przesyłanie, operacje arytmetyczne i logiczne, przesunięcia, itd., realizują rozkazy (instrukcje) określane jako **niesterujące** .
- Charakterystyczną cechą tej klasy rozkazów jest to, że nie zmieniają one naturalnego porządku wykonywania rozkazów, co oznacza, że po wykonaniu rozkazu z tej klasy procesor rozpoczyna wykonywać rozkaz bezpośrednio przylegający w pamięci do rozkazu właśnie wykonanego.

Rozkazy niesterujące (2)

- W procesorach zgodnych z architekturą x86 kolejna zawartość rejestru EIP jest obliczana wg formuły:

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

Zapis asemblerowy typowych rozkazów niesterujących (1)

- Operacja przesłania zawartości komórki pamięci do rejestru realizowana przez rozkaz oznaczony skrótem literowym (mnemonikiem) **MOV**.
- Rozkaz ten ma dwa argumenty: pierwszy argument określa cel, czyli "*dokąd przesłać*", drugi zaś określa źródło, czyli "*skąd przesłać*" lub "*co przesłać*":

MOV dokąd przesłać , skąd (lub co) przesłać

Zapis asemblerowy typowych rozkazów niesterujących (2)

- Argumenty rozkazów wykonujących operacje dodawania ADD i odejmowania SUB zapisuje się podobnie jak argumenty rozkazu MOV

ADD dodajna , dodajnik

SUB odjemna , odjemnik

wynik wpisywany jest do obiektu wskazanego przez pierwszy argument

Zapis asemblerowy typowych rozkazów niesterujących (3)

- Rozkaz wykonuje operację na dwóch wartościach wskazanych przez pierwszy i drugi operand, a wynik wpisywany jest do pierwszego operandu
- Ogólnie, rozkaz postaci

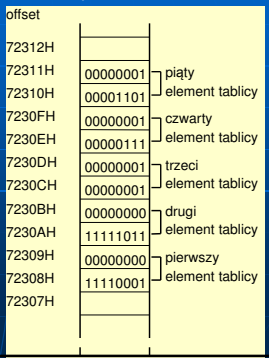
„operacja” cel, źródło

wykonuje działanie

cel ← cel „operacja” źródło

Przykład: sumowanie elementów tablicy (1)

- Przykładowa tablica zawiera pięć liczb binarnych 16-bitowych całkowitych bez znaku.
- Zakładamy, że w trakcie sumowania wszystkie wyniki pośrednie uzyskiwane w trakcie sumowania dadzą się przedstawić w postaci liczb 16-bitowych, tzn. nie wystąpi przepełnienie (nadmiar).



Przykład: sumowanie elementów tablicy (2)

- Operacje sumowania w postaci symbolicznej
  AX ← [72308H]
  AX ← AX + [7230AH]
  AX ← AX + [7230CH]
  AX ← AX + [7230EH]
  AX ← AX + [72310H]
- AX oznacza tu zawartość 16-bitowego rejestru AX
- Zapis [72308H] oznacza zawartość komórki pamięci znajdującej się w obszarze danych o adresie podanym w nawiasach kwadratowych.
- Litera H oznacza, że liczba podana jest w zapisie szesnastkowym.

Przykład: sumowanie elementów tablicy (3)

```
mov   ax, ds:[72308H]
add   ax, ds:[7230AH]
add   ax, ds:[7230CH]
add   ax, ds:[7230EH]
add   ax, ds:[72310H]
```

- Operacje sumowania w postaci sekwencji rozkazów procesora podanych w języku asemblera
- symbol ds: oznacza, że pobierane dane znajdują się w obszarze danych programu

Przykład: sumowanie elementów tablicy (4)

- Omawiane operacje w postaci zrozumiałej dla procesora wyglądają tak:
- 01100110 10100001 00001000 00100011 00000111 00000000
  (mov ax, ds:[72308H])
- 01100110 00000011 00000101 00001010 00100011 00000111 00000000
  (add ax, ds:[7230AH])

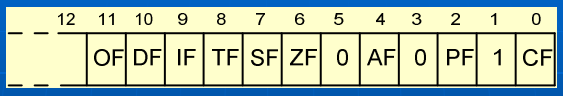
Przykład: sumowanie elementów tablicy (5)

