



**POLITECHNIKA LUBELSKA  
WYDZIAŁ ELEKTROTECHNIKI  
I INFORMATYKI**

**KIERUNEK STUDIÓW  
INFORMATYKA**

***MATERIAŁY DO ZAJĘĆ  
LABORATORYJNYCH***

Architektura komputerów i programowanie niskopoziomowe

Dr inż. Krzysztof Skorupski

Lublin 2020

## **INFORMACJA O PRZEDMIOCIE**

### **Cele przedmiotu:**

- Cel 1. Poznanie architektury komputerów
- Cel 2. Zapoznanie z zasadami programowania niskopoziomowego
- Cel 3. Nabycie umiejętności obsługi systemów wieloprocesorowych

### **Efekty kształcenia w zakresie umiejętności:**

- Efekt 1. Znajomość architektury komputerów
- Efekt 2. Znajomość podstawowych przykładów organizacji komputerów
- Efekt 3. Znajomość organizacji jednostki centralnej
- Efekt 4. Znajomość niskopoziomowych struktur sterujących wykonaniem programu
- Efekt 5. Umiejętność posługiwania się podstawowymi operacjami arytmetycznymi
- Efekt 6. Umiejętność wykorzystywania podstawowych funkcji API
- Efekt 7. Umiejętność obsługi systemów wieloprocesorowych
- Efekt 8. Umiejętność programowania równoległego z wykorzystaniem pamięci współdzielonej i rozproszonej
- Efekt 9. Student zna potrzebę ciągłego pogłębiania i zdobywania wiedzy, jak też dzielenia się nią z innymi osobami

### **Literatura do zajęć:**

#### Literatura podstawowa

1. Tanenbaum A. S., Strukturalna organizacja systemów komputerowych. Helion, 2006.
2. Nissan N., Schocken S., Elementy systemów komputerowych- budowa nowoczesnego komputera od podstaw, WNT 2008.
3. Surtel W., Wójcik W., Kisała P., W: Architektura komputerów i systemy operacyjne; [Red:] Wójcik Waldemar - Lublin: Komitet Inżynierii Środowiska PAN, 2011.
4. R. Hyde, The Art. of Assembly Language, 2004.
5. Ogrodzki J., Wstęp do systemów komputerowych, oficyna Wydawnicza Politechniki Warszawskiej 2005.

#### Literatura uzupełniająca

1. W. Stallings, Organizacja i architektura systemu komputerowego. Projektowanie systemu a jego wydajność. WNT, W-wa, 2004
2. A. Błaszczuk, Win32ASM. Assembler w Windows, 2004
3. Null L., Lobur J., Struktura organizacyjna i architektura systemów komputerowych, Helion 2004.
4. Intel® 64 and IA-32 Architectures Software Developer's Manual, 2009.

### **Metody i kryteria oceny:**

#### Oceny cząstkowe:

- Ocena 1 Przygotowanie merytoryczne do zajęć laboratoryjnych na podstawie: wykładów, literatury, pytań kontrolnych do zajęć.
- Ocena 2 Wykonanie na ocenę wybranych poleceń podczas realizacji zajęć laboratoryjnych



**Fundusze Europejskie**  
Wiedza Edukacja Rozwój



**Rzeczpospolita  
Polska**

**Unia Europejska**  
Europejski Fundusz Społeczny



Ocena końcowa - zaliczenie przedmiotu:

- Pozytywne oceny cząstkowe.

**Plan zajęć laboratoryjnych:**

Lab1.	Wstęp do języka HLA
Lab2.	Instrukcje warunkowe i pętle w HLA
Lab3.	Sposoby reprezentacji danych w programowaniu niskopoziomowym
Lab4.	Liczby ze znakiem i bez znaku
Lab5.	Działania na liczbach niecałkowitych
Lab6.	Deklaracje i odwołania do tablic jednowymiarowych, wykorzystanie trybów adresowania
Lab7.	Deklaracje i odwołania do tablic wielowymiarowych, wykorzystanie trybów adresowania
Lab8.	Tworzenie programów zawierających procedury, rola stosu w funkcjach i procedurach
Lab9.	Tworzenie programów zawierających parametryczne funkcje i procedury
Lab10	Tworzenie okien i komunikatów. Programowanie równoległe z wykorzystaniem pamięci współdzielonej i rozproszonej. Programowanie systemów wieloprocesorowych.

## LABORATORIUM 1. WSTĘP DO JĘZYKA HLA.

### Cel laboratorium:

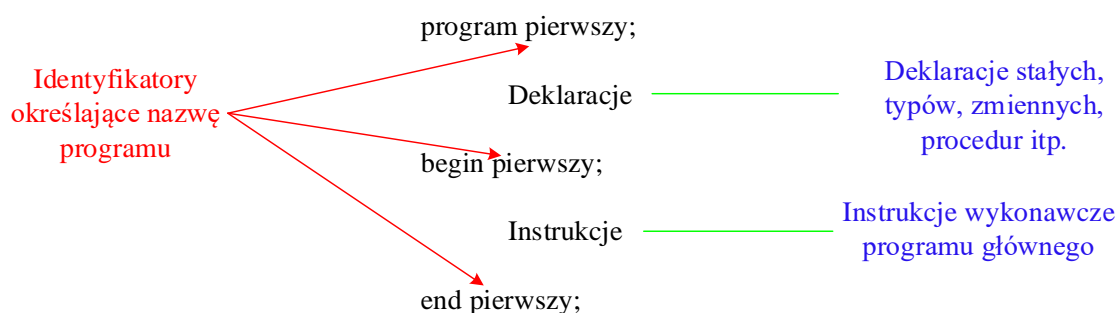
Celem zajęć jest rozpoczęcie programowania w języku assemblerowym.

### Zakres tematyczny zajęć:

- struktura programu w języku HLA
- podstawowe deklaracje w języku HLA
- pierwsze uruchomienie programu w języku assemblerowym.

### Struktura programu HLA

Struktura typowego programu HLA została przedstawiona na rysunku 1.



Rys 1.1 Struktura programu HLA

Ograniczenia stosowania identyfikatorów w HLA:

- identyfikator może zaczynać się od znaku podkreślenia lub litery
- identyfikator powinien zachowywać wielkość wpisanych liter
- nie można stworzyć dwóch identyfikatorów różniących się wielkością liter
- nie można używać nazw procedur i poleceń używanych w języku HLA.

### Zadanie 1.1. Pierwsze programy

Napisz pierwszy program w języku HLA, którego zadaniem będzie wyświetlanie na ekranie dowolnego tekstu.

Program należy napisać używając aplikacji Notatnik, WordPad lub Notepad++.

### Polecenie 1.

Napisz program o następującej treści:

```
program pierwszy;  
  
#include( "stdlib.hhf" );  
  
begin pierwszy;  
  
    stdout.put ( "To jest nasz pierwszy program" );  
  
end pierwszy;
```

Instrukcja `#include` służy do włączenia przez kompilator do kodu programu biblioteki standardowej HLA, której plik `stdlib.hhf` jest plikiem nagłówkowym.

Użyta instrukcja `stdout.put` służy do wyświetlenia napisów na ekranie.

### Polecenie 2.

Wyświetl zawartość pliku nagłówkowego `stdlib.hhf` i sprawdź czy zawiera on instrukcję włączenia procedury `stdout.put`. Plik nagłówkowy znajduje się typowo w następującej lokalizacji:

```
c:\hla\include
```

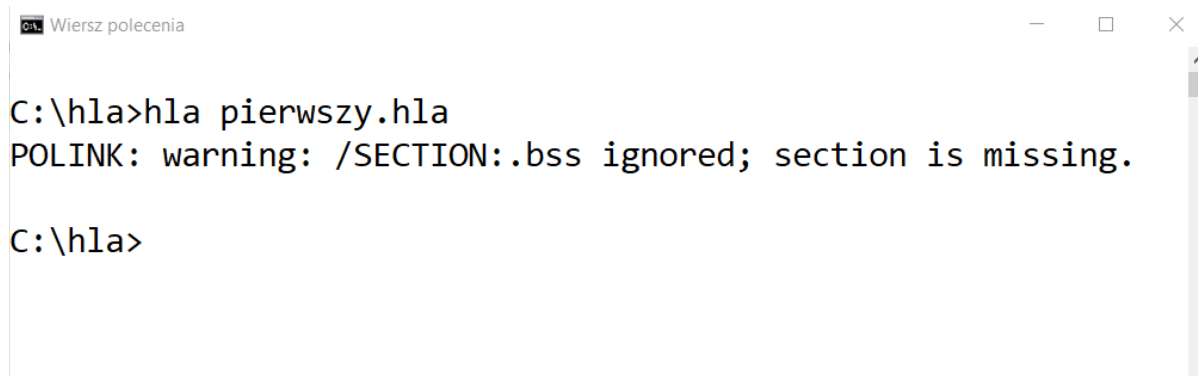
### Polecenie 3.

Kompilacja programu.

Kompilator języka HLA jest typowym kompilatorem wiersza poleceń, może być zatem uruchamiany z poziomu wiersza poleceń (Windows) czy też powłoki `bash` (Linux). Przed kompilacją kod programu należy zapisać w pliku o rozszerzeniu `*.hla`. Plik należy umieścić w katalogu programu HLA, typowo w lokalizacji:

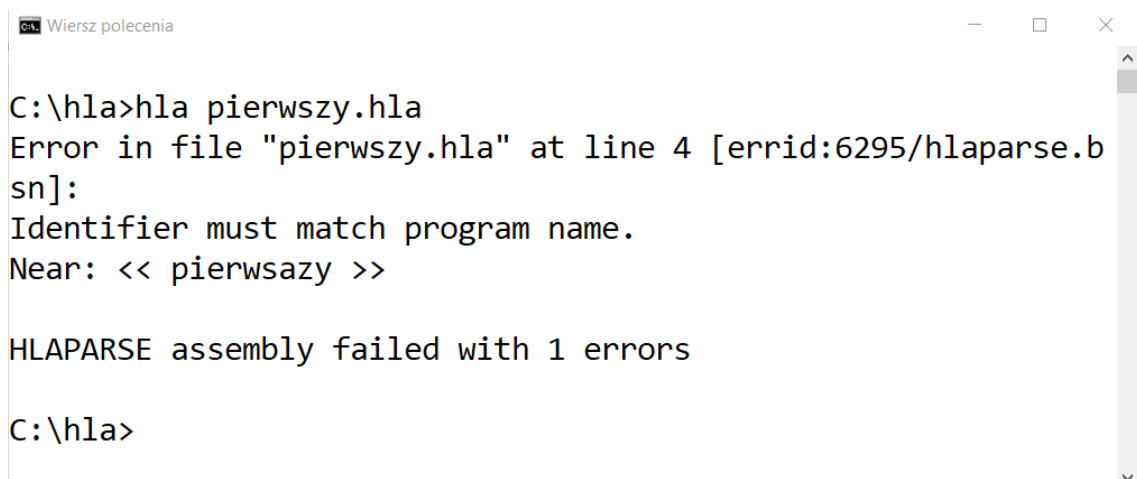
```
c:\hla
```

Korzystając z nawigacji w wierszu poleceń należy przejść do katalogu HLA a następnie wykonać kompilację:



```
Wiersz polecenia  
  
C:\hla>hla pierwszy.hla  
POLINK: warning: /SECTION:.bss ignored; section is missing.  
  
C:\hla>
```

W przypadku błędu kompilacji wskazana zostanie linia kodu i przyczyna błędu:

