# Języki asemblerowe

## WYKŁAD 7

Dr Krzysztof Balicki

# Modele pamięci

- Model pamięci określa konfigurację decydująca o sposobie użycia i łączenia segmentów.
- Każdy model cechuje się innymi ograniczeniami w zakresie maksymalnej przestrzeni dostępnej dla kodu programu i danych.
- Dobór modelu pamięci wpływa na sposób dostępu do danych (zmiennych) i procedur.

# Modele pamięci

| Model | Dane | Kod programu |
|---|---|---|
| ***Tiny*** | jeden segment maks. 64 kB | |
| ***Small*** | jeden segment maks. 64 kB | jeden segment maks. 64 kB |
| ***Medium*** | jeden segment maks. 64 kB | dowolna liczba segmentów |
| ***Compact*** | dowolna liczba segmentów | jeden segment maks. 64 kB |
| ***Large*** | dowolna liczba segmentów | dowolna liczba segmentów |
| ***Huge*** | dowolna liczba segmentów, pojedyncze zmienne mogą być większe niż 64 kB | dowolna liczba segmentów |
| ***Flat*** | brak segmentów, używany tylko w trybie chronionym | |

# Modele pamięci

- Deklaracja modelu pamięci odbywa się za pomocą dyrektywy:

```
.model typ
```

- Programy typu COM wykorzystują model pamięci **tiny**

# Programy typu COM

- COM - ang. command (rozkaz).
- Nazywane są mapą pamięci - to co ładowane jest i uruchamiane w pamięci jest wierną kopią programu na dysku.
- Podczas ładowania wszystkim rejestrom segmentowym (CS, DS, ES, SS) przypisywana jest ta sama wartość (wartość segmentu bloku PSP).

# Programy typu COM

- Zawartość rejestru IP na początku zawsze wynosi 100h. Sterowanie w programie rozpoczyna się za 256 bajtowym blokiem PSP od adresu CS:0100h.
- W kodzie asemblerowym musi zostać umieszczona dyrektywa przesunięcia:

```
org 100h
```

# Programy typu COM

- Programy nie mogą być dłuższe niż 64 kB, muszą zmieścić się w jednym segmencie.
- Programy nie mają dostępu do danych lub procedur umieszczonych w innych segmentach.
- Programy są szybciej ładowane do pamięci niż programy typu EXE.
- Programy nie wymagają stosu.

# Programy typu EXE

- EXE - ang. execution (wykonanie).
- Są to programy relokowalne.
- Programy mogą być ładowane i uruchamiane w pamięci począwszy od dowolnego adresu będącego wielokrotnością 10h.
- Mogą zajmować wiele segmentów.
- Segmenty mają zapewnioną wewnętrzną komunikację za pomocą 32 bitowych adresów logicznych **`segment:offset`**.

# Programy typu EXE

- Program składa się z nagłówka (zawierającego m.in. opis pliku, wymagania dotyczące pamięci operacyjnej, początkowe zawartości rejestrów) i modułu przeznaczonego do umieszczenia w pamięci.
- Wielkość programu ograniczona jest tylko wielkością dostępnej pamięci.

# Pliki OBJ

- Pliki w formacie OBJ używane są m.in. przez linkery w systemie DOS do generowania plików EXE.
- Pliki OBJ posiadają rozszerzenie .obj.
- Format OBJ nie definiuje specjalnych nazw segmentów. Segmenty mogą posiadać dowolne nazwy.
- W przypadku braku zdefiniowania segmentu NASM umieszcza kod programu w domyślnym segmencie o nazwie:

**__NASMDEFSEG**

# Pliki OBJ

- NASM traktuje zdefiniowane nazwy segmentów jako etykiety, będące adresami segmentów.
- Format OBJ pozwala specyfikować różne właściwości segmentów, np.:

```
segment kod public
segment stos stack
```

# Pliki OBJ

- Format OBJ pozwala grupować segmenty, np.:

```
segment dane1
segment dane2
group dane dane1 dane2
```

- Pojedynczy rejestr segmentowy może być używany do uzyskiwania dostępu do wszystkich segmentów w grupie.
- Nazwa grupy jest także traktowana przez NASM jako etykieta.

# Pliki EXE

- Każdy program wymagający powyżej 64kB przestrzeni w pamięci musi być tworzony jako plik EXE.

# Tworzenie plików .exe

- Plik .exe może zostać stworzony przez linker przez połączenie plików .obj.
- Dokładnie jeden z plików .obj musi mieć zdefiniowany punkt startu.
- Punkt startu definiowany jest w pliku źródłowym (.asm) za pomocą:

```
..start
```

# Tworzenie plików .exe

- Inicjalizacja rejestrów segmentowych:
  - rejestr DS powinien wskazywać segment danych
  - rejestr SS powinien wskazywać segment stosu
- Ustawienie wierzchołka stosu:
  - rejestr SP powinien wskazywać wierzchołek stosu (adres offsetowy wierzchołka)

# Linkery

- Przykładowe linkery:
  - VAL
  - FREELINK
  - djlink
  - ALINK

# Tworzenie plików .exe

- Przykładowy program:

```
segment dane
napis: DB 'Program EXE', 13, 10, '$'

segment program
..start:
mov AX,dane
mov DS,AX
mov AX,stos
mov SS,AX
mov SP,wierzcholek
mov DX,napis
mov AH,09h
int 21h
mov AH,4Ch
int 21h

segment stos stack
resb 64
wierzcholek:
```

# Tworzenie plików .exe

- Komplilacja:

```
nasm -fobj program.asm
```

- Linkowanie:

```
val program.obj, program.exe
```

# Modele pamięci

- NASM wspiera różne modele pamięci:
  - modele wykorzystujące pojedynczy segment kodu (tiny, small, compact)
    - w tych modelach używane są tzw. funkcje bliskie (NEAR)
    - wskaźniki funkcji są 16 bitowe i zawierają tylko adres offsetowy, rejestr segmentu kodu (CS) nie zmienia swojej wartości i zawsze dostarcza taki sam adres segmentu
    - funkcje bliskie wywoływane są za pomocą instrukcji **`CALL`**, a powrót z funkcji odbywa się za pomocą instrukcji **`RET`**

# Modele pamięci

- NASM wspiera różne modele pamięci (cd.):
  - modele wykorzystujące więcej niż jeden segment kodu (medium, large, huge)
    - w tych modelach używane są tzw. funkcje dalekie (FAR)
    - wskaźniki funkcji są 32 bitowe i zawierają 16 bitowy adres segmentu oraz 16 bitowy adres offsetowy
    - funkcje dalekie wywoływane są za pomocą instrukcji **`CALL FAR`** lub **`CALL segm:offset`**, a powrót z funkcji odbywa się za pomocą instrukcji **`RETF`**

# Modele pamięci

- NASM wspiera różne modele pamięci:
  - modele wykorzystujące pojedynczy segment danych (tiny, small, medium)
    - wskaźniki danych są 16 bitowe i zawierają tylko adres offsetowy, rejestr segmentu danych (DS) nie zmienia swojej wartości i zawsze dostarcza taki sam adres segmentu
    - funkcje bliskie wywoływane są za pomocą instrukcji **`CALL`**, a powrót z funkcji odbywa się za pomocą instrukcji **`RET`**

# Modele pamięci

- NASM wspiera różne modele pamięci (cd.):
  - modele wykorzystujące więcej niż jeden segment danych (compact, large, huge)
    - wskaźniki danych są 32 bitowe i zawierają 16 bitowy adres segmentu oraz 16 bitowy adres offsetowy
    - należy pamiętać o niemodyfikowaniu zawartości rejestru DS przed wcześniejszym zapisaniem jego pierwotnej wartości

# Stos

- Przestrzeń pamięci zarezerwowana w segmencie stosu jest używana do implementacji stosu.
- Implementacja stosu realizowana jest w procesorze Pentium za pomocą rejestrów SS oraz ESP.
- Wierzchołek stosu jest identyfikowany przez adres SS:ESP, gdzie SS wskazuje początek segmentu stosu zaś ESP adres offsetowy wierzchołka stosu.

# Stos

- Implementacja stosu w procesorze Pentium posiada następujące właściwości:
  - na stosie mogą być odkładane tylko wartości o długości słowa (16 bitów) lub podwójnego słowa (32 bity), nie można odkładać wartości jednobajtowych,
  - wierzchołek stosu przesuwa się w kierunku adresów o mniejszych wartościach,
  - wierzchołek stosu wskazuje zawsze ostatni element umieszczony na stosie, młodszy bajt słowa umieszczonego na stosie.

# Operacje arytmetyczne

- Sześć znaczników (flag) w rejestrze znaczników używane jest do monitorowania wyników operacji arytmetycznych:
  - znacznik zera (ZF),
  - znacznik przeniesienia (CF),
  - znacznik przepełnienia (OF),
  - znacznik znaku (SF),
  - znacznik pomocniczego przepełnienia (AF),
  - znacznik parzystości (PF).

  Znaczniki te nazywane są znacznikami statusu.

# Operacje arytmetyczne

- Przykładowe instrukcje modyfikujące znacznik zera:
  - add
  - sub
  - inc
  - dec
- Instrukcje skoku testujące znacznik zera:
  - jz (skocz jeśli ZF równy 0)
  - jnz (skocz jeśli ZF różny od 0)

# Operacje arytmetyczne

- Przykładowe instrukcje modyfikujące znacznik przeniesienia:
  - add
  - sub
- UWAGA: Instrukcje „inc” oraz „dec” nie modyfikują znacznika przeniesienia
- Instrukcje skoku testujące znacznik przeniesienia:
  - jc (skocz jeśli wystąpiło przeniesienie, tj. CF równy 1)
  - jnc (skocz jeśli CF równy 0)

# Operacje arytmetyczne

- Przykładowe instrukcje modyfikujące znacznik przepełnienia:
  - add
  - sub
  - inc
  - dec
- Instrukcje skoku testujące znacznik przeniesienia:
  - jo (skocz jeśli wystąpiło przepełnienie, tj. OF równy 1)
  - jno (skocz jeśli OF równy 0)

## Operacje arytmetyczne

- Dodatkowa instrukcja przerwania programowego „into” testuje znacznik przepełnienia.

# Operacje arytmetyczne

- Przykład:
  - przy dodawaniu dwóch wartości 72h i 0Eh procesor traktuje te wartości zarówno jako:
    - wartości bez znaku (*unsigned*) - wynik mieści się w zakresie 00h (0)... 0FFh (255) liczb bez znaku i znacznik przeniesienia (CF) nie jest ustawiany
    - wartości ze znakiem (*signed*) - wynik nie mieści się w zakresie 0FFh (-128) ... 7F (127) liczb ze znakiem i znacznik przepełnienia (OF) jest ustawiany

# Operacje arytmetyczne

- Znacznik znaku jest kopią najstarszego bitu liczby.
- Instrukcje skoku testujące znacznik znaku:
  - js (skocz jeśli SF równy 1)
  - jns (skocz jeśli SF równy 0)

# Operacje arytmetyczne

- Znacznik pomocniczego przeniesienia jest generowany jeśli pojawi się przeniesienie (pożyczka) dla czterech najmłodszych bitów liczby.
- Instrukcje poprawek testujące znacznik pomocniczego przeniesienia:
  - aaa
  - aas
  - aam
  - aad
  - daa
  - das

# Operacje arytmetyczne

- Znacznik parzystości określa parzystość 8 najmłodszych bitów wyniku operacji (nawet jeśli operacje wykonywane są na wartościach 16 lub 32 bitowych).
- Instrukcje skoku testujące znacznik znaku:
  - jp (skocz jeśli PF równy 1)
  - jnp (skocz jeśli PF równy 0)

# Mnożenie

- AL * 8-bitowy operand = AH:AL
- AX * 16-bitowy operand = DX:AX
- EAX * 32-bitowy operand = EDX:EAX

# Dzielenie

- AX / 8-bitowy operand = AL, reszta AH
- DX:AX / 16-bitowy operand = AX, reszta DX
- EDX:EAX / 32-bitowy operand = EAX, reszta EDX

# Dodawanie liczb wielobajtowych

- Przykład - dodawanie liczb 64 bajtowych:
  - operandy wejściowe: EBX:EAX oraz EDX:ECX
  - wynik: EBX:EAX

```
add EAX,ECX
adc EBX,EDX
```

- Rozkaz „adc” realizuje dodawanie z przeniesieniem (dodawane są dwa operandy i jeśli CF=1 dodawana jest wartość 1)

# Dodawanie liczb zmiennoprzecinkowych

$$m_S 2^{e_S} = m_A 2^{e_A} + m_B 2^{e_B}$$

$$m_S = m'_A + m'_B$$

$$e_S = max(e_A, e_B)$$

$$m'_A = m_A \qquad m'_B = m_B \qquad \text{jeśli } e_A = e_B$$

$$m'_A = m_A \qquad m'_B = m_B 2^{-|e_A - e_B|} \qquad \text{jeśli } e_A > e_B$$

$$m'_A = m_A 2^{-|e_A - e_B|} \qquad m'_B = m_B \qquad \text{jeśli } e_A < e_B$$

# Odejmowanie liczb zmiennoprzecinkowych

$$m_R 2^{e_R} = m_A 2^{e_A} - m_B 2^{e_B}$$

$$m_R = m'_A - m'_B$$

$$e_R = max(e_A, e_B)$$

$$m'_A = m_A \qquad m'_B = m_B \qquad \text{jeśli } e_A = e_B$$

$$m'_A = m_A \qquad m'_B = m_B 2^{-|e_A - e_B|} \qquad \text{jeśli } e_A > e_B$$

$$m'_A = m_A 2^{-|e_A - e_B|} \qquad m'_B = m_B \qquad \text{jeśli } e_A < e_B$$

# Mnożenie liczb zmiennoprzecinkowych

$$m_I 2^{e_I} = m_A 2^{e_A} \cdot m_B 2^{e_B}$$

$$m_I = m_A \cdot m_B$$

$$e_I = e_A + e_B$$

# Dzielenie liczb zmiennoprzecinkowych

$$m_I 2^{e_I} = \frac{m_A 2^{e_A}}{m_B 2^{e_B}}$$

$$m_I = \frac{m_A}{m_B}$$

$$e_I = e_A - e_B$$

# Dodawanie mantys w kodzie ZM

- $A$, $B$ - mantysy
- Dodawanie
  - jeśli *znak* $A$ = *znak* $B$, to:

$$|S| = |A| + |B|$$
$$znak\ S = znak\ A = znak\ B$$

# Dodawanie mantys w kodzie ZM

- $A$, $B$ - mantysy
- Dodawanie
  - jeśli *znak* $A \neq$ *znak* $B$, to:
    - jeśli $|A|>|B|$, to:

$$|S| = |A| - |B|$$
$$znak\ S = znak\ A$$

    - jeśli $|A|<|B|$, to:

$$|S| = |B| - |A|$$
$$znak\ S = znak\ B$$

# Odejmowanie mantys w kodzie ZM

- $A$, $B$ - mantysy
- Odejmowanie
  - jeśli $znak\ A \neq znak\ B$, to:

$$|R| = |A| + |B|$$
$$znak\ R = znak\ A$$

# Odejmowanie mantys w kodzie ZM

- $A$, $B$ - mantysy
- Odejmowanie
  - jeśli *znak* $A$ = *znak* $B$, to:
    - jeśli $|A| \geq |B|$, to:

$$|R| = |A| - |B|$$
$$znak\ R = znak\ A$$

    - jeśli $|A| < |B|$, to:

$$|R| = |B| - |A|$$
$$znak\ R = znak\ B$$

# Mnożenie mantys w kodzie ZM

- $A$, $B$ - mantysy
- Mnożenie

$$|I| = |A| \cdot |B|$$
$$znak\ I = xor\left(znak\ A\ ,\ znak\ B\right)$$

# Dzielenie mantys w kodzie ZM

- $A$, $B$ - mantysy
- Dzielenie

$$|I| = \frac{|A|}{|B|}$$

$$znak\ I = xor\left(znak\ A\ ,\ znak\ B\right)$$