- 01100110 00000011 00000101 00001100 00100011 00000111 00000000
  (add ax, ds:[7230CH])
- 01100110 00000011 00000101 00001110 00100011 00000111 00000000
  (add ax, ds:[7230EH])
- 01100110 00000011 00000101 00010000 00100011 00000111 00000000
  (add ax, ds:[72310H])

Rejestr stanu procesora (rejestr znaczników) (1)

- Niektóre rozkazy niesterujące, prócz właściwych czynności (np. odejmowania dwóch liczb) przekazują dodatkowe informacje opisujące własności wyniku operacji: czy wynik jest równy zero, czy wynik jest liczbą ujemną, czy wystąpiło przeniesienie, itp. — omawiane informacje wpisywane są do 32-bitowego **rejestru stanu procesora** (nazywanego także *rejestrem znaczników*).
- Cały rejestr stanu procesora można traktować jako zespół rejestrów 1- lub 2-bitowych, które określane są jako **znaczniki**.

Rejestr stanu procesora (2)



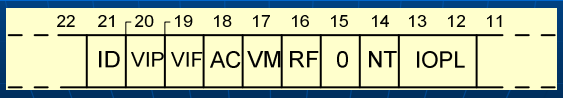
- Powyższy rysunek pokazuje fragment rejestru stanu procesora.
- Do znacznika zera ZF (ang. zero flag) wpisywana jest liczba 1, jeśli wynik operacji arytmetycznej lub logicznej wynosił 0 — w przeciwnym razie znacznik ZF jest zerowany.
- Znacznik CF ustawiany jest w stan 1, jeśli w trakcie dodawania wystąpiło przeniesienie, albo w trakcie odejmowania wystąpiła prośba o pożyczkę.

Rejestr stanu procesora (3)

- W miarę potrzeby stan znaczników może być testowany przez (omawiane dalej) *instrukcje sterujące* .
- Pozostałe bity rejestru stanu procesora opisują dalsze własności wyniku operacji (np. czy wynik jest liczbą ujemną — znacznik SF), jak również aktualnie włączony tryb adresowania dla operacji na blokach danych, zdolność procesora do przyjmowania sygnałów z urządzeń zewnętrznych i inne.

Rejestr stanu procesora (4)

- Bity rejestru stanu o numerach 12, 13, …, 21 zawierają informacje o stanie procesora, które są odczytywane i zapisywane przez system operacyjny.
- Bity o numerach 22 ÷ 31 nie są używane.



Rejestr stanu procesora (5)

- Zawartość rejestru stanu procesora można zapisać na wierzchołku stosu (zob. dalszy opis) za pomocą rozkazu **pushf**.
- Tak samo za pomocą rozkazu **popf** można przesłać zawartość wierzchołka stosu do rejestru stanu procesora. Jednak ewentualne zmiany znaczników wykorzystywanych przez system operacyjny zostaną zignorowane.

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).

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).

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 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)

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.

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*>

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**.

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.

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:
```

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.

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.

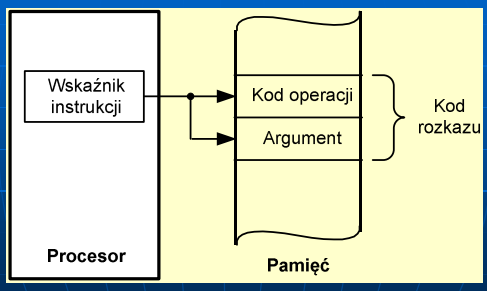
Tryby adresowania (3)

- 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.

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.

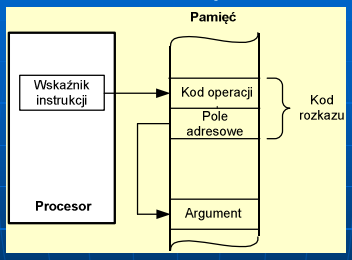
Adresowanie natychmiastowe (2)



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
```

Adresowanie rejestrowe (1)