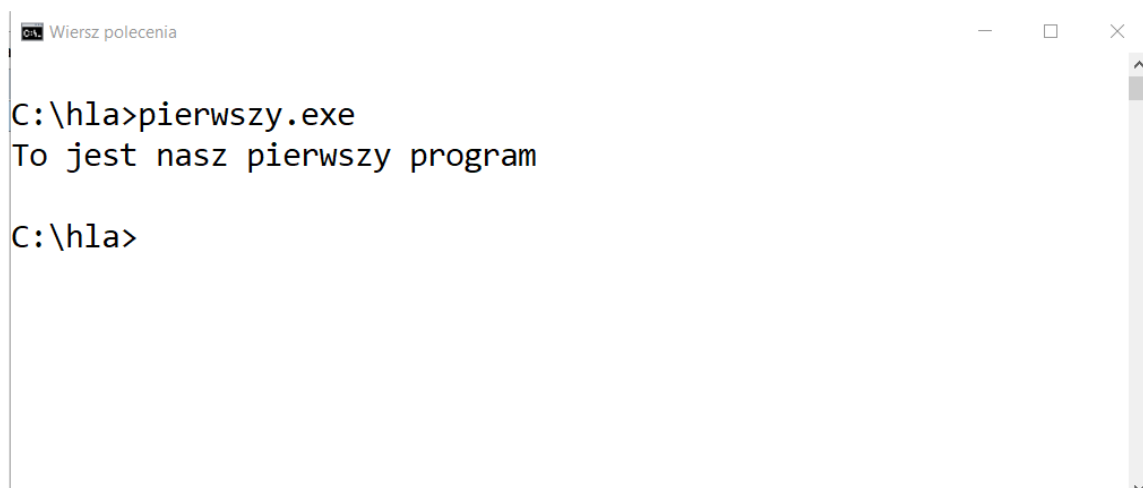


```
C:\hla>hla pierwszy.hla
Error in file "pierwszy.hla" at line 4 [errid:6295/hlaparse.b
sn]:
Identifier must match program name.
Near: << pierwszy >>

HLAPARSE assembly failed with 1 errors

C:\hla>
```

Poprawne skompilowanie kodu spowoduje utworzenie aplikacji wiersza poleceń. Pierwsze uruchomienie programu przeprowadzamy następująco:



```
C:\hla>pierwszy.exe
To jest nasz pierwszy program

C:\hla>
```

## **Zadanie 1.2 Deklaracje zmiennych**

### **Polecenie 1**

Napisz program, w którym zadeklarowane zostaną dwie zmienne całkowite. Jednej ze zmiennych zostanie przyporządkowana wartość początkowa. Wartość drugiej zmiennej powinna być wprowadzona z klawiatury po uruchomieniu programu. Program powinien następnie wyświetlić obie zmienne wraz z informacją, która zmienna jest wyświetlana.

W programie należy użyć następujące procedury:

*Stdout.put* - procedura wyprowadzania napisów na standardowe wyjście programu, dostępna w module obsługi standardowego wyjścia.

Składnia procedury:

```
stdout.put (lista wyprowadzanych wartości);
```

Lista argumentów wywołania procedury `stdout.put` może zostać konstruowana ze stałych, rejestrów i zmiennych. Kolejne argumenty oddziela się przecinkami.

Przykład:

```
stdout.put ("wprowadzona wartosc: ", zmienna1, nl, "zadeklarowana  
wartosc poczatkowa: ", zmienna2);
```

Użyty parametr `nl` jest stałą w HLA oznaczającą przejście do nowego wiersza.

`Stdin.get` - procedura odczytuje wartość wprowadzaną ze standardowego urządzenia wejściowego (zwykle klawiatura), konwertuje ją do postaci całkowitej i przypisuje otrzymaną wartość do zmiennej określonej parametrem wywołania

Składnia:

```
stdin.get (nazwa zmiennej);
```

### **Typy zmiennych całkowitych:**

Zmienna całkowita ze znakiem:

- `int8` - liczba całkowita 8-bitowa przyjmująca wartości z przedziału od -128 do 127
- `int16` - liczba całkowita 16-bitowa przyjmująca wartości z przedziału od -32768 do 32767
- `int32` - liczba całkowita 32-bitowa przyjmująca wartości z przedziału od -2147483648 do 2147483647

Zmienna całkowita bez znaku:

- `uns8` - liczba całkowita 8-bitowa przyjmująca wartości z przedziału od 0 do 255
- `uns16` - liczba całkowita 16-bitowa przyjmująca wartości z przedziału od 0 do 65536
- `uns32` - liczba całkowita 32-bitowa przyjmująca wartości z przedziału od 0 do 4294967296

Zmienne powinny zostać zadeklarowane w sekcji deklaracji zmiennych statycznych:

```
static  
    zmienna1: int8;  
    zmienna2: int16;  
    zmienna3: int32;
```

W sekcji deklaracji zmiennych statycznych można nadać deklarowanej zmiennej wartość początkową. Wartość ta zostanie przypisana do zmiennej podczas wczytywania programu do pamięci przez system operacyjny. Przykład zadeklarowania wartości początkowej:

```
static  
    zmienna1: int8      := 9;  
    zmienna2: int16     := 4532;  
    zmienna3: int32     := -614254;
```

Zmienną logiczną w HLA deklaruje się, określając w miejsce typu typ *boolean*.  
Przykład:





```
static
    zmiennalogiczna:    boolean;
    falszlogiczny:      boolean := false;
    prawdalogiczna:     boolean := true;
```

Zmiennej `zmiennalogiczna` nie przyporządkowano wartości początkowej (domyślnie 0), Pozostałym zmiennym przyporządkowano wartość fałszu i prawdy logicznej. Jako że zmienne logiczne są obiektami jednobajtowymi, można nimi manipulować przy wykorzystaniu dowolnych instrukcji operujących bezpośrednio na operandach ośmiobitowych.

Uwaga! Procedura `stdin.get` nie obsługuje wprowadzenia wartości do zmiennej logicznej.

## **Polecenie 2**

Zmodyfikuj poprzedni program w taki sposób, aby zostały w nim zadeklarowane i wyświetlone wartości zmiennych logicznych.

### **Zmienne znakowe**

W HLA możliwa jest również deklaracja zmiennych znakowych. Wartości znakowe są obiektami jednobajtowymi. Typ obiektów, którymi są wartości znakowe to: *char*. Zmienne znakowe można inicjalizować literałami znakowymi. Literały takie należy ograniczyć znakami pojedynczego cudzysłowu.

Przykład:

```
static
    zmiennaznakowa1:    char;
    zmiennaznakowa2:    char := 'A';
```

W HLA rozróżnia się również ciągi znaków, będące łańcuchami. Umieszcza się je w podwójnym cudzysłowie.

"To jest łańcuch znaków"

który należy rozróżnić od zmiennej znakowej. Łańcuch znaków nie jest zmienną i w szczególności należy zwrócić uwagę na zapis na przykładzie znaku A:

'A' ≠ "A"

## **Polecenie 3**

Napisz program o nazwie `znaki`, w którym zadeklarowany zostanie znak A, i następnie zostanie on wyświetlony na ekranie. Następnie program powinien umożliwić pobranie znaku z klawiatury i wyświetlić go również na ekranie.

### **Zadanie 1.3. Wykorzystanie rejestrów ogólnego przeznaczenia oraz podstawowe instrukcje maszynowe**

#### **Rejestry ogólnego przeznaczenia**

Rejestry procesorów 80x86 możemy podzielić na następujące grupy:

- rejestry ogólnego przeznaczenia
- rejestry specjalne trybu użytkownika
- rejestry segmentowe
- rejestry specjalne trybu nadzoru

W tworzonych programach wykorzystane zostaną rejestry ogólnego przeznaczenia. Dostępne są następujące rejestry:

8 rejestrów 32-bitowych:

- EAX, EBX, ECX, EDX, ESI, EDI, EBP, ESP.  
Przedrostek E od extended, rozróżnia on rejestr 32 od 16-bitowych
- 8 rejestrów 16-bitowych:  
AX, BX, CX, DX, SI, DI, BP, SP.
- 8 rejestrów 8-bitowych:  
AL, AH, BL, BH, CL, CH, DL, DH.

Sposób wykorzystania tych rejestrów zależy wyłącznie od programisty. Za pewne ograniczenie można uznać jedynie fakt, że część rejestrów ogólnego przeznaczenia ma zwyczajowo przypisane określone wykorzystanie. Wśród tych najistotniejszy jest rejestr ESP określający wskaźnik stosu oraz rejestr EBP, który jest rejestrem bazowym służącym do adresowania.

Ważną cechą rejestrów ogólnego przeznaczenia jest brak ich niezależności. Polega ona na tym, że rejestr 32-bitowy zawiera w sobie rejestr 16-bitowy. Rejestr 16-bitowy może natomiast zawierać w sobie dwa rejestry 8 bitowe. Przykładowo rejestr EAX zawiera rejestr AX, który stanowi jego 16 młodszych bitów. Osiem najstarszych bitów rejestru AX stanowi z kolei rejestr AH, a 8 młodszych rejestr AL. Wykonując operacje z wykorzystaniem rejestrów należy zwrócić uwagę, że zmiana wartości jednego rejestru może pociągać za sobą zmianę wartości innych, zależnych od niego, rejestrów. Rysunek 1.1 przedstawia schematycznie architekturę rejestrów ogólnego przeznaczenia procesora x86.





Rys 1.2 Architektura rejestrów ogólnego przeznaczenia procesora x86

W systemach assemblerowych rejestry pełnią bardzo istotną funkcję. W zasadzie każda operacja angażuje rejestry. Dla przykładu, aby dodać do siebie 2 wartości i umieścić ich sumę w trzeciej, należy załadować jeden ze składników do rejestru, dodać do niego (w rejestrze) drugi składnik sumy i dopiero potem wynik skopiować z rejestru do miejsca przechowywania sumy. Rejestry stanowią więc bazę wszelkich obliczeń.

### Podstawowe instrukcje maszynowe

Instrukcja maszynowa *mov*

Instrukcja służy do przemieszczania danych pomiędzy lokacjami:

```
mov(operand źródłowy, operand docelowy);
```

Operandem źródłowym może być zmienna, stała (wartość) bądź rejestr. Operandem docelowy może być zmienna lub rejestr. Ponieważ lista instrukcji procesorów x86 nie obejmuje instrukcji, której obydwie operandy przechowywane byłyby w pamięci, jeden z operandów musi być rejestrem. Na przykład:

```
mov(zmienna1, eax);
mov(al, zmienna2);
mov(256, zmienna3);
```

Równie istotnym ograniczeniem jest fakt, że operandy muszą być tych samych rozmiarów.

### Instrukcja maszynowa *add*

Służy do dodawania danych:

```
add(operand źródłowy, operand docelowy);
```



Fundusze Europejskie  
Wiedza Edukacja Rozwój



Rzeczpospolita  
Polska

Unia Europejska  
Europejski Fundusz Społeczny



### Instrukcja maszynowa *sub*

Służy do odejmowania danych:

```
sub(operand źródłowy, operand docelowy);
```

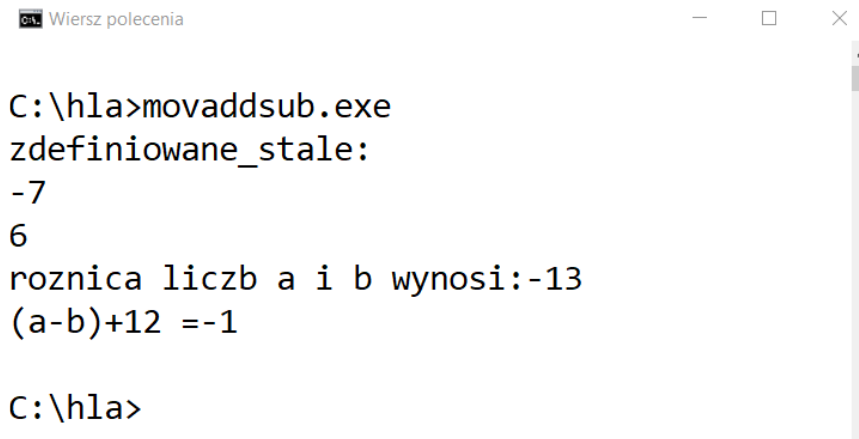
Ograniczenia dotyczące formatów danych są takie same jak dla instrukcji *mov*. Wykonując ćwiczenia należy zwrócić uwagę, w którym operandzie zapisywany jest wynik działania.

### Polecenie 1

Napisz program o nazwie *movaddsub*, w którym należy:

- Zadeklarować dwie zmienne *a* i *b* o długości 1-bajta i wartościach odpowiednio -7,6.
- Wyświetlić zmienne na ekranie.
- Wykonać operację odejmowania (*a-b*) na zmiennych. Wyświetlić wyniki operacji.
- Do uzyskanej wartości dodać 12.
- Wyświetlić wyniki dodawania.

Przykładowy wynik działania programu:

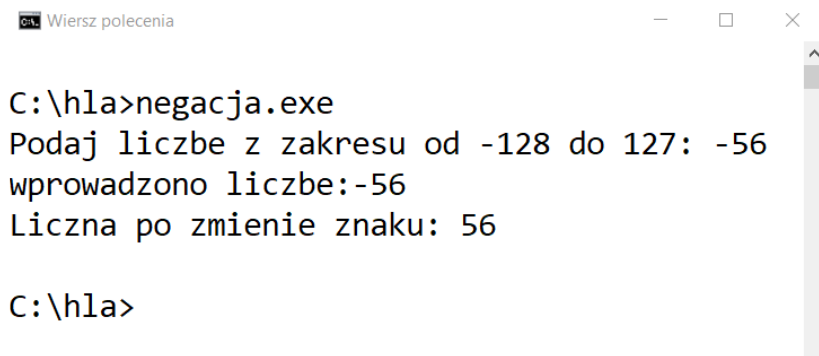


```
C:\hla>movaddsub.exe
zdefiniowane_stale:
-7
6
roznica liczb a i b wynosi:-13
(a-b)+12 =-1

C:\hla>
```

### Polecenie 2

Napisz program, który w oparciu o poznane polecenia dokona negacji wartości wprowadzonej z klawiatury. Przykładowy wynik działania programu:



```
C:\hla>negacja.exe
Podaj liczbe z zakresu od -128 do 127: -56
wprowadzono liczbe:-56
Liczna po zmienie znaku: 56

C:\hla>
```

## **LABORATORIUM 2. INSTRUKCJE WARUNKOWE I PĘTLE W HLA.**

Cel laboratorium:

Celem zajęć jest programowanie z wykorzystaniem wybranych struktur HLA.

Zakres tematyczny zajęć:

- instrukcje warunkowe w HLA
- pętle w HLA

### **Zadanie 2.1 Operacje na rejestrach ogólnego przeznaczenia**

#### **Polecenie 1**

Napisz program, w którym należy:

- Zadeklarować zmienne o długości 1,2 i 3 bajtów
- Wartości poszczególnych zmiennych:  
zmienna 1-bajtowa = -7  
zmienna 2-bajtowa = -277  
zmienna 3-bajtowa = -66000
- Dokonać ich negacji oparciu o rejestry ogólnego przeznaczenia:  
odpowiednio AL, AX, EAX  
Negacji należy dokonać w oparciu o poznane operacje arytmetyczne
- Dodać do wartości zanegowanej zmiennej  
3-bajtowej wartość 666
- Wyświetlić za pomocą jednej procedury trzy zmienne.

Wykonując zadanie należy zwrócić uwagę na brak niezależności wykorzystywanych rejestrów ogólnego przeznaczenia, co wpływa na kolejność wykonywania fragmentów kodu.

Przykładowe działanie programu:

```
Wiersz polecenia

C:\hla>movaddsub2.exe

Początkowe wartosci: liczba jednobajtowa= -7 slowo= -277 dwuslowo= -66000

Po zanegowaniu: liczba jednobajtowa= 7 slowo= 277 dwuslowo= 66000

Po wykonaniu (ADD 666 + dwuslowo zanegowane) uzyskalismy: 66666

C:\hla>
```

## Zadanie 2.2 Instrukcje warunkowe oraz pętle w HLA

### Instrukcje *inc* oraz *dec*

Jedną z najczęściej występujących operacji w języku assemblerowym 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. 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).

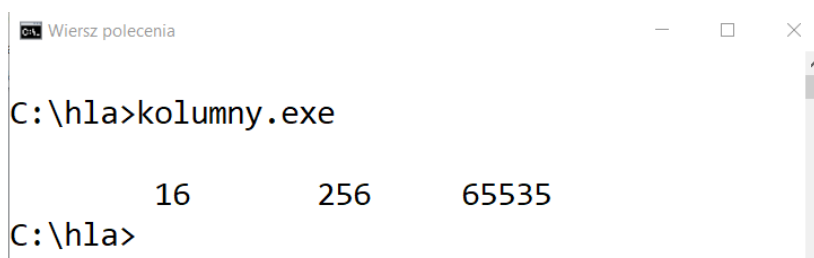
Nowa funkcjonalność procedury *stdout.put*:

- Lista argumentów może zostać skonstruowana ze stałych, rejestrów i zmiennych, kolejne argumenty oddziela się przecinkami.
- Każdy z argumentów wywołania może być zadany w jednej z dwóch postaci:
  - wartość
  - wartość:szerokość

Wpisując po dwukropku liczbę, określa się minimalną szerokość napisu reprezentującego wartość. Na przykład wykonanie kodu:

```
...  
static  
    zmienna1: int32:=16;  
    zmienna2: int32:=256;  
    zmienna3: int32:=65535;  
...  
...  
stdout.put(zmienna1:10,zmienna2:10,zmienna3:10);  
...
```

Spowoduje wyświetlenie wartości zmiennych w następującej postaci:



```
C:\hla>kolumny.exe  
16 256 65535  
C:\hla>
```

### Pętla *while* w HLA:

```
while (warunek) do  
  
    instrukcja  
    bądź ich  
    cały blok  
  
endwhile;
```

#### Polecenie 1

Wykorzystując pętlę *while* napisz program, który w jednym wierszu wyświetli liczby od 10 do 1

Przykład działania programu:



```
C:\hla>szerokosc  
10 9 8 7 6 5 4 3 2 1  
C:\hla>
```

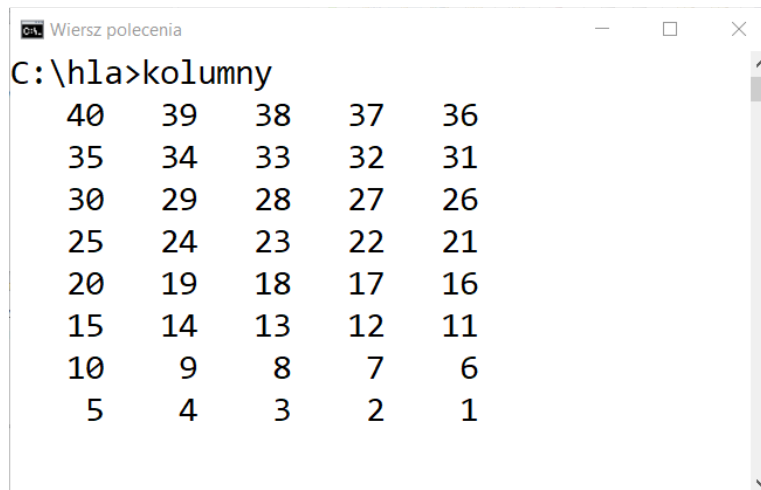
### Instrukcja *for* w HLA:

```
for(wyrażenie inicjalizujące; warunek; instrukcja licznika)  
do  
    instrukcja  
    bądź ich  
    cały blok  
endfor;
```

#### Polecenie 2

Wykorzystując instrukcje *while* i *for* ułożyć liczby od 40-1 w 5 kolumnach i 8 wierszach

Przykład działania programu:



```
C:\hla>kolumny
40  39  38  37  36
35  34  33  32  31
30  29  28  27  26
25  24  23  22  21
20  19  18  17  16
15  14  13  12  11
10   9   8   7   6
 5   4   3   2   1
```

W programie można użyć procedurę *stdout.newln()*. Procedura ta nie przyjmuje argumentu i pozwala na przejście do nowej linii. Jej działanie odpowiada użyciu procedury *stdout.put(nl)*.

### Instrukcja *if* w HLA:

```
if(wyrażenie logiczne) then
    instrukcja
    bądź ich
    cały blok
else
    instrukcja
    bądź ich
    cały blok
endif;
```

### Polecenie 3

Zmodyfikuj program z polecenia 2 tak, by wykorzystywał instrukcję *if*.



## LABORATORIUM 3. SPOSOBY REPREZENTACJI DANYCH W PROGRAMOWANIU NISKOPOZIOMOWYM.

Cel laboratorium:

Celem zajęć jest zapoznanie ze sposobami reprezentowania danych w programowaniu niskiego poziomu.

Zakres tematyczny zajęć:

- sposoby reprezentacji danych
- konwersja między systemami liczbowymi

### Zadanie 3.1 Konwersja między systemami liczbowymi

Procedury wejścia-wyjścia w językach programowania niskiego poziomu różnią się formatem danych wejściowych i wyjściowych w zależności od typu danych, które są ich argumentem.

Dla procedur *stdin.get* i *stdout.put*:

- dane typu *int* – reprezentowane decymalnie
- dane zawarte w rejestrach 8, 16, 32 bitowych – reprezentowane heksadecymalnie
- dane typu *byte*, *word*, *dword*, *qword*, *lword* – reprezentowane heksadecymalnie.

Oznacza to, że w zależności od typu danych na standardowym wejściu i wyjściu programu pojawią się dane w systemie liczbowym dziesiętnym lub szesnastkowym.

Wykonanie kodu:

```
program prog3;
#include( "stdlib.hhf" );
static
    zmienna1: int8;
    zmienna2: byte;
begin prog3;
    mov(15,al);
    mov(15,zmienna1);
    mov(15,zmienna2);
    stdout.put ( "Wartosc wyswietlana ze zmiennej typu
int8 to: ",zmienna1 );
    stdout.newln();
    stdout.put ( "Wartosc wyswietlana ze zmiennej typu
byte to: ",zmienna2 );
    stdout.newln();
    stdout.put ( "Wartosc wyswietlana z rejestru al to:
",al );
end prog3;
```

Spowoduje wyświetlenie wartości ze zmiennej *int* w dziesiętnym systemie liczbowym oraz wartości z rejestru w systemie szesnastkowym:



```

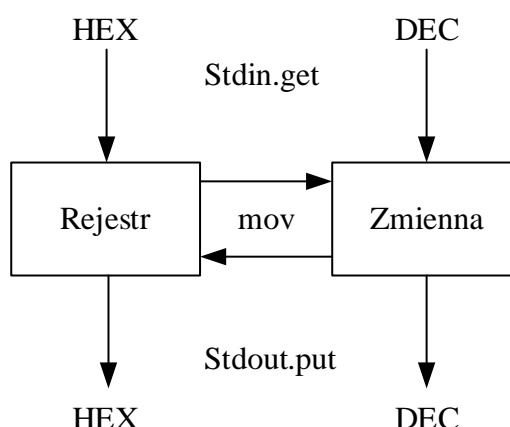
C:\hla>prog3.exe
Wartosc wyswietlana ze zmiennej typu int8 to: 15
Wartosc wyswietlana ze zmiennej typu byte to: 0F
Wartosc wyswietlana z rejestru al to: 0F
C:\hla>
    
```

### Polecenie 1

Napisz program wykorzystujący procedurę *stdin.get*, w którym dane są prowadzane do rejestru AL i zmiennej *int8*, a następnie za pomocą procedury *stdout.put* są wyświetlane w konsoli.

### Polecenie 2

Wykorzystując przedstawione powyżej własności procedur wejścia-wyjścia, napisz program, który umożliwia konwersję liczby z systemu szesnastkowego na dziesiętny według poniższego schematu:



Rys 3.1 Schemat konwersji pomiędzy systemami liczbowymi

Przykładowe działanie programu:

```

C:\hla>hex_do_dec.exe
Wprowadz liczbe w zapisie szesnastkowym: ff
Wartosc 000000FFH w zapisie dziesiętnym to: 255
C:\hla>
    
```

Uwaga! Dla rozróżnienia wartości dla różnych systemów liczbowych przyjmuje się, że do liczby wyświetlanej w systemie szesnastkowym dodaje się na końcu literę H.

### Zadanie 3.2 Inne procedury wejścia-wyjścia.

Istnieje możliwość wyświetlania zawartości rejestrów w postaci dziesiętnej. Służy do tego procedura *stdout.putiN*. Gdzie N oznacza ilość bitów, adekwatną do użytego argumentu.

Na przykład procedura *stdout.puti8* traktuje przekazany w wywołaniu argument jako liczbę całkowitą ze znakiem i wyprowadza ją w zapisie dziesiętnym. Do procedury można przekazać 8-bitowy argument, również 8-bitowy rejestr. Dla obiektów 16 i 32-bitowych należy zastosować procedury *stdout.puti16* oraz *stdout.puti32*. Wszystkie wyżej wymienione procedury przyjmują jeden argument.

Przeanalizuj działanie programu:

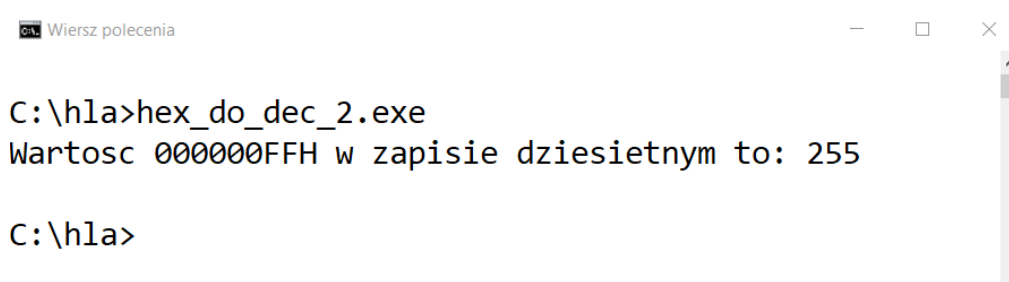
```
program hex_do_dec_2;
#include( "stdlib.hhf" );

begin hex_do_dec_2;

    mov(255,eax);
    stdout.put ( "Wartosc ",eax,"H"," w zapisie dziesiętnym
to: " );
    stdout.puti32( eax );
    stdout.newln();

end hex_do_dec_2;
```

Wykonanie programu spowoduje wyświetlenie wartości z rejestru *eax* w dziesiętnym systemie liczbowym:



```
C:\hla>hex_do_dec_2.exe
Wartosc 000000FFH w zapisie dziesiętnym to: 255

C:\hla>
```

#### Polecenie 2

Napisać program dokonujący konwersji pomiędzy systemem szesnastkowym a dziesiętnym bez angażowania dodatkowych zmiennych.

#### Procedura *stdin.getiN*

Procedura *stdin.get* przyjmuje taką samą podstawę systemu liczbowego dla danych wejściowych jak procedura *stdout.put* dla danych wyprowadzanych. Jeśli więc chodzi o wczytanie wartości do umieszczenia w zmiennej *int8*, *int16*, *int32* procedura spodziewa się



wprowadzenia wartości w zapisie dziesiętnym. Gdy wczytywana zmienna ma być umieszczona w rejestrze lub zmiennej typu byte, word, dword, spodziewana jest wartość w zapisie szesnastkowym.

Aby zmienić przyjmowaną domyślnie podstawę systemu liczbowego dla danych mających trafić do rejestrów należy skorzystać z wywołań:

- `stdin.geti8()`
- `stdin.geti16()`
- `stdin.geti32()`

Procedura nie przyjmuje argumentu. Jej działanie polega na umieszczeniu wartości podanej w dziesiętnym systemie liczbowym i umieszczenie jej w rejestrze:

Procedura	Rejestr
<code>stdin.geti8</code>	AL
<code>stdin.geti16</code>	AX
<code>stdin.geti32</code>	EAX

Przeanalizuj przykład:

```
program stdin_getin;
#include( "stdlib.hhf" );

static
    liczba:    int32;

begin stdin_getin;

    stdout.put ( "Wprowadz liczbe w zapisie dziesiętnym: " );
    stdin.geti32();
    mov( eax,liczba );
    stdout.put ( "Wprowadzona wartosc to: " );
    stdout.put( liczba );

end stdin_getin;
```

Działanie programu z wprowadzaną przykładową liczbą 15:

```

C:\hla>stdin_getin.exe
Wprowadz liczbe w zapisie dziesiętnym: 15
Wprowadzona wartosc to: 15
C:\hla>
    
```

### Procedura *stdin.gethN*

Działanie procedury polega na umieszczeniu wartości podanej w systemie szesnastkowym w rejestrze ogólnego przeznaczenia wg tabeli:

Procedura	Rejestr
stdin.geth8()	AL
stdin.geth16()	AX
stdin.geth32()	EAX

Procedura nie przyjmuje argumentu. Przykładowy kod programu wykorzystujący procedurę *stdin.gethN*:

```

program stdin_gethn;
#include( "stdlib.hhf" );

static
    liczba:    int32;

begin stdin_gethn;

    stdout.put ( "Wprowadz liczbe w zapisie szesnastkowym: "
);
    stdin.geth32();
    stdout.put ( "Wprowadzona wartosc wyswietlona z rejestru
EAX: " );
    stdout.put( eax );
end stdin_gethn;
    
```

Działanie programu:

```

C:\hla>stdin_gethn.exe
Wprowadz liczbe w zapisie szesnastkowym: FF
Wprowadzona wartosc wyswietlona z rejestru EAX: 000000FF
C:\hla>
    
```



### Procedura *stdout.puthN*

Jeżeli zachodzi potrzeba wyprowadzenia zmiennej typu *int8*, *int16*, *int32* w postaci szesnastkowej należy posłużyć się procedurami:

- *stdout.puth8(zmienna)*
- *stdout.puth16(zmienna)*
- *stdout.puth32(zmienna)*.

Przeanalizuj przykład:

```
program stdout_puthn;
#include( "stdlib.hhf" );

static
    liczba:    int16;

begin stdout_puthn;

    mov( 4095,liczba );
    stdout.put ( "Wartosc zmiennej liczba: " );
    stdout.puth16( liczba );

end stdout_puthn;
```

Działanie programu:

Wiersz polecenia

```
C:\hla>stdout_puthn.exe
Wartosc zmiennej liczba: FFF
C:\hla>
```

### Polecenie 3

Zaproponuj program, który dokona zapisu 32-bitowej liczby wprowadzonej w postaci szesnastkowej z użyciem procedury *stdin.gethN*, a następnie umieści tą liczbę w zmiennej typu *int* i wyświetli postać szesnastkowej w oparciu o procedurę *stdout.puthN*

## LABORATORIUM 4. LICZBY ZE ZNAKIEM I BEZ ZNAKU.

Cel laboratorium:

Celem zajęć jest zapoznanie ze systemem liczbowym używanym w procesorach x86 oraz poznanie instrukcji rozszerzenia bitowego

Zakres tematyczny zajęć:

- binarne systemy liczbowe
- zmiana znaku liczby
- instrukcje rozszerzenia bitowego
- instrukcje kopiowania z rozszerzeniem bitowym

### Zmiana znaku liczby z kodzie uzupełnienia do 2

#### Binarny system zapisu liczb

System dwójkowy należy do pozycyjnych systemów liczbowych, w którym podstawą jest liczba 2. System posiada dwie cyfry 0 i 1, zatem można je kodować bezpośrednio jednym bitem informacji. Wartości wag w systemie dwójkowym o liczbie bitów wynoszącej  $n$  są następujące:

bit	$b_{n-1}$	$b_{n-2}$	...	$b_2$	$b_1$	$b_0$
waga	$2^{n-1}$	$2^{n-2}$	...	$2^2$	$2^1$	$2^0$

gdzie  $b$  jest wartością 0 lub 1,  $n$  oznacza  $n$ -ty bit liczby liczony od bitu najmłodszego.

Wartość dziesiętną liczby zapisanej w systemie dwójkowym obliczamy następująco:

$$\text{Wartość dziesiętna} = b_{n-1} \cdot 2^{n-1} + b_{n-2} \cdot 2^{n-2} + \dots + b_1 \cdot 2^1 + b_0 \cdot 2^0$$

Zapis dwójkowy nie przewiduje możliwości zapisu liczb ujemnych. W celu zapisu liczby ujemnej wykorzystuje się system znak-moduł, w którym najstarszy bit określa czy liczba jest liczbą ujemną czy dodatnią. Główna wada tego systemu polega na tym, że jeden bit liczby jest zajęty tylko i wyłącznie na określenie jej znaku. Innym systemem jest system uzupełnienia do 1 nazywany U1.

Wartości wag w systemie uzupełnienia do 1 o liczbie bitów wynoszącej  $n$  są następujące:

bit	$b_{n-1}$	$b_{n-2}$	...	$b_2$	$b_1$	$b_0$
waga	$-2^{n-1}+1$	$2^{n-2}$	...	$2^2$	$2^1$	$2^0$

Wartość dziesiętną liczby zapisanej w systemie uzupełnienia do 1 obliczamy następująco:

$$\text{Wartość dziesiętna} = b_{n-1} \cdot (-2^{n-1} + 1) + b_{n-2} \cdot 2^{n-2} + \dots + b_1 \cdot 2^1 + b_0 \cdot 2^0$$



Najstarszy bit określa więc znak liczby ale dzięki przypisanej wadze określa też wartość liczby.

Wadą tego systemu zapisu jest to, że wykonywanie operacji arytmetycznych na zasadach zgodnych z systemem dwójkowym wymaga dodatkowych założeń - przy dodawaniu do wyniku należy dodać przeniesienie poza bit znaku, aby otrzymać poprawny wynik.

Do zapisu liczb ujemnych w procesorach z rodziny 80x86 przyjęto więc notację z uzupełnieniem do dwóch U2.

Wartości wag w kodzie U2 dla liczby o n-bitach:

bit	$b_{n-1}$	$b_{n-2}$	...	$b_2$	$b_1$	$b_0$
waga	$-2^{n-1}$	$2^{n-2}$	...	$2^2$	$2^1$	$2^0$

Jeśli bit najstarszy ma wartość zero, to liczba traktowana jest jako dodatnia i interpretuje się ją zgodnie z regułą systemu dwójkowego.

W przypadku kiedy najstarszy bit ma wartość jeden, liczba jest traktowana jako ujemna a jej zapis odpowiada notacji U2.

Wartość dziesiętną liczby zapisanej w systemie uzupełnienia do 2 obliczamy następująco:

$$\text{Wartość dziesiętna} = b_{n-1} \cdot (-2^{n-1}) + b_{n-2} \cdot 2^{n-2} + \dots + b_1 \cdot 2^1 + b_0 \cdot 2^0$$

Przykład obliczenia wartości dziesiętnej z liczby zapisanej w kodzie U2 :

$$11101010 = (-2^7) + 64 + 32 + 8 + 2 = -128 + 106 = -22$$

#### **Zadanie 4.1 Negacja wartości liczby zapisanej w kodzie uzupełnienia do 2**

Negacji liczby w kodzie U2 można dokonać poprzez inwersję jej bitów, a następnie dodanie jedności.

Inwersja bitów liczby binarnej:

```
01101110
10010001
```

Procesory x86 udostępniają polecenie niskopoziomowe *not* umożliwiające inwersję bitów:

```
not ( operand-docelowy );
```

Operandem musi być zmienną w pamięci lub rejestrem.

Kolejnym krokiem jest dodanie jedności:

```

      10010001
+     00000001
=     10010010
```





Procesory rodziny 80x86 udostępniają również specjalną instrukcję o nazwie *neg*, która realizuje konwersję do liczby odwrotnej w kodzie uzupełnienia do dwóch.  
Składnia polecenia *neg*:

```
neg( operand-docelowy );
```

Operandem musi być zmienna w pamięci lub rejestrze.

### **Polecenie 1**

Napisać program który:

- wczytuje liczby jako zmienne 1-bajtowe typu `int`.
- wyświetla wprowadzone liczby w postaci szesnastkowej
- wyświetla wprowadzone liczby w postaci hex po inwersji bitów
- wyświetla wprowadzone liczby w postaci hex po inwersji bitów i dodaniu jedności
- wyświetla wprowadzone liczby w postaci dec po inwersji bitów i dodaniu jedności
- wyświetla wprowadzone liczby w postaci dec w kodzie U2 po wykorzystaniu instrukcji „*neg*”

Przykładowe działanie programu:

 Wiersz polecenia

```
C:\>cd hla
```

```
C:\hla>negacja.exe
```

```
Podaj liczbe z zakresu od -128 do 127: -28
```

```
Szesnastkowo: E4
```

```
Po inwersji bitow: 1B
```

```
Po dodaniu jednosci: 1C
```

```
Wynik dziesietnie: 28
```

```
Ta sama operacja z wykorzystaniem instr. neg:
```

```
Wynik szesnastkowo: 1C
```

```
Wynik dziesietnie: 28
```

### **Zadanie 4.2 Rozszerzenie bitowe liczby zapisanej w kodzie U2**

Aby rozszerzyć wartość interpretowaną jako liczbę ze znakiem do dowolnej większej liczby bitów, należy skopiować bit znaku do wszystkich nadmiarowych bitów nowego formatu.



Przykład:

liczba 8-bitowa	liczba 16-bitowa	liczba 32-bitowa
80H (1000 0000)	FF80H	FFFF_FF80H
28H (0010 1000)	0028H	0000_0028H
9A (1001 1010)	FF9A	FFFF_FF9A
7F (0111 1111)	007F	0000_007F

Aby rozszerzyć wartość bez znaku, należy wykonać tak zwane rozszerzenie zerem. Starszy bajt (bajty) większego operandu są po prostu zerowane:

Przykład:

liczba 8-bitowa	liczba 16-bitowa	liczba 32-bitowa
80H (1000 0000)	0080H	0000_0080H
28H (0010 1000)	0028H	0000_0028H
9A (1001 1010)	009A	0000_009A
7F (0111 1111)	007F	0000_007F

Procesory rodziny 80x86 udostępniają programiście szereg instrukcji rozszerzania znakiem i zerem:

Instrukcja	Działanie
cbw();	rozszerza znakiem bajt z rejestru AL na rejestr AX
cwd();	rozszerza znakiem słowo z rejestru AX na rejestr DX:AX
cdq();	rozszerza znakiem podwójne słowo z rejestru EAX na rejestry EDX:EAX
cwde();	rozszerza znakiem słowo z rejestru AX na rejestr EAX

Zapis DX:AX oznacza wartość 32-bitową, której starsze słowo umieszczone jest w rejestrze DX, a młodsze w rejestrze AX

Możliwe jest również kopiowanie z rozszerzeniem bitowym. Instrukcja *movsx* przy kopiowaniu operandu źródłowego automatycznie rozszerza go znakiem do rozmiaru operandu docelowego.

`movsx( operand-źródłowy, operand-docelowy );`

Instrukcja wymaga spełnienia dwóch warunków:

- operand docelowy musi mieć rozmiar większy od operandu rozmiaru źródłowego,
- operand docelowy musi być rejestrem; tylko operand źródłowy może być zmienną w pamięci.



Możliwe jest również rozszerzenie zerem. Instrukcja *movzx* ma identyczną składnię i nakłada identyczne ograniczenia na operandy jak instrukcja *movsx*.

Rozszerzenie zerem niektórych 8-bitowych rejestrów (AL,BL,CL oraz DL) na odpowiednie rejestry 16-bitowe można łatwo osiągnąć bez pośrednictwa instrukcji *movzx* – wystarczy proste wyzerowanie starszych połówek tych rejestrów (AH, BH, CH i DH).

## Polecenie 2

Napisać program wyświetlający (w postaci dziesiętnej i szesnastkowej) wprowadzoną liczbę jednobajtową:

1. bez rozszerzenia
2. z rozszerzeniem do 16-bitów
3. z rozszerzeniem do 32-bitów

za pomocą:

1. instrukcji rozszerzenia
2. instrukcji kopiowania z rozszerzeniem

Przykład działania programu:

```
Wiersz polecenia

C:\hla>rozszerzenie.exe
Wprowadz liczbe calkowita od -128 do 127: -108

Rozszerzanie znakiem (instrukcje rozszerzenia):

Wprowadzono: -108 (94H)
Rozszerzenie do 16-bitow: -108 (FF94H)
Rozszerzenie do 32-bitow: -108 (FFFFFF94H)

Rozszerzenie znakiem (instrukcja kopiowania ze znakiem):

Rozszerzenie do 16-bitow: -108 (FF94H)
Rozszerzenie do 32-bitow: -108 (FFFFFF94H)

C:\hla>
```



### Zadanie 4.3 Dodatkowe struktury sterujące wykonaniem programu w HLA

#### Instrukcja *repeat*

Składnia instrukcji:

```
Repeat
```

```
    instrukcje
```

```
until (warunek) ;
```

Analogicznie do języków programowania wysokiego poziomu test warunku wykonywany jest po wykonaniu instrukcji umieszczonych w pętli. Oznacza to, że instrukcja zostanie wykonana przynajmniej jeden raz, nawet wtedy, gdy warunek nie jest spełniony.

#### Polecenie 1

Napisz program wykorzystujący instrukcje *repeat*.

#### Instrukcje *break* i *breakif*

Instrukcje te służą do przerywania wykonywania kodu umieszczonego w pętli. W kodzie instrukcji pętli należy umieścić polecenia:

```
break;
```

```
breakif (warunek) ;
```

W przypadku polecenia *break* jego naturalnym zastosowaniem jest umieszczenie w alternatywie instrukcji *If*. Np.:

```
if (warunek) then
    //instrukcje
else
    break;
endif;
```

#### Instrukcja *forever*

Instrukcja *forever* tworzy pętlę nieskończoną:

```
forever
    //instrukcje
endfor;
```

Do przerywania wykonywania instrukcji pętli można użyć polecenia *breakif*.

## Polecenie 2

Napisz program wykorzystujący instrukcje *forever* i *breakif*

### Instrukcja *try..exception..endtry*

Instrukcja *try..exception..endtry* służy do implementacji bloków obsługi wyjątków. Składnia instrukcji:

```
try
    //instrukcje
exception(nazwa_wyjatku1)
    //instrukcje
exception(nazwa_wyjatku2)
    //instrukcje
endtry;
```

Dostępne wyjątki umieszczone są w pliku nagłówkowym *excepts.hhf* w katalogu HLA.

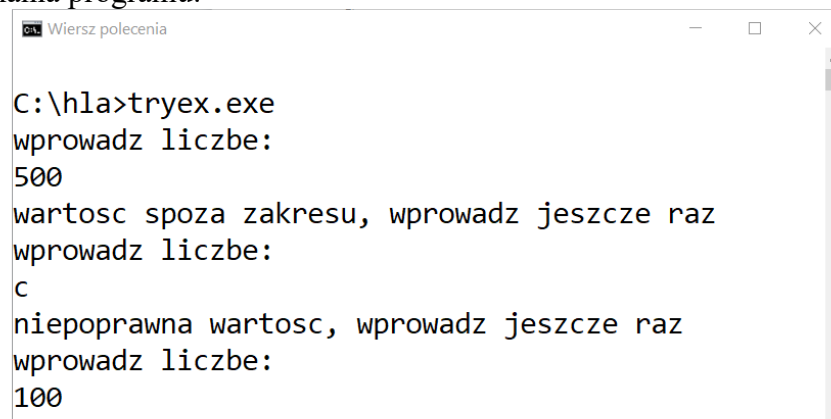
## Polecenie 3

Napisz program, w którym:

- Wykorzystane zostanie instrukcja *try..exception..endtry*
- Z klawiatury będą wprowadzane wartości aż do momentu wprowadzenia liczby całkowitej z zakresu <-128..127>
- W przypadku wprowadzenia liczby spoza zakresu powinien zostać wyświetlony odpowiedni komunikat
- W przypadku wprowadzenia innego niż liczba znaku powinien zostać wyświetlony odpowiedni komunikat
- Jako warunek użyj:

```
ex.ConversionError
ex.ValueOutOfRangeException
```

Przykład działania programu:



```
C:\hla>tryex.exe
wprowadz liczbe:
500
wartosc spoza zakresu, wprowadz jeszcze raz
wprowadz liczbe:
c
niepoprawna wartosc, wprowadz jeszcze raz
wprowadz liczbe:
100
```



**Fundusze Europejskie**  
Wiedza Edukacja Rozwój



**Rzeczpospolita  
Polska**

**Unia Europejska**  
Europejski Fundusz Społeczny



## **LABORATORIUM 5. DZIAŁANIA NA LICZBACH NIECAŁKOWITYCH.**

Cel laboratorium:

Celem zajęć jest zapoznanie z operacjami na liczbach zmiennoprzecinkowych oraz operacje na zmiennych znakowych w kodzie ASCII

Zakres tematyczny zajęć:

- liczby zmiennoprzecinkowe
- deklarowanie liczb zmiennoprzecinkowych
- wprowadzanie i wyprowadzanie wartości zmiennoprzecinkowych
- znaki kodu ASCII
- operacje na zmiennych znakowych

### **Zadanie 5.1 Wyprowadzanie liczb zmiennoprzecinkowych**

Liczby zmiennoprzecinkowe (ang. floating point numbers) stanowią reprezentację liczb rzeczywistych zapisanych w określonym formacie:

$$x = SMB^E$$

Gdzie:

$x$ - wartość liczby zmiennoprzecinkowej

$S$  (ang. sign) – znak liczby, 1 lub  $-1$ ,

$M$  (ang. mantissa) – znormalizowana mantysa, liczba ułamkowa

$B$  (ang. base) – podstawa systemu liczbowego

$E$  (ang. exponent) – wykładnik, cecha, liczba całkowita

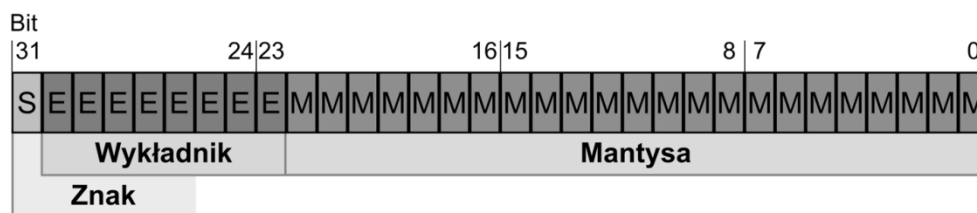
Przykład zapisu liczb zmiennoprzecinkowej:

$$x = -2.78 \cdot 10^{33}$$

Liczby zmiennoprzecinkowe umożliwiają zapis bardzo dużych liczb lub bardzo małych w notacji, która wymaga mniejszej ilości cyfr niż notacja normalna.

W implementacjach sprzętowych podstawą systemu liczbowego dla liczb zmiennoprzecinkowych jest cyfra 2. Poniżej przedstawiony został format zapisu liczby zmiennoprzecinkowej z wykorzystaniem 32 bitów. Wykładnik liczby zapisany jest w formacie znak-moduł. Najstarszy bit stanowi więc znak wykładnika. Na zapis wykładnika przeznaczono 8 bitów. Kolejne 23 bity przeznaczone są na zapis mantysy.





Rys 5.1 Zapis liczby zmiennoprzecinkowej o pojedynczej precyzji

Ponieważ na zapis poszczególnych składowych liczby określona jest ograniczona ilość bitów, oczywistym jest, że zapis liczby rzeczywistej w tej postaci jest jej przybliżeniem. W celu zwiększenia precyzji zapisu liczby zmiennoprzecinkowej można zwiększyć ilość bitów przeznaczonych do jej zapisu. W celu ujednolicenia zasad operacji na liczbach zmiennoprzecinkowych na różnych platformach sprzętowych, opracowano standard IEEE 754, w oparciu, o który realizuje się obecnie wszystkie implementacje sprzętowe liczb zmiennoprzecinkowych. Definiuje on dwie klasy liczb:

- pojedynczej precyzji
- podwójnej precyzji
- rozszerzonej precyzji

Liczby te różnią się ilością bitów przeznaczonych na zapis liczby zmiennoprzecinkowej według tabeli:

Liczba zmiennoprzecinkowa o precyzji	Znak	Wykładnik	Mantysa	Szerokość słowa
Pojedynczej	1	8	23	32
Podwójnej	1	11	52	64
Rozszerzonej	1	15	64	80

W HLA liczby zmiennoprzecinkowe reprezentowane są za pomocą literalów zmiennoprzecinkowych o podstawie liczbowej równej 10. Liczbę może poprzedzać nieobowiązkowy znak (plus lub minus), w przypadku braku znaku zakłada się, że liczba jest dodatnią, za ewentualnym znakiem występuje jedna bądź więcej cyfr dziesiętnych. Cyfry te uzupełnione są przecinkiem dziesiętnym (kropka) oraz jedną większą liczbą cyfr dziesiętnych. Całość może być uzupełniona literą „e” lub „E”, nieobowiązkowym znakiem oraz kolejnymi cyframi dziesiętnymi.

Deklarowanie liczb zmiennoprzecinkowych:

Do wykorzystania są następujące typy: real32, real64, real80. Podobnie jak w ich odpowiednikach całkowitych liczba kończąca nazwę typu określa rozmiar (w bitach) zmiennych tego typu.

real32 – odpowiada liczbom o pojedynczej precyzji

real64 – odpowiada liczbom o podwójnej precyzji

real80 – odpowiada liczbom o rozszerzonej precyzji

Deklarację zmiennej zmiennoprzecinkowej realizuje się w następujący sposób:



Fundusze Europejskie  
Wiedza Edukacja Rozwój



Rzeczpospolita  
Polska

Unia Europejska  
Europejski Fundusz Społeczny





```
static
```

```
    zmienna1: real32;  
    zmienna2: real64;  
    zmienna3: real80;
```

Wartość początkową zmiennej zmiennoprzecinkowej deklaruje się podając wartość liczby w postaci dziesiętnej bądź wykładniczej w następujących postaciach:

```
static
```

```
    zmienna1: real32:= -4.87e-2;  
    zmienna2: real32:= -0.0487;
```

### **Wyprowadzanie liczb zmiennoprzecinkowych**

Do wyprowadzenia liczby zmiennoprzecinkowej może posłużyć jedna z procedur:

- `stdout.putr32`
- `stdout.putr64`
- `stdout.putr80`

Procedura służy do wyświetlania liczb w postaci dziesiętnej.

Składnia wywołania jest identyczna dla wszystkich procedur:

```
stdout.putr32( liczba_zmiennoprzecinkowa, szerokość,  
liczba_cyfr_po_przecinku, znak_wypełnienia )
```

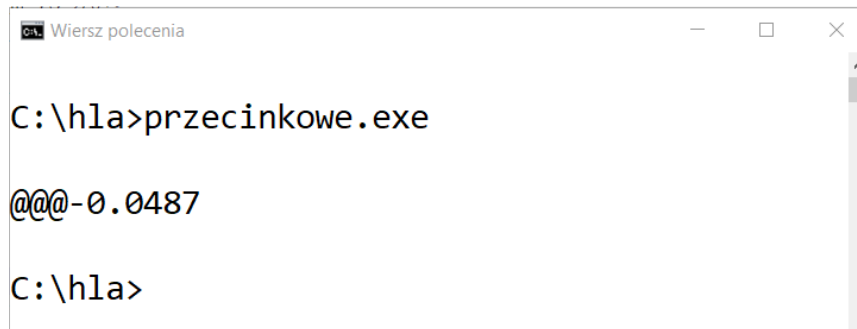
Pierwszym argumentem powinna być wartość zmiennoprzecinkowa, która ma zostać wyprowadzona na standardowe wyjście programu. Argument musi mieć odpowiedni rozmiar określany przez szerokość rozumianą, jako ilość miejsc przeznaczonych na wyświetlenie liczby. Jeżeli liczba miejsc przeznaczona na wyświetlenie liczby okaże się niewystarczająca, wówczas zostaną wyświetlone znaki #. Znak wypełnienia zostanie wyświetlony, jeżeli ilość miejsc przeznaczonych na wyświetlenie liczby jest większa niż wymagana ilość miejsc.

Przykład:

```
program przec;  
#include( "stdlib.hhf" );  
  
static  
  
    a:    real32:= -4.87e-2;  
  
begin przec;  
  
    stdout.putr32(a,10,4,'@');  
  
end przec;
```



Działanie programu:



```
Wiersz polecenia
C:\hla>przecinkowe.exe
@@@-0.0487
C:\hla>
```

Do wyprowadzenia wartości liczby zmiennoprzecinkowej w postaci wykładniczej służą procedury:

- stdout.pute32,
- stdout.pute64
- stdout.pute80,

Wywołanie procedury:

```
stdout.pute32( liczba-zmiennoprz , szerokość );
```

Szerokość określa wszystkie pozycje po przecinku (razem z pozycjami wykładnika)

Przykład:

```
program przec;
#include( "stdlib.hhf" );

static

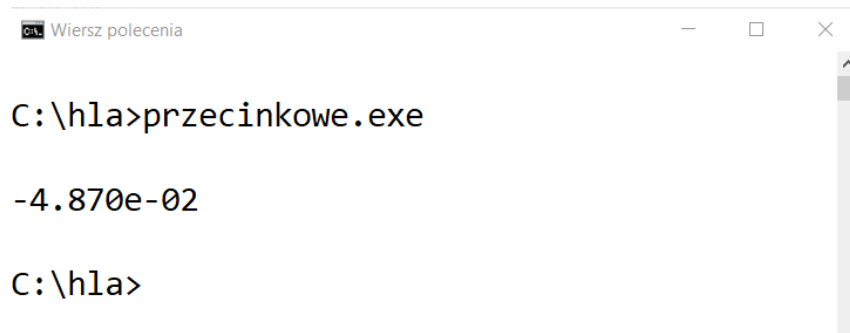
    a:   real32:= -4.87e-2;

begin przec;

    stdout.pute32(a,10);

end przec;
```

Działanie programu:



```
C:\hla>przecinkowe.exe

-4.870e-02

C:\hla>
```

Do wyświetlenia liczby zmiennoprzecinkowej można również wykorzystać procedurę `stdout.put`, jeśli w wywołaniu tej procedury znajdzie się nazwa obiektu zmiennoprzecinkowego jego wartość zostanie skonwertowana do postaci napisu zawierającego notację wykładniczą liczby. Szerokość ustalana jest automatycznie.

Przykład:

```
program przec;
#include( "stdlib.hhf" );

static

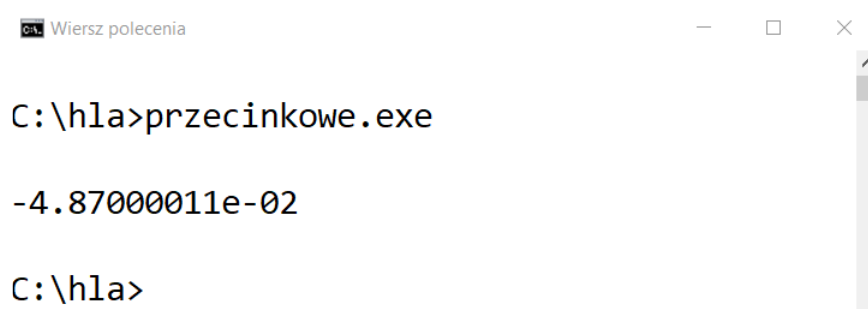
    a:  real32:= -4.87e-2;

begin przec;

    stdout.put(a);

end przec;
```

Działanie programu:



```
C:\hla>przecinkowe.exe

-4.87000011e-02

C:\hla>
```

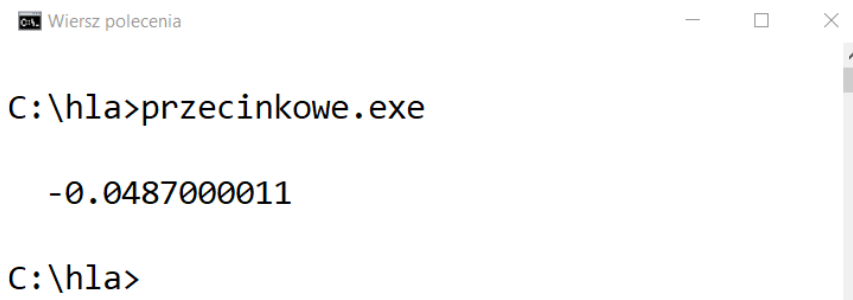
Jeśli procedura `stdout.put` ma konwertować liczby zmiennoprzecinkowe do zapisu dziesiętnego należy argument procedury `stdout.put` określać następująco:

```
stdout.put(nazwa:szerokość:liczba-cyfr_po_przecinku)
```

Przykład:

```
program przec;  
#include( "stdlib.hhf" );  
  
static  
  
    a:    real32:= -4.87e-2;  
  
begin przec;  
  
    stdout.put(a:15:10);  
  
end przec;
```

Działanie programu:



```
C:\hla>przecinkowe.exe  
  
-0.0487000011  
  
C:\hla>
```

### Polecenie 1

Napisz program w którym:

- zadeklarowane zostaną zmienne o wartościach  
 $-4.87660 \times 10^{-2}$   
 $562.3456 \times 10^1$   
 $3.14 \times 10^{-15}$

Zmienne te powinny zostać wyświetlone czterokrotnie:

- w postaci dziesiętnej za pomocą dwóch procedur
- w postaci wykładniczej - również dwie procedury

Przykład działania programu:

```
Wiersz polecenia

C:\hla>przecinkowe.exe
Zmienna 1:
Procedura stdout.putr32:      @@@-0.0488
Procedura stdout.pute32:      -4.877e-02
Procedura stdout.put postac dziesiętna:  -0.0487659983
Procedura stdout.put postac wykładnicza:  -4.87659983e-02

Zmienna 2:
Procedura stdout.putr32:      5623.4561
Procedura stdout.pute32:      5.623e+03
Procedura stdout.put postac dziesiętna:  5623.4560546875
Procedura stdout.put postac wykładnicza:  5.62345605e+03

Zmienna 3:
Procedura stdout.putr32:      @@@@@@@@@@@@ 0.00000000000000314
Procedura stdout.pute32:      3.140e-15
Procedura stdout.put postac dziesiętna:  0.00000000000000314
Procedura stdout.put postac wykładnicza:  3.14000005e-15
C:\hla>
```

## Zadanie 5.2 Zmienna znakowa cd.

Zmienna znakowa poznana na pierwszym laboratorium obsługuje standard kodowania znaków zgodny z kodem ASCII:

**ASCII** (ang. American Standard Code for Information Interchange) – 7-bitowy kod przyporządkowujący liczby z zakresu 0–127: literom alfabetu angielskiego, cyfrom, znakom przestankowym i innym symbolom oraz poleceniom sterującym.

Standard ten jest przyjęty przez znaczną część przemysłu. Jeżeli więc do reprezentowania znaku „A” wykorzystywana jest liczba 65, zgodnie z zestawem znaków ASCII, wtedy można mieć pewność że liczba ta zostanie zinterpretowana jako znak „A” również przez drukarkę czy terminal.

Zestaw znaków ASCII podzielony jest na 4 grupy po 32 znaki:

- Pierwsze 32 są to znaki niedrukowane (sterujące), np. znak wysuwu wiersza, znak cofania.
- Kolejne 32 to znaki specjalne, przystankowe i cyfry.
- Trzecia grupa to znaki wielkich liter alfabetu. Jako, że znaków w alfabecie łacińskim jest 26, sześć kodów przypisanych zostało do różnych znaków specjalnych.
- Czwartą grupę stanowią kody 26 małych liter alfabetu, pięć znaków specjalnych i znak sterujący DELETE.

Tabela kodu ASCII:

Dec	Hx	Oct	Char	Dec	Hx	Oct	Html	Chr	Dec	Hx	Oct	Html	Chr	Dec	Hx	Oct	Html	Chr
0	0	000	<b>NUL</b> (null)	32	20	040	&#32;	Space	64	40	100	&#64;	@	96	60	140	&#96;	`
1	1	001	<b>SOH</b> (start of heading)	33	21	041	&#33;	!	65	41	101	&#65;	A	97	61	141	&#97;	a
2	2	002	<b>STX</b> (start of text)	34	22	042	&#34;	"	66	42	102	&#66;	B	98	62	142	&#98;	b
3	3	003	<b>ETX</b> (end of text)	35	23	043	&#35;	#	67	43	103	&#67;	C	99	63	143	&#99;	c
4	4	004	<b>EOT</b> (end of transmission)	36	24	044	&#36;	\$	68	44	104	&#68;	D	100	64	144	&#100;	d
5	5	005	<b>ENQ</b> (enquiry)	37	25	045	&#37;	%	69	45	105	&#69;	E	101	65	145	&#101;	e
6	6	006	<b>ACK</b> (acknowledge)	38	26	046	&#38;	&	70	46	106	&#70;	F	102	66	146	&#102;	f
7	7	007	<b>BEL</b> (bell)	39	27	047	&#39;	'	71	47	107	&#71;	G	103	67	147	&#103;	g
8	8	010	<b>BS</b> (backspace)	40	28	050	&#40;	(	72	48	110	&#72;	H	104	68	150	&#104;	h
9	9	011	<b>TAB</b> (horizontal tab)	41	29	051	&#41;	)	73	49	111	&#73;	I	105	69	151	&#105;	i
10	A	012	<b>LF</b> (NL line feed, new line)	42	2A	052	&#42;	*	74	4A	112	&#74;	J	106	6A	152	&#106;	j
11	B	013	<b>VT</b> (vertical tab)	43	2B	053	&#43;	+	75	4B	113	&#75;	K	107	6B	153	&#107;	k
12	C	014	<b>FF</b> (NP form feed, new page)	44	2C	054	&#44;	,	76	4C	114	&#76;	L	108	6C	154	&#108;	l
13	D	015	<b>CR</b> (carriage return)	45	2D	055	&#45;	-	77	4D	115	&#77;	M	109	6D	155	&#109;	m
14	E	016	<b>SO</b> (shift out)	46	2E	056	&#46;	.	78	4E	116	&#78;	N	110	6E	156	&#110;	n
15	F	017	<b>SI</b> (shift in)	47	2F	057	&#47;	/	79	4F	117	&#79;	O	111	6F	157	&#111;	o
16	10	020	<b>DLE</b> (data link escape)	48	30	060	&#48;	0	80	50	120	&#80;	P	112	70	160	&#112;	p
17	11	021	<b>DC1</b> (device control 1)	49	31	061	&#49;	1	81	51	121	&#81;	Q	113	71	161	&#113;	q
18	12	022	<b>DC2</b> (device control 2)	50	32	062	&#50;	2	82	52	122	&#82;	R	114	72	162	&#114;	r
19	13	023	<b>DC3</b> (device control 3)	51	33	063	&#51;	3	83	53	123	&#83;	S	115	73	163	&#115;	s
20	14	024	<b>DC4</b> (device control 4)	52	34	064	&#52;	4	84	54	124	&#84;	T	116	74	164	&#116;	t
21	15	025	<b>NAK</b> (negative acknowledge)	53	35	065	&#53;	5	85	55	125	&#85;	U	117	75	165	&#117;	u
22	16	026	<b>SYN</b> (synchronous idle)	54	36	066	&#54;	6	86	56	126	&#86;	V	118	76	166	&#118;	v
23	17	027	<b>ETB</b> (end of trans. block)	55	37	067	&#55;	7	87	57	127	&#87;	W	119	77	167	&#119;	w
24	18	030	<b>CAN</b> (cancel)	56	38	070	&#56;	8	88	58	130	&#88;	X	120	78	170	&#120;	x
25	19	031	<b>EM</b> (end of medium)	57	39	071	&#57;	9	89	59	131	&#89;	Y	121	79	171	&#121;	y
26	1A	032	<b>SUB</b> (substitute)	58	3A	072	&#58;	:	90	5A	132	&#90;	Z	122	7A	172	&#122;	z
27	1B	033	<b>ESC</b> (escape)	59	3B	073	&#59;	;	91	5B	133	&#91;	[	123	7B	173	&#123;	{
28	1C	034	<b>FS</b> (file separator)	60	3C	074	&#60;	<	92	5C	134	&#92;	\	124	7C	174	&#124;	
29	1D	035	<b>GS</b> (group separator)	61	3D	075	&#61;	=	93	5D	135	&#93;	]	125	7D	175	&#125;	}
30	1E	036	<b>RS</b> (record separator)	62	3E	076	&#62;	>	94	5E	136	&#94;	^	126	7E	176	&#126;	~
31	1F	037	<b>US</b> (unit separator)	63	3F	077	&#63;	?	95	5F	137	&#95;	_	127	7F	177	&#127;	DEL

Źródło: LookupTables.com

Rys 5.2 Tabela kodu ASCII

Literały znakowe w języku HLA mogą przyjmować jedną z dwóch postaci: pojedynczego znaku otoczonego znakami pojedynczego cudzysłowu albo wartości (z zakresu 0 do 127) określającej kod ASCII poprzedzonej znakiem kratki (#):

‘A’ #65 #\$41 %#01000001

Wszystkie powyższe literały reprezentują ten sam znak „A”. Zapis #65 oznacza zapis znaku, który w kodzie ASCII znajduje się na 65 pozycji. Zapis #\$41 oznacza zapis liczby znaku nr 65 w postaci szesnastkowej, zapis %#01000001 oznacza zapis liczby znaku nr 65 w postaci binarnej.

Warto przyjąć zasadę, że znaki drukowalne określone są w kodzie programu literałami w pierwszej z prezentowanych form, czyli jako znaki ujęte w znaki pojedynczego cudzysłowu.

Znak kratki i literał liczbowy należy stosować wyłącznie do określania znaków niedrukowalnych: specjalnych i sterujących, ewentualnie do określenia znaków z rozszerzonej części zestawu ASCII, które w kodzie źródłowym mogą być wyświetlane niepoprawnie.



Fundusze Europejskie  
Wiedza Edukacja Rozwój



Rzeczpospolita  
Polska

Unia Europejska  
Europejski Fundusz Społeczny



Aby zadeklarować w HLA zmienną znakową należy skorzystać z typu danych o nazwie `char`:

```
static
    znakA:                char;
    znak_rozszerzony_DEL: char;
```

Zmienne znakowe można przy deklaracji inicjalizować:

```
static
    znakA:                char:='A'
    znak_rozszerzony:     char:=#127
```

Zmienne znakowe są obiektami 8-bitowymi, można nimi manipulować za pośrednictwem ośmiobitowych rejestrów i odwrotnie – zawartość takiego rejestru można skopiować do zmiennej znakowej.

### **Procedury wyprowadzania i wprowadzania zmiennej znakowej:**

Procedura wyprowadzenia zmiennej znakowej `stdout.putc`. Składnia:

```
stdout.putc( zmienna_znakowa/rejestr );
```

Działanie:

- procedura wyprowadza na standardowe wyjście programu pojedynczy znak określony wartością argumentu wywołania procedury
- argument może zostać określony jako stała lub zmienna znakowa bądź rejestr 8-bitowy

Procedura wyprowadzenia zmiennej znakowej `stdout.putcSize`. Procedura pozwala na wyprowadzenie znaków z określeniem szerokości wyprowadzanego napisu i wypełnienia.

Składnia:

```
stdout.putcSize( zmienna_znakowa, szerokosc, wypełnienie );
```

Procedura wyprowadza znak określony zmienną znakową, umieszczając go w napisie o określonej szerokości. Jeżeli bezwzględna wartość argumentu szerokość jest większa niż jeden, napis zostanie uzupełniony znakami wypełnienia. Jeśli argument szerokość zostanie określony jako ujemny, wyprowadzany znak będzie wyrównany do lewej krawędzi napisu. Wartość dodatnia powoduje wyrównanie do prawej krawędzi napisu.

Przykład:

```
program znaki;
#include( "stdlib.hhf" );

static
```

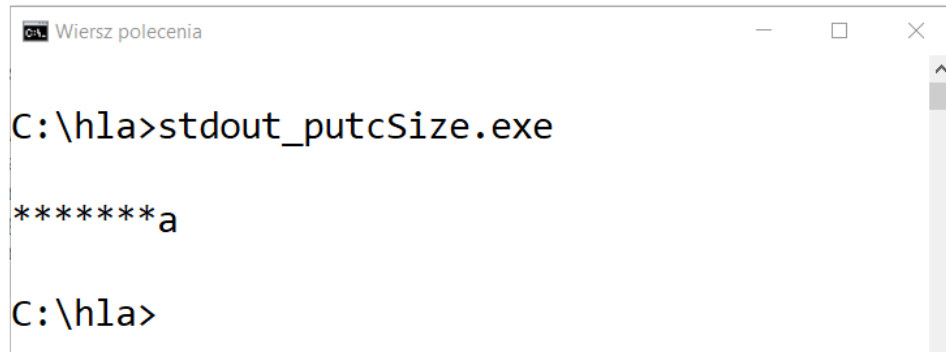
```
znak:      char:= 'a';
szerokosc: int32:=8;

begin znaki;

    stdout.putcSize(znak,szerokosc,'*');

end znaki;
```

Działanie programu:



```
C:\h1a>stdout_putcSize.exe

*****a

C:\h1a>
```

Wartości znakowe mogą być też wyprowadzane za pośrednictwem uniwersalnej procedury wyjścia *stdout.put*. Jeśli na liście wywołania tej procedury znajduje się zmienna znakowa, kod procedury automatycznie wyświetli odpowiadający jej znak, np.:

```
stdout.put( "Znak c = ",c," ' ",nl );
```

Pobieranie znaków za można zrealizować za pomocą procedur *stdin.getc* i *stdin.get*. Procedura *stdin.getc* nie przyjmuje żadnych argumentów. Jej działanie ogranicza się do wczytania z bufora urządzenia standardowego wejścia pojedynczego znaku i umieszczenia go w rejestrze AL. Po wczytaniu do rejestru można tą wartością manipulować na miejscu albo skopiować do zmiennej w pamięci.

### **Polecenie 1**

Napisz program, który wczytuje z klawiatury duże litery, zamienia je na małe i wyświetla na ekranie. W programie wykorzystać procedurę *stdin.getc()*



Przykładowe działanie programu:

```
Wiersz polecenia

C:\hla>duże_na_male_w_petli.exe
Wprowadz znaki: X
Wprowadzony znak po konwersji
do malej litery to: 'x'

C:\hla>duże_na_male_w_petli.exe
Wprowadz znaki: f
wprowadzono nieodpowiedni znak

C:\hla>
```

## Polecenie 2

Zmodyfikuj działanie programu z polecenia 1 tak by dokonywał zamiany znaków z wykorzystaniem operacji logicznej umieszczonej poniżej.

Podpowiedź:

Algebra Boole'a - operacje logiczne:

- negacja (zaprzeczenie logiczne - NOT)
- alternatywa (suma logiczna - OR)
- koniunkcja (iloczyn logiczny - AND)
- różnicy symetrycznej (suma modulo dwa – XOR)

### Negacja – zaprzeczenie logiczne

Jest to operacja jednoargumentowa. Wynikiem jest wartość logiczna przeciwna do tej, którą ma argument negacji.

a	NOT a
0	1
1	0

Składnia polecenia niskopoziomowego *not*:

`not (argument) ;`

Argumentem może być zmienna bądź rejestr.

### Alternatywa – suma logiczna

a	b	a OR b
0	0	0
0	1	1
1	0	1
1	1	1

Składnia polecenia niskopoziomowego *or*:

```
or(argument1, argument2);
```

Argumentem może być zmienna bądź rejestr. Wynik operacji zapisywany jest w lokacji drugiego argumentu.

Operację OR możemy zastosować, gdy w słowie binarnym chcemy ustawić n-ty bit na 1. W tym celu przygotowujemy maskę o długości słowa binarnego, w której wszystkie bity są wyzerowane z wyjątkiem bitu n-tego. Następnie wykonujemy operację alternatywy nad słowem binarnym i maską. W wyniku n-ty bit słowa zostanie ustawiony na 1. Na przykład:

```

110001010011 zmieniane słowo
OR 000000001000 maska bitowa
-----
110001011011 wynik operacji
    
```

### Koniunkcja - iloczyn logiczny

a	b	a AND b
0	0	0
0	1	0
1	0	0
1	1	1

Składnia polecenia niskopoziomowego *and*:

```
and(argument1, argument2);
```

Argumentem może być zmienna bądź rejestr. Wynik operacji zapisywany jest w lokacji drugiego argumentu.

Operację AND możemy zastosować, gdy potrzebne jest np. wyzerowanie bitu w słowie. W tym celu tworzymy maskę, w której wszystkie bity są ustawione na 1 z wyjątkiem bitu na pozycji do wyzerowania. Następnie wykonujemy operację koniunkcji nad słowem



Fundusze Europejskie  
Wiedza Edukacja Rozwój



Rzeczpospolita  
Polska

Unia Europejska  
Europejski Fundusz Społeczny



zawierającym bit oraz maską. W wyniku otrzymujemy słowo z wyzerowanym bitem na zadanej pozycji.

$$\begin{array}{r}
 110001011111 \text{ zmieniane słowo} \\
 \text{AND } 111111110111 \text{ maska bitowa} \\
 \hline
 110001010111 \text{ wynik operacji}
 \end{array}$$

## Różnica symetryczna - suma modulo dwa

a	b	a XOR b
0	0	0
0	1	1
1	0	1
1	1	0

Składnia polecenia niskopoziomowego *and*:

```
xor(argument1, argument2);
```

Argumentem może być zmienna bądź rejestr. Wynik operacji zapisywany jest w lokacji drugiego argumentu.

Operacja ta znajdzie zastosowanie, gdy zależy nam na zmianie stanu wybranego bitu na przeciwny. W tym celu przygotowujemy maskę z wyzerowanymi bitami za wyjątkiem pozycji do zamiany stanu. Nad słowem i maską wykonujemy operację różnicy symetrycznej.

$$\begin{array}{r} 110001010111 \text{ zmieniane słowo} \\ \text{XOR } 000000001000 \text{ maska bitowa} \\ \hline 110001010111 \text{ wynik operacji} \end{array}$$

### Zadanie 5.3 Moduł Stdio

Biblioteka standardowa HLA posiada predefiniowane stałe znakowe wymienione poniżej:

- `stdio.bell` – znak dzwonka (głośniczek systemowy)
- `stdio.bs` – znak cofania kursora
- `stdio.tab` – znak tabulacji
- `stdio.lf` – znak wysuwu wiersza
- `stdio.cr` – znak powrotu karetki

występowanie – jak „nl” w procedurze stdout.put

Aby wymusić każdorazowe wczytywanie nowego wiersza danych przy pobieraniu kolejnych znaków, należy wywołanie procedury wczytującej znak poprzedzić wywołaniem procedury *stdin.flushInput()*.

Wymusi to wprowadzenie nowego wiersza danych w ramach realizacji wywołania *stdin.getc* albo *stdin.get*

Biblioteka standardowa języka HLA zawiera definicję "końca wiersza". Procedura *stdin.eoln()* zwraca w rejestrze AL wartość jeden, jeśli bieżący wiersz znaków wejściowych został wyczerpany (w innym przypadku rejestr AL zawiera po wywołaniu wartość zero).

### **Polecenie 1**

Przeanalizuj kod programu wczytującego z klawiatury ciąg znaków i następnie wyprowadzającego każdy ze znaków w kolumnie pionowej jednocześnie dekodując wartości poszczególnych znaków na postać liczbową szesnastkową:

```
program eoln;
#include( "stdlib.hhf" );

begin eoln;

    stdout.put( "Wprowadz krotki wiersz tekstu: " );
    stdin.flushInput();
    repeat


        stdin.getc();
        stdout.putc( al );
        stdout.put( "=", al, nl );

    until( stdin.eoln() );

end eoln;
```



Działanie programu:

 Wiersz polecenia

```
C:\hla>demo_eoln.exe
Wprowadz krotki wiersz tekstu: POLITECHNIKA LUBELSKA
P=50
O=4F
L=4C
I=49
T=54
E=45
C=43
H=48
N=4E
I=49
K=4B
A=41
=20
L=4C
U=55
B=42
E=45
L=4C
S=53
K=4B
A=41

C:\hla>
```

## **Polecenie 2**

Napisz program wykorzystujący wybrane stałe modułu *stdio* oraz procedurę *stdin.flushInput()*

## **LABORATORIUM 6. DEKLARACJE I ODWOŁANIA DO TABLIC JEDNOWYMIAROWYCH, WYKORZYSTANIE TRYBÓW ADRESOWANIA**

Cel laboratorium:

Celem zajęć jest zapoznanie z trybami adresowania oraz ze zmienną tablicową

Zakres tematyczny zajęć:

- tryby adresowania
- deklaracja zmiennej tablicowej
- wprowadzanie i wyprowadzanie elementów z wykorzystaniem zmiennej tablicowej
- koercja typów

### **Zadanie 6.1 Tryby adresowania**

Tryby adresowania procesorów 80x86 obejmują:

- adresowanie bezpośrednie.
- adresowanie bazowe,
- adresowanie bazowe indeksowane,
- adresowanie indeksowe,
- adresowanie bazowe indeksowane z przemieszczeniem

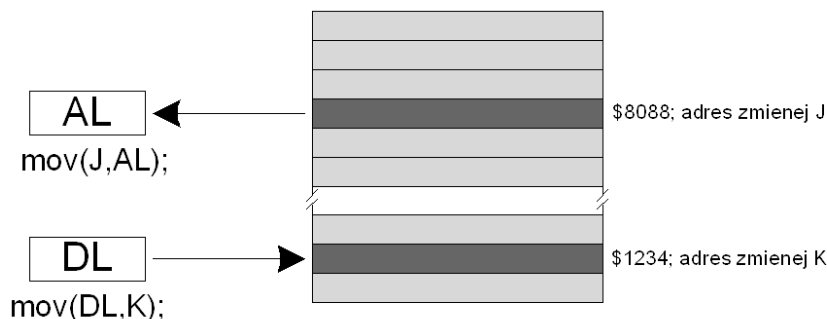
Pozostałe tryby adresowania to odmiany owych trybów podstawowych.

Dobór odpowiedniego trybu adresowania to klucz do efektywnego programowania w assemblerze.

### **Adresowanie bezpośrednie**

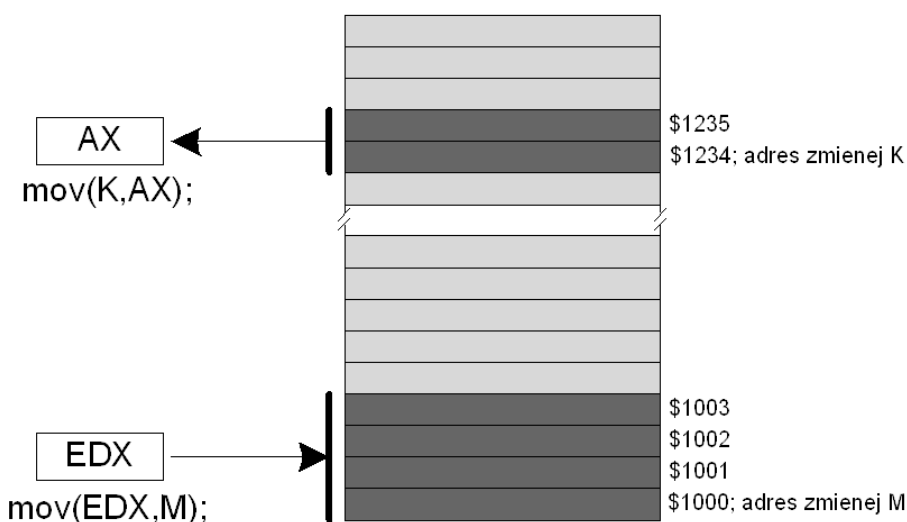
Adresowanie bezpośrednie jest najczęściej wykorzystywane. Adres docelowy określany jest 32-bitową stałą. Tryb adresowania bezpośredniego doskonale nadaje się do realizacji odwołań do prostych zmiennych skalarnych. Firma Intel przyjęła dla tego trybu nazwę adresowania z przemieszczeniem, ponieważ bezpośrednio po kodzie instrukcji *mov* w pamięci zapisana jest 32-bitowa stała przemieszczana. Przesunięcie w procesorach 80x86 definiowane jest jako przesunięcie (ang. offset) od początkowego adresu pamięci (czyli adresu zerowego).

Adresowanie bezpośrednie zmiennej jednobajtowej:



Rys 6.1 Schemat adresowania bezpośredniego dla zmiennej o wielkości 1 bajta

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



Rys 6.2 Schemat adresowania bezpośredniego dla zmiennej o wielkości 16 i 32bitów

Adresowanie wykorzystywane na laboratorium do chwili obecnej było adresowaniem bezpośrednim.

### Adresowanie pośrednie przez rejestr

Adresowanie pośrednie oznacza, że operand nie jest właściwym adresem, dopiero wartość operandu określa adres odwołania. Wartość rejestru to docelowy adres pamięci. Tryb adresowania pośredniego przez rejestr jest sygnalizowany nawiasami prostokątnymi. Instrukcja:

```
mov ( EAX, [EBX] );
```



Fundusze Europejskie  
Wiedza Edukacja Rozwój



Rzeczpospolita  
Polska

Unia Europejska  
Europejski Fundusz Społeczny



informuje procesor, aby ten zachował zawartość rejestru EAX w miejscu, którego adres znajduje się w rejestrze EBX.

Procesowy 80x86 obsługuje 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 wymaga stosowania rejestrów 32-bitowych. Umożliwia ono odwoływanie się do danych, dysponując jedynie wskaźnikami na nie.

HLA udostępnia prosty operator pozwalający na załadowanie 32-bitowego rejestru adresem zmiennej. Ograniczeniem jest rodzaj zmiennej, musi być ona zmienną statyczną.

```
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 &.

### **Polecenie 1**

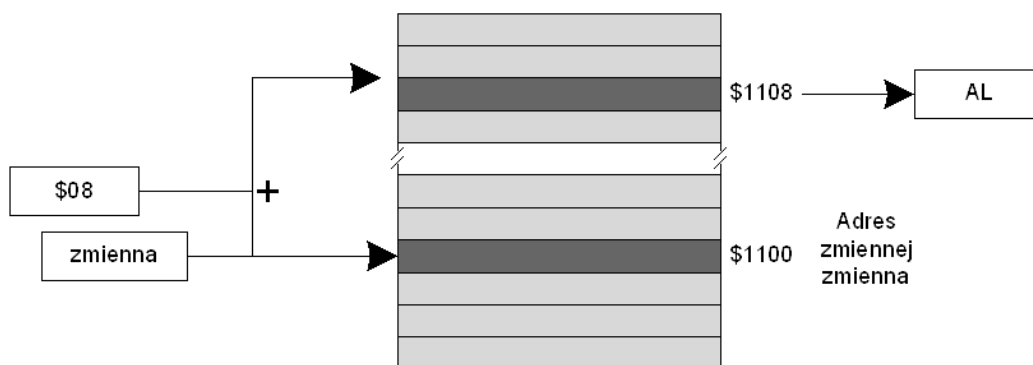
Napisz program, w którym zostanie pobrany i wyświetlony adres zmiennej statycznej oraz zostanie do niej zapisana wartość z wykorzystaniem adresowania pośredniego przez rejestr.

### **Adresowanie indeksowe**

W tym trybie adresowania 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:



`mov( zmienna[EBX], AL );`



Rys 6.3 Schemat adresowania indeksowanego

Tryb adresowania indeksowego jest szczególnie poręczny do odwoływania się do elementów tablic.

### Adresowanie indeksowe skalowane

Ten tryb adresowania pozwala na wymnożenie rejestru indeksowego przez współczynnik (skalę) o wartości 1, 2, 4 bądź 8.

Składnię tego trybu określa się następująco:

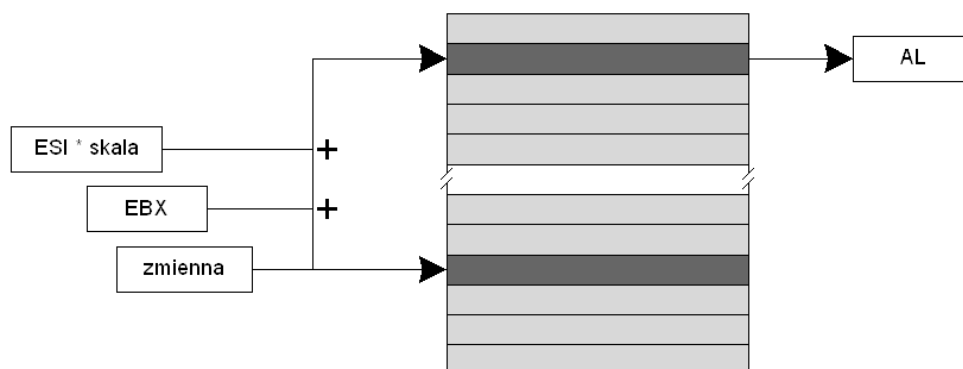
- `zmienna[rejestr-indeksowