Programowanie niskopoziomowe

Prowadzący:
Piotr Kisała

LABORATORIUM 3

Przegląd tematów

- Dostęp do pamięci – tryby adresowania
- Organizacja pamięci fazy wykonania
- Koercja typów
- Pamięć obszaru stosu
- Dekrementacja i inkrementacja pamięci programu i rejestrów

Tryby adresowania procesorów 80x86

Adresowanie 32-bitowe:

wykorzystywane obecnie w większości systemów oper. (Windows, Linux, BeOS)

Adresowanie 16-bitowe:

przestarzałe, obecnie nie wykorzystywane, wszystkie operacje możliwe do wykonania w trybach 16-bitowych da się wykonać (efektywniej) w trybach 32-bitowych

Adresowanie przez rejestr

Większość instrukcji maszynowych procesora 80x86 wykorzystuje w roli operandów rejestry ogólnego przeznaczenia. Dostęp do rejestru – poprzez określenie w miejsce operandu instrukcji nazwę rejestru, np.:

mov(operand-źródłowy, operand-docelowy);

Operandami mogą być 8-,16-,32-bitowe rejestry procesora.

Instrukcje odwołujące się do rejestrów są wykonywane szybciej od tych, które odwołują się do pamięci, ich zapis jest również krótszy.

Większość instrukcji obliczeniowych wymaga wprost aby jeden z operandów był umieszczony w rejestrze, stąd adresowanie przez rejestr jest w kodzie asemblerowym procesora 80x86 bardzo częste.

32-bitowe tryby adresowania

- Dobór odpowiedniego trybu adresowania to klucz do efektywnego programowania w asemblerze.
- Tryby adresowania procesorów 80x86 obejmują:
  - adresowanie bezpośrednie.
  - adresowanie bazowe,
  - adresowanie bazowe indeksowane,
  - adresowanie indeksowe,
  - adresowanie bazowe indeksowane z przemieszczeniem

  Pozostałe tryby adresowania to odmiany owych trybów podstawowych.

Adresowanie bezpośrednie

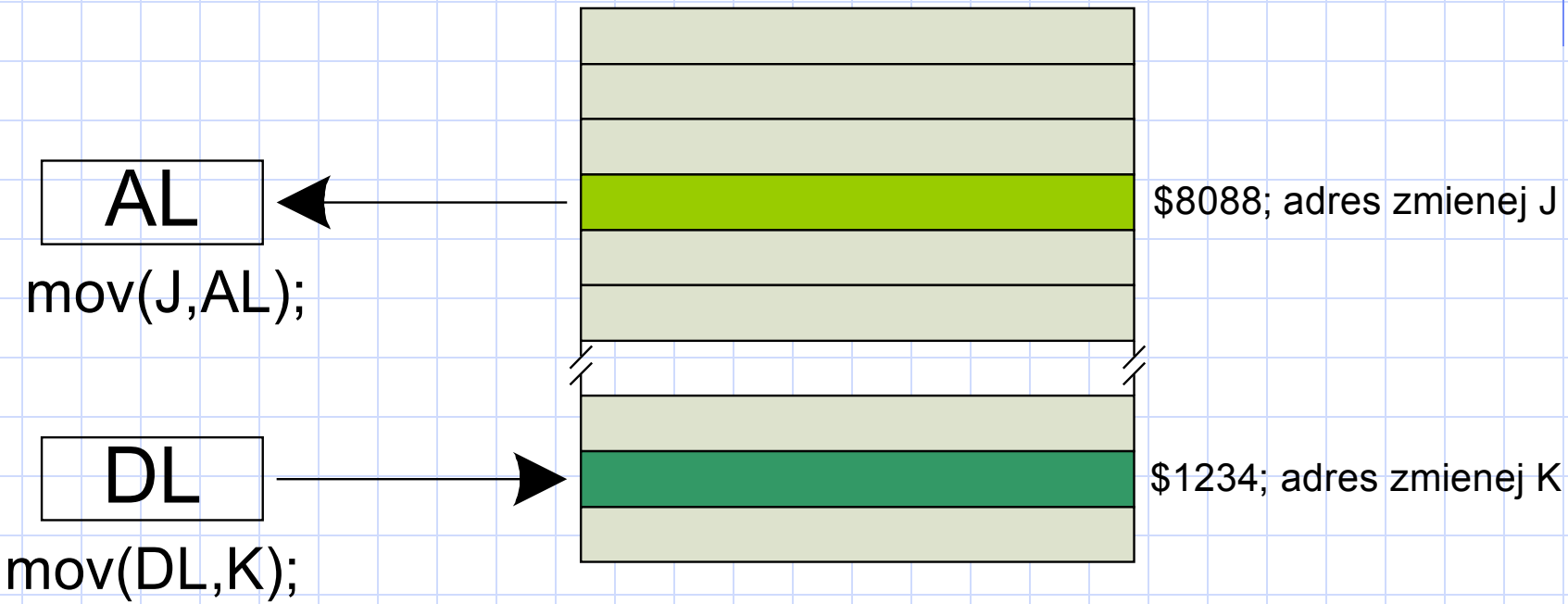
Najczęściej wykorzystywane i najprostsze do opanowania. Adres docelowy określany jest 32-bitową stałą.