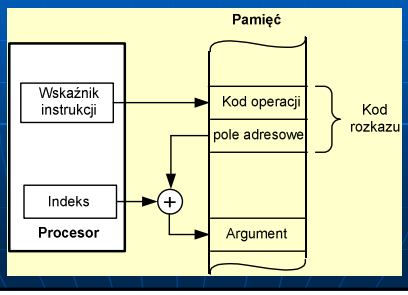
- Odmianą adresowania bezpośredniego jest *adresowanie rejestrowe* gdy argumenty operacji są zawarte w rejestrach procesora. Wtedy w rozkazie adres argumentu przyjmuje postać identyfikatora (numeru) rejestru.
- Ponieważ w celu pobrania argumentów nie trzeba (dodatkowo) sięgać do pamięci, rozkazy z adresowaniem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

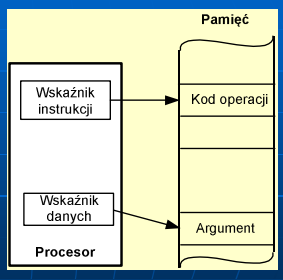
    add R4, 360(R1)
  - podobny zapis w asemblerze Intel:
    `add EAX, [EBX + 360]`

Adresowanie indeksowe (3)

- W architekturze x86rolę 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]`

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.

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]

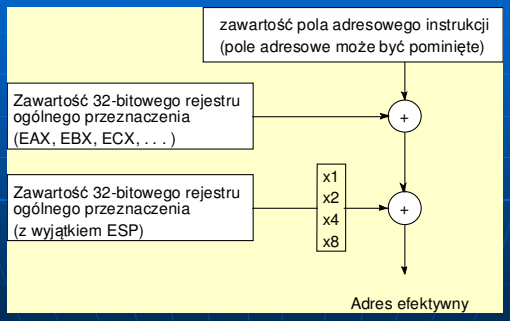
  - podobny zapis w asemblerze Intel:

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

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 2, 4 lub 8 (tzw. współczynnik skali).

Adresowanie bazowo-indeksowe (4)



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.

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

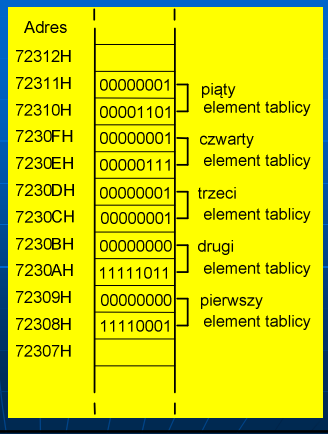
- Posługując się adresowaniem bazowo-indeksowym (lub tylko indeksowym) można łatwo budować pętle rozkazowe, w których ten sam rozkaz wykonuje działania na kolejnych elementach tablicy.
- W pętlach tego typu zwykle rejestr bazowy wskazuje położenie tablicy w pamięci, a rejestr indeksowy wskazuje położenie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

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.

Przykład sumowania elementów tablicy (2)

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



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).

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.

Przykład sumowania elementów tablicy (5)

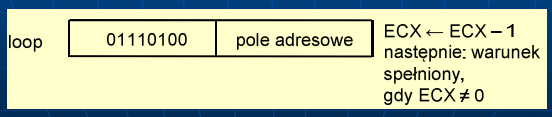
```
    mov     ecx, 4              ; licznik obiegów pętli
    mov     ax, ds:[72308H] ; pocz. wartość sumy
    mov     ebx, 0              ; pocz .zawartość
                                ; rejestru indeksowego
ptl_suma:

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

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

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.

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).



Rozkaz loop (3)

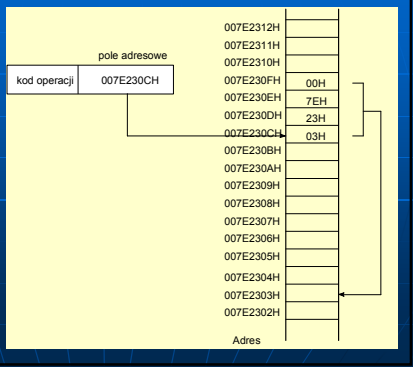
- W niektórych sytuacjach rozkaz **loop** można zastąpić parą rozkazów:

```
dec     ecx
jnz     jakaś_etykieta
```

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

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)

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.

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)+

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.

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:

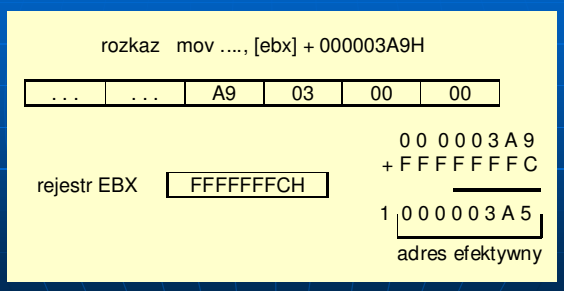
Regs[R2] ← Regs[R2] – *d*
Regs[R1] ← Regs[R1] + Mem[Regs[R2]]

add R1, –(R2)

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:

Obliczanie adresu efektywnego (2)



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.

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.

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.

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.

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.

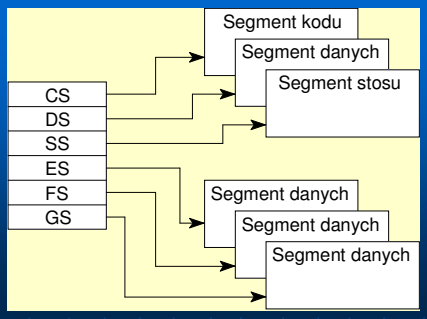
Adresowanie pamięci w trybie rzeczywistym (1)

- 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.

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.

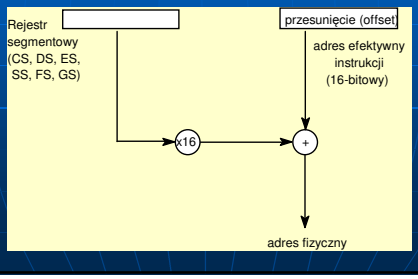
Adresowanie pamięci w trybie rzeczywistym (3)



Adresowanie pamięci w trybie rzeczywistym (4)

- W trybie rzeczywistym:

adres fizyczny =
= zawartość rejestru segmentowego * 16 + przesunięcie



Adresowanie pamięci w trybie rzeczywistym (5)

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

segment : offset

- Przykładowo, 32-bitowy programowy licznik czasu umieszczony jest w lokacji pamięci o adresie 40H : 6CH, tzn. w lokacji pamięci o adresie fizycznym 40H * 16 + 6CH = 46CH.

Adresowanie pamięci w trybie chronionym

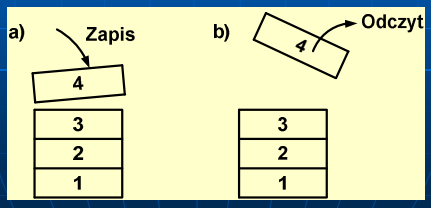
- Sposób obliczania adresu fizycznego w trybie chronionym jest znacznie bardziej skomplikowany: zawartości rejestrów segmentowych traktowane są jako indeksy do tablic systemowych, w których zawarte są adresy podlegające jeszcze dalszym przekształceniom w ramach mechanizmu stronicowania.
- We współczesnych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

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.

Organizacja stosu (2)

- 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 (3)

- Często działanie stosu ilustruje się w postaci stosu książek: kolejne książki kładzie się na wierzch stosu, i zdejmuje się, jeśli zachodzi taka potrzeba, również z wierzchołka stosu.
- Aby wydobyć książkę poniżej wierzchołka stosu, trzeba najpierw usunąć wszystkie książki znajdujące się nad książką żądaną.

Stos z punktu widzenia architektury komputerów

- W architekturze komputerów stos jest rozumiany jako obszar w pamięci głównej (operacyjnej), w którym dane są dopisywane lub usuwane wg reguł LIFO (ang. Last In, First Out).
- Zazwyczaj kolejne dane ładowane na stos umieszczane są w lokacjach pamięci o coraz niższych adresach — czasami mówimy, że stos rośnie w kierunku malejących adresów.

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.

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.

Typowe zastosowania stosu

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

Wierzchołek stosu

- 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 wskazuje adres komórki pamięci, w której znajduje się dana ostatnio zapisana na stosie.

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.

Operacje push i pop (1)

- Procesor realizuje dwie podstawowe operacje stosu:
  push — zapisywanie danych na stosie
  pop — odczytywanie danych ze stosu
- W architekturze x86 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. Dodatkowo wymaga się by zawartość rejestru ESP była podzielna przez 4, czyli zawartość rejestru ESP musi wskazywać lokację pamięci o adresie podzielnym przez 4.

Operacje push i pop (2)

- 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 w wierzchołka stosu i wpisanie jej do rejestru EDI.

Operacje push i pop (3)

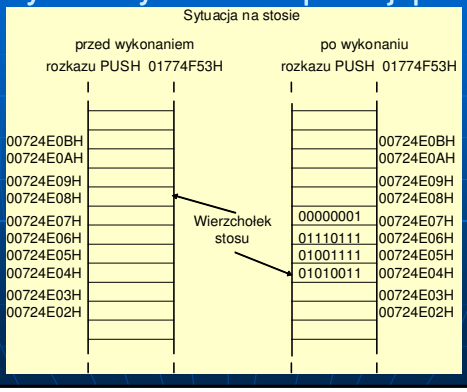
- Operandem rozkazów push i pop może być także lokacja pamięci, np.

```
wynik     dd  2345
  —   —   —   —   —   —
push      wynik
```

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

```
push      01774F53H
```

Przykład wykonania operacji push



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.

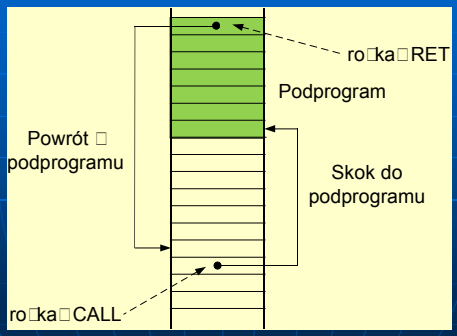
Organizacja stosu w innych architekturach

- 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)



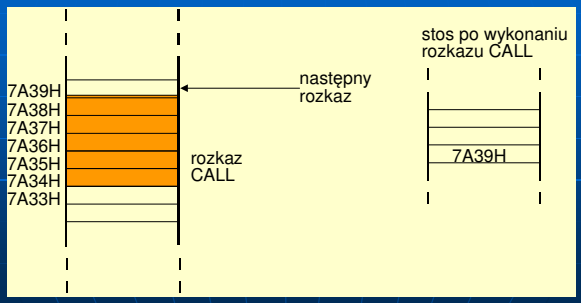
Rozkazy CALL i RET (1)

- 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.

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)



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)

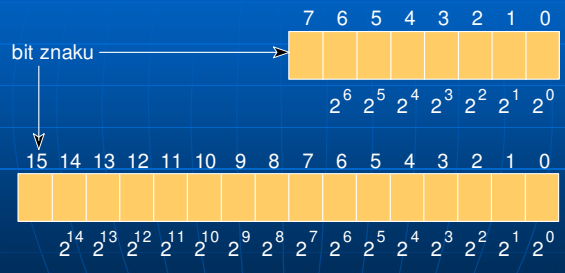
- Przykładowe wywołanie podprogramu

```
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.

Kodowanie liczb całkowitych ze znakiem



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.

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)

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

Kodowanie w systemie U2 (1)

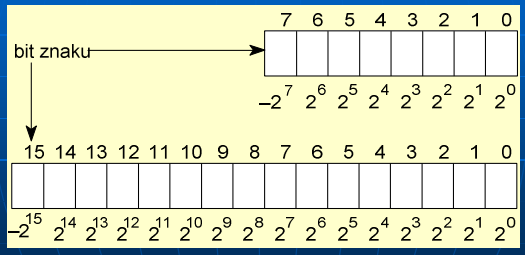
- Kodowanie liczb w systemie U2 jest obecnie powszechnie stosowane w wielu systemach komputerowych.
- Taki rodzaj kodowania upraszcza i przyśpiesza wykonywanie operacji arytmetycznych przez procesor.

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