Tryb adresowania bezpośredniego doskonale nadaje się do realizacji odwołań do prostych zmiennych skalarnych.

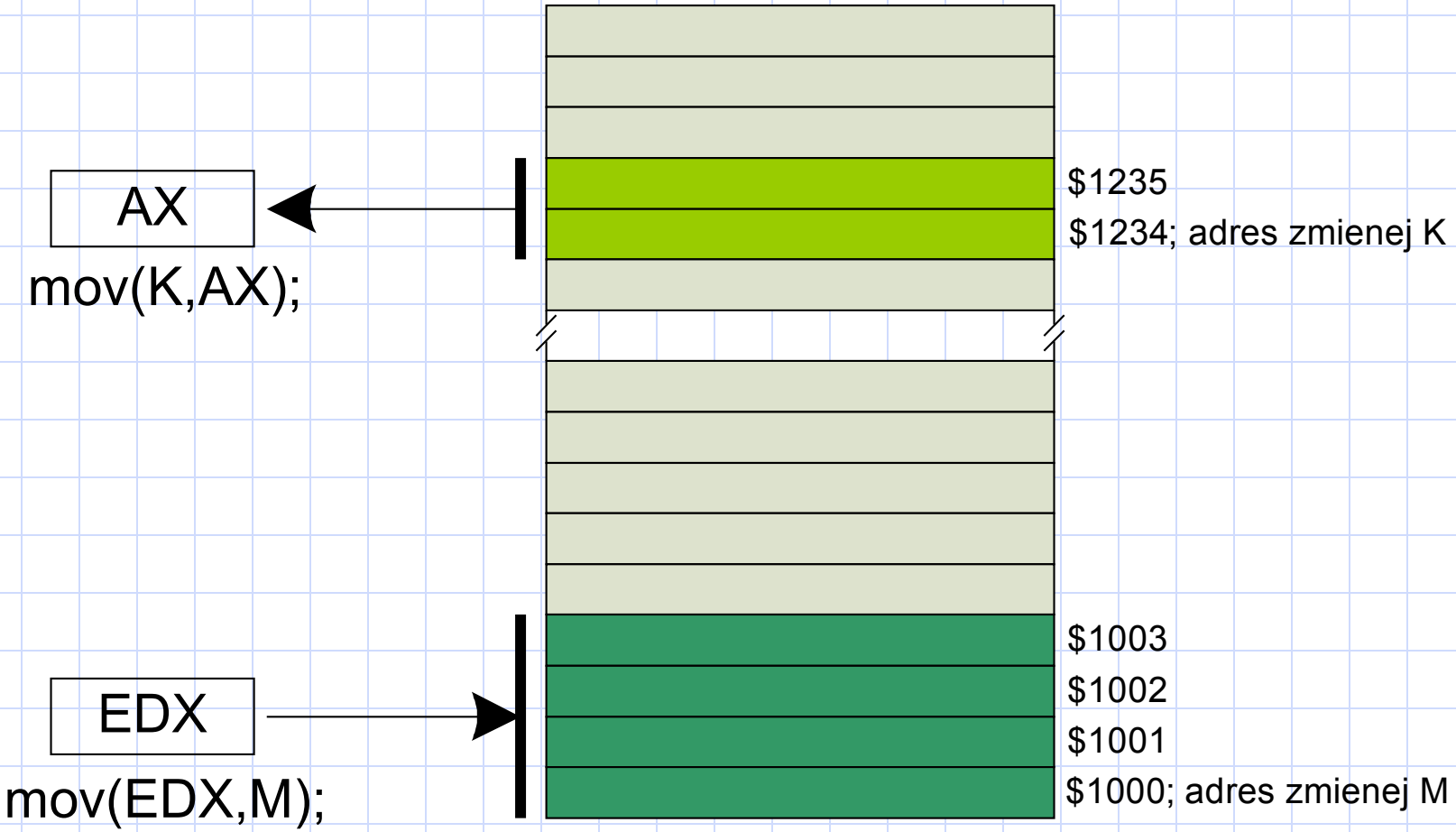
Intel przyjęła dla tego trybu nazwę adresowania z przemieszczeniem, ponieważ bezpośrednio po kodzie instrukcji mov w pamięci zapisana jest 32-bitowa stała przemieszczana. Przemieszczenie w procesorach 80x86 definiowane jest jako przesunięcie (ang. offset) od początkowego adresu pamięci (czyli adresu zerowego).

Adresy obiektów o rozmiarze słowa, podwójnego słowa i większych określa się analogicznie podając adres pierwszego bajta obiektu.

Tryby adresowania bezpośredniego



Odwołanie do słowa i podwójnego słowa – adr. bezp.



Adresowanie pośrednie przez rejestr

Pośrednie – operand nie jest właściwym adresem, dopiero wartość operandu określa adres odwołania.

Wartość rejestru do docelowy adres pamięci.

Tryb adresowania pośredniego przez rejestr jest sygnalizowany nawiasami prostokątnymi.

Instrukcja mov(EAX, [EBX]) informuje procesor, aby ten zachował zawartość rejestru EAX w miejscu, którego adres znajduje się w rejestrze EBX.

Adresowanie pośrednie przez rejestr

Procesowy 80x86 obsługują osiem wersji adresowania pośredniego przez rejestr:

```
mov( [eax], al );
mov( [ebx], al );
mov( [ecx], al );
mov( [edx], al );
mov( [edi], al );
mov( [esi], al );
mov( [ebp], al );
mov( [esp], al );
```

Wersje te różnią się tylko rejestrem, w którym przechowywany jest właściwy adres operandu.

Adresowanie pośrednie przez rejestr

- Konieczne jest stosowanie rejestrów 32-bitowych
- Umożliwia odwoływanie się do danych, dysponując jedynie wskaźnikami na nie.
- Tryb nadaje się do modyfikowania adresu docelowego odwołania w czasie działania programu.

Adresowanie pośrednie przez rejestr

HLA udostępnia prosty operator pozwalający na załadowanie 32-bitowego rejestru adresem zmiennej, o ile jest to zmienna statyczna.

```
mov( &J, EBX );          //załadowanie rejestru EBX adresem zmiennej J
mov(EAX, [EBX]);         // zapisanie w zmiennej J wartości rejestru EAX
```

Operator pobrania adresu ma postać identyczną jak w C i C++ - jest to znak &.

Adresowanie indeksowe

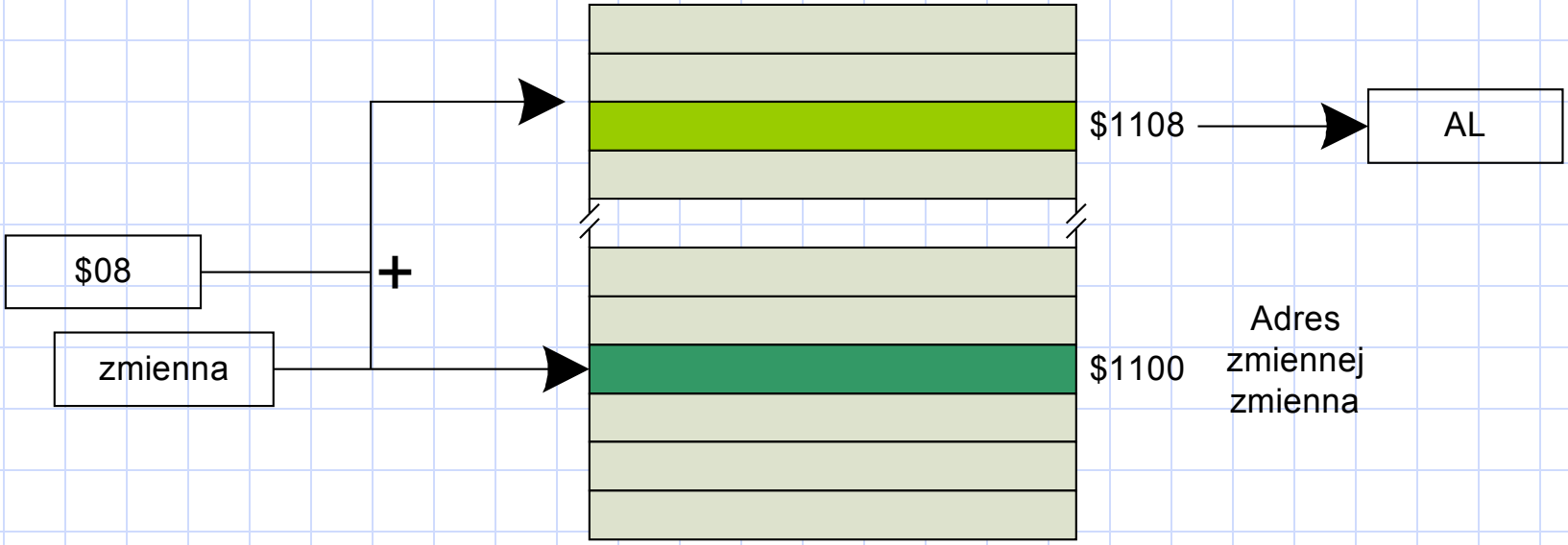
- Wykorzystuje następującą składnię instrukcji:
  - mov(zmienna[EAX], AL);
  - mov(zmienna[EBX], AL);
  - mov(zmienna[ECX], AL);
  - mov(zmienna[EDX], AL);
  - mov(zmienna[EDI], AL);
  - mov(zmienna[ESI], AL);
  - mov(zmienna[EBP], AL);
  - mov(zmienna[ESP], AL);

Adresowanie indeksowe

- Obliczany jest efektywny adres obiektu docelowego. Polega to na dodaniu do adresu zmiennej wartości zapisanej w 32-bitowym rejestrze umieszczonym w nawiasach prostokątnych. Dopiero suma tych wartości określa właściwy adres pamięci, do którego ma nastąpić odwołanie.
- Jeśli więc zmienna przechowywana jest w pamięci pod adresem $1100, a rejestr EBX zawiera wartość 8 to wykonanie instrukcji mov(zmienna [EBX], AL) powoduje umieszczenie w rejestrze AL wartości zapisanej w pamięci pod adresem $1108
- Tryb adresowania indeksowego jest szczególnie poręczny do odwoływania się do elementów tablic.

Adresowanie indeksowe

mov(zmienna[EBX], AL);



Adresowanie indeksowe skalowane

- Umożliwia uwikłanie oprócz wartości przemieszczenia, zawartość dwóch rejestrów.
- Pozwala na wymnożenie rejestru indeksowego przez współczynnik (skalę) o wartości 1, 2, 4 bądź 8.

Adresowanie indeksowe skalowane

Składnię tego trybu określa się następująco:

zmienna[rejestr-indeksowy32 * skala]
zmienna[rejestr-indeksowy32 * skala + przesunięcie]
zmienna[rejestr-indeksowy32 * skala - przesunięcie]

[rejestr-bazowy32 + rejestr-indeksowy32 * skala]
[rejestr-bazowy32 + rejestr-indeksowy32 * skala + przesunięcie]
[rejestr-bazowy32 + rejestr-indeksowy32 * skala - przesunięcie]

zmienna[rejestr-bazowy32 + rejestr-indeksowy32 * skala]
zmienna[rejestr-bazowy32 + rejestr-indeksowy32 * skala + przesunięcie]
zmienna[rejestr-bazowy32 + rejestr-indeksowy32 * skala - przesunięcie]

Adresowanie indeksowe skalowane

rejestr-bazowy32 reprezentuje dowolny z 32-bitowych rejestrów ogólnego przeznaczenia

podobnie jak rejestr-indeksowy32
(z puli dostępnych dla tego operandu rejestrów należy jednak wykluczyć rejestr ESP)

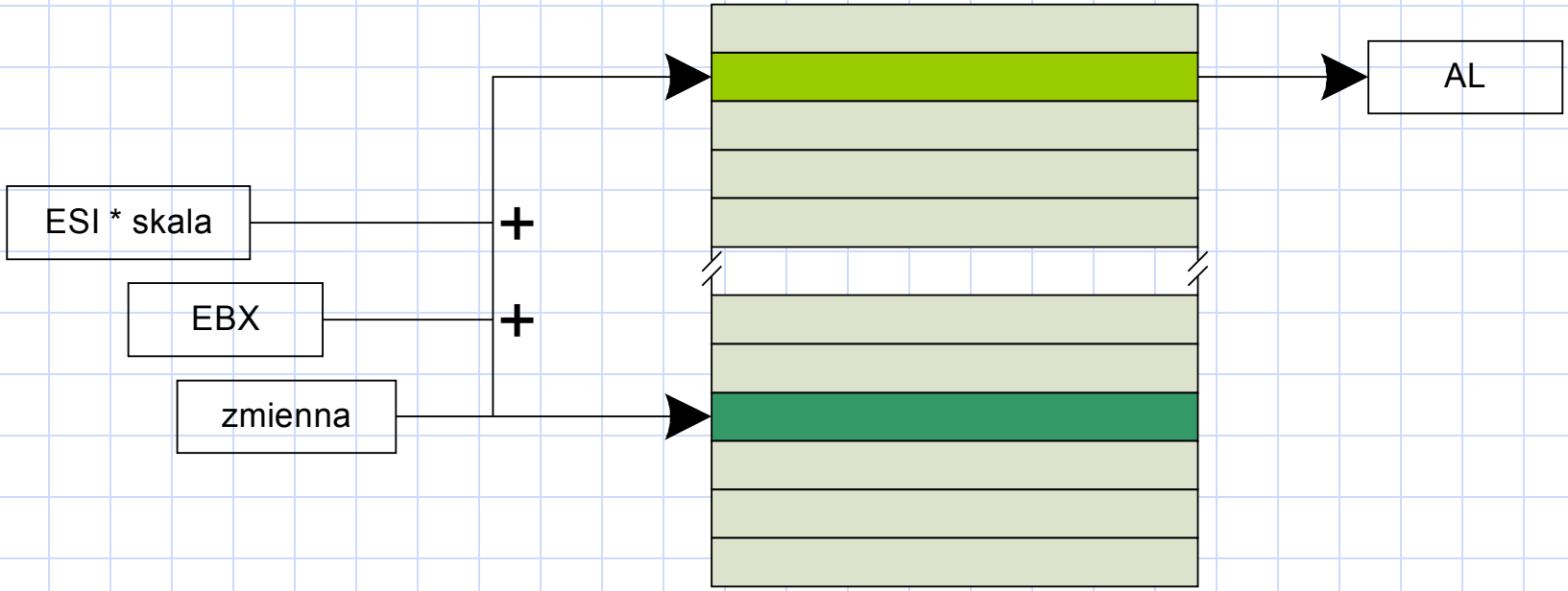
skala jest stałą o wartości 1, 2, 4 bądź 8

Adresowanie indeksowe skalowane

- Skalowane adresowanie indeksowe różni się od prostego adresowania indeksowego przede wszystkim składową rejestr-indeksowy32 * skala.
- W trybie tym adres efektywny obliczany jest przez dodanie wartości rejestru indeksowego pomnożonej przez współczynnik skalowania. Dopiero ta wartość wykorzystywana jest w roli indeksu.

Adresowanie indeksowe skalowane

mov(zmienna[EBX+ESI*skala], AL);



W roli rejestru bazowego występuje rejestr EBX; rejestrem indeksowym jest ESI.

Adresowanie indeksowe skalowane

Jeżeli przyjąć, że rejestr EBX zawiera wartość $100, rejestr ESI zawiera wartość $20, a zmienna została umieszczona w pamięci pod adresem $2000, wtedy instrukcja:

mov(zmienna[EBX + ESI*4 + 4], AL);

spowoduje skopiowanie do rejestru AL. pojedynczego bajta spod adresu $2184 ($2000+$100+$20*4+4)

Adresowanie indeksowe skalowane

- Tryb ten przydatny jest w odwołaniach do elementów tablicy, w której wszystkie elementy mają rozmiary dwóch, czterech bądź ośmiu bajtów.

Organizacja pamięci fazy wykonania

typowe rozmieszczenie elementów programu niskopoziomowego w pamięci

Organizacja pamięci fazy wykonania

- Najniższe adresy przestrzeni adresowej programu rezerwowane są przez system operacyjny.
  W ogólności aplikacje nie mogą odwoływać się do tego obszaru, ani wykonywać w nim instrukcji.

- Pozostałych 6 obszarów mapy pamięci programu to obszary przypisane do poszczególnych rodzajów danych. Mamu tu obszar stosu, sterty, kodu, danych niemodyfikowalnych (readonly), zmiennych statycznych, pamięci niezainicjalizowanej (storage).

Obszar kodu

- Obszar kodu zawiera instrukcje maszynowe tworzące właściwy program. Asembler tłumaczy instrukcje maszynowe kodu źródłowego do postaci sekwencji wartości jedno- bądź kilkubajtowych. Procesor interpretuje owe wartości jako instrukcje maszynowe (i ich operandy) i wykonuje je.
- Asembler przez domniemanie podczas konsolidacji programu informuje system operacyjny, że program może z obszaru kodu czytać instrukcje i dane. Nie może natomiast zapisywać danych w obszarze kodu. W przypadku próby takiego zapisu system operacyjny wygeneruje błąd ochrony pamięci.

Obszar zmiennych statycznych

- Obszar sygnalizowany słowem kluczowym static to domyślnie obszar deklarowania zmiennych.
- Deklaracje zmiennych za słowem kluczowym static powodują rezerwowanie pamięci, nawet jeśli do zmiennych nie przypisano żadnej wartości.

Obszar niemodyfikowalny

- Przechowuje stałe, tablice i inne dane programu, które nie mogą w czasie jego wykonywania podlegać żadnym modyfikacjom.
- Obiekty niemodyfikowalne deklaruje się w sekcji kodu sygnalizowanej słowem readonly.
- Wszystkie stałe deklarowane w sekcji readonly są inicjalizowane.

Obszar niemodyfikowalny

Przykład:

```
readonly
    pi:       real32    :=   3.1459;
    e:        real32    :=   2.71;
    Liczba1:uns16       :=   65_535;
    Liczba2:uns16       :=   32_767;
```

Obszar danych niezainicjalizowanych

W obszarze danych niezainicjalizowanych, którego deklaracje są w kodzie źródłowym programu zapowiadane słowem storage, wszystkie zmienne pozostają niezainicjalizowane:

```
storage
        niezainicjalizowana32:      uns32;
        i:                          int32;
        znak:                       char;
        b:                          byte;
```

w systemach Linux i Windows obszar zmiennych niezainicjalizowanych jest przy ładowaniu programu do pamięci wypełniany zerami

Sekcja deklaracji var

Sekcja rozpoczyna się słowem var, w ramach sekcji tworzone są zmienne automatyczne. Zmienne takie tworzone są w pamięci przy okazji rozpoczęcia wykonania pewnej jednostki programu (np. programu głównego lub procedury).

Pamięć zmiennych automatycznych jest zwalniana przy wychodzeniu z procedury czy programu.

Koercja typów

Jest to proces, w ramach którego asembler informowany jest o tym, że dany obiekt będzie traktowany jako obiekt typu określonego wprost w kodzie, niekoniecznie zgodnego z typem podanym w deklaracji. Składnia koercji typu zmiennej wygląda następująco:

(type nowa-nazwa-typu wyrażenie-adresowe)

Nowa nazwa typu określa typ docelowy koercji, który ma zostać skojarzony z adresem pamięci wyznaczonym wyrażeniem adresowym. Operator koercji może być wykorzystywany wszędzie tam, gdzie dozwolone jest określenie adresu w pamięci.

mov((type word zmienna8bitowa), AX);

Pamięć obszaru stosu

- Wszytkie zmienne deklarowane w sekcji var umieszczane są w obszarze pamięci zwany obszarem stosu.
- Obszar stosu to ten fragment pamięci programu, w której procesor przechowuje swój stos. Stos jest dynamiczną strukturą danych, która zwiększa lub zmniejsza swój rozmiar w zależności od bieżących potrzeb programu. Stos zawiera też ważne dla poprawnego działania programu informacje, w tym zmienne lokalne (automatyczne), informacje o wywołaniach procedur i dane tymczasowe.

Pamięć obszaru stosu

Pamięć stosu jest kontrolowana za pośrednictwem rejestru ESP zwanego też wskaźnikiem stosu. Kiedy program zaczyna działanie, system operacyjny inicjalizuje wskaźnik stosu adresem ostatniej komórki pamięci w obszarze pamięci stosu (największy możliwy adres w obszarze pamięci stosu).

Zapis danych do tego obszaru odbywa się jako "odkładanie danych na stos" (ang. pushing) i "zdejmowanie danych ze stosu (ang. popping).

Pamięć obszaru stosu

- Odłożenie danych na stos powoduje każdorazowo skopiowanie danych do obszaru pamięci wskazywanego przez rejestr ESP, a następnie zmniejszenie wartości wskaźnika stosu o rozmiar odłożonych danych.
- Stos rośnie w miarę odkładania na niego kolejnych danych i – analogicznie – maleje przy zdejmowaniu danych ze stosu.

Instrukcja push

Służy do składowania danych na stosie. Składnia:

```
push( rejestr16 );
push( rejestr32 );
push( pamięć16 );
push( pamięć32 );
pushw( stała );
pushd( stała );
```

operandami instrukcji pushw i pushd są zawsze stałe o rozmiarze odpowiednio słowa bądź podwójnego słowa.

Instrukcja push

Działanie instrukcji push można rozpisać następującym pseudokodem:

ESP := ESP – rozmiar-operandu (2 lub 4)

[ESP] := wartość-operandu

Rejestr ESP zawiera wartość $00FF_FFE8 to wykonanie instrukcji push(EAX); spowoduje ustawienie rejestru ESP na wertość $00FF_FFE4 i skopiowanie bieżącej wartości rejestru EAX pod adres $00FF_FFE4.

Instrukcja push

- Choć procesory 80x86 obsługują 16-bitowe wersje instrukcji manipulujących pamięcią stosu, to owe wersje mają zastosowanie głównie w środowiskach 16-bitowych jak system DOS.
- Gwoli maksymalnej wydajności należy utrzymywać wskaźnik stosu jako całkowitą wielokrotność liczby cztery.
- Jedynym uzasadnieniem dla odkładania na stosie danych innych niż 32-bitowe jest konstruowanie za pośrednictwem stosu wartości o rozmiarze podwójnego słowa składanej z dwóch słów umieszczonych na stosie jedno po drugim.

Instrukcja pop

Służy do zdejmowania danych umieszczonych wcześniej na stosie. Składnia:

```
pop( rejestr16 );
pop( rejestr32 );
pop( pamięć16 );
pop( pamięć32 );
```

Podobnie jak w instrukcji push, pop obsługuje jedynie operandy 16- i 32-bitowe; ze stosu nie można zdejmować wartości ośmiobitowych.

Instrukcja pop

Zdejmowanie ze stosu wartości 16-bitowych powinno się unikać, chyba, że operacja taka stanowi jedną z dwóch operacji zdejmowania ze stosu (realizowanych po kolei)

Sposób działania pop wygląda następująco:

operand := [ESP]

ESP := ESP + rozmiar-operandu (2 lub 4)

Operacja zdejmowania ze stosu jest operacją dokładnie odwrotną do operacji odkładania danych na stosie.

Zachowywanie wartości rejestrów – push i pop

Najważniejszym zastosowaniem push i pop jest zachowywanie wartości rejestrów w obliczu ich czasowego, innego niż dotychczasowe wykorzystania.

Wobec małej liczby rejestrów 80x86 stos przechowuje wartości tymczasowe (wyniki pośrednich etapów obliczeń).

Stos jako koljka LIFO

LIFO – last in, first out

Dane ze stosu należy zdejmować w kolejności odwrotnej do ich odkładania.

Zdejmować ze stosu należy dokładnie tyle bajtów, ile się wcześniej nań odłożyło.

Jeśli liczba instrukcji pop jest zbyt mała, na stosie pozostaną osierocone dane, co może w dalszym przebiegu programu doprowadzić do błędów wykonania. Jeszcze gorsza jest sytuacja, kiedy liczba instrukcji pop jest zbyt duża – to niemal zawsze prowadzi do załamania programu.

Instrukcje obsługi stosu

- pusha
- pushad
- pushf
- pushfd
- popa
- popad
- popf
- popfd

Instrukcja pusha

- Powoduje odłożenie na stos wszystkich 16-bitowych rejestrów ogólnego przeznaczenia.
- Instrukcja wykorzystywana jest głównie w 16-bitowych systemach operacyjnych takich jak DOS.
- Rejestry odkładane są na stos w następującej kolejności:

  AX, CX, DX, BX, SP, BP, SI, DI

Instrukcja pushad

- Powoduje odłożenie na stosie wszystkich 32-bitowych rejestrów ogólnego przeznaczenia.
- Zawartość rejestrów 32-bitowych odkładana jest na stosie w następującej kolejności:

  EAX, ECX, EDX, EBX, ESP, EBP, ESI, EDI

Instrukcje popa i popad

- To odpowiadające instrukcjom pusha i pushad instrukcje zdejmowania ze stosu całych grup rejestrów ogólnego przeznaczenia.
- Instrukcje te zachowują właściwy porządek zdejmowania ze stosu zawartości poszczególnych rejestrów, odwrotny do kolejności ich odkładania.

pushf, pushfd, popf, popfd

Powodują odpowiednio: umieszczenie i zdjęcie ze stosu rejestru znaczników FLAGS (EFLAGS).

Instrukcje te pozwalają na zachowanie słowa stanu programu na czas wykonania pewnej sekwencji instrukcji.

Niestety nie powodują one zachowania wartości pojedynczych znaczników.

Na stosie można zachowywać jedynie wszystkie znaczniki naraz.

Rejestr znaczników można przywrócić tylko w całości.

Usuwanie danych ze stosu bez ich zdejmowania

Realizowane poprzez ingerencję w wartość rejestru wskaźnika stosu.

Rejestr ESP przechowuje wartość wskaźnika stosu, czyli szczytowego elementu stosu.

Wystarczy dostosować tę wartość tak, aby wskaźnik stosu wskazywał na niższy, kolejny element stosu.

Usuwanie danych ze stosu bez ich zdejmowania

Ze szczytu stosu należy usunąć dwie wartości o rozmiarze podwójnego słowa.

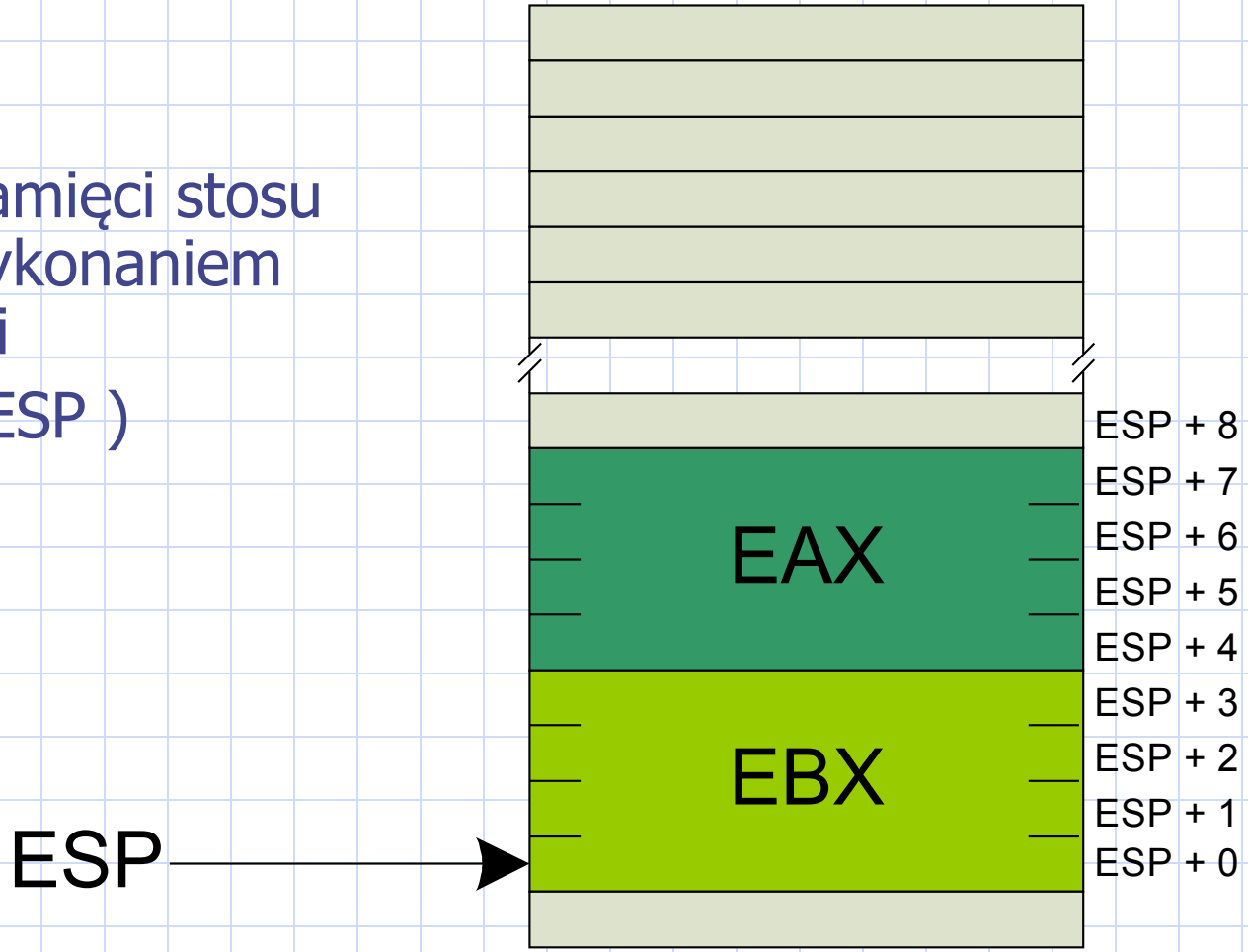
Efekt usunięcia ich ze stosu można osiągnąć dodając do wskaźnika stosu liczbę "osiem".

```
push( EAX );
push( EBX );
if(pewien_warunek); then
            add( 8, ESP );
else
pop( EBX );
pop( EAX );
```

Usuwanie danych ze stosu

Obraz pamięci stosu przed wykonaniem instrukcji

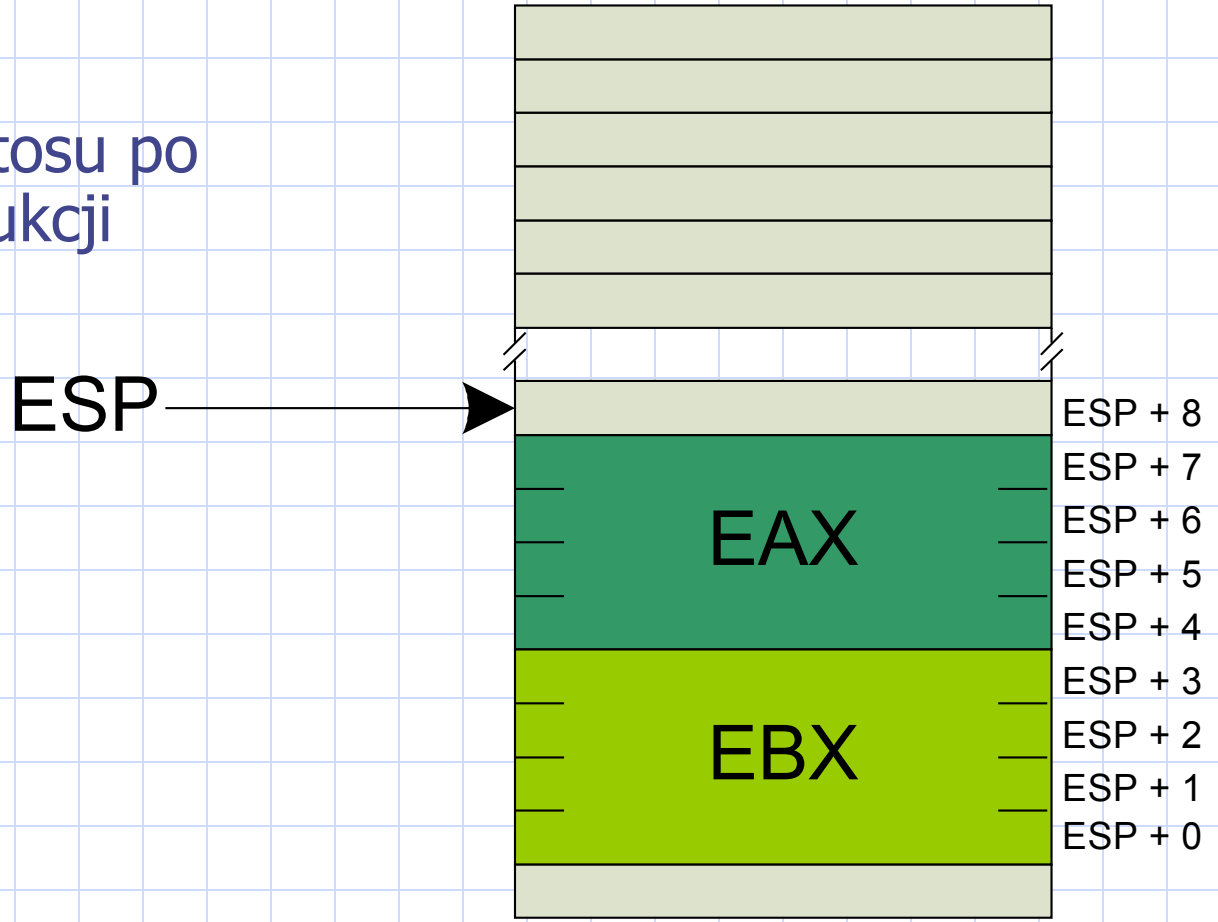
add(8, ESP)



Usuwanie danych ze stosu

Obraz pamięci stosu po wykonaniu instrukcji

add(8, ESP)



Instrukcje inc oraz dec

Jedną z najczęściej występujących operacji w języku asemblerowym jest zwiększanie bądź zmniejszanie o jeden wartości rejestru czy zmiennej w pamięci. Do powyższego służą następujące instrukcje:

```
inc(rej/pam);
dec(rej/pam);
```

Operandem może być dowolny rejestr 8-, 16-, 32-bitowy albo dowolny operand pamięciowy.

inc-dec oraz add-sub

- Instrukcje inc-dec realizowane są nieco szybciej niż odpowiadające im add-sub.
- Zapis inc-dec w kodzie maszynowym jest również bardzo prosty (tylko jeden operand).
- Manipulowanie wartością operandu za pośrednictwem inc-dec nie wpływa na wartość znacznika przeniesienia.

koniec laboratorium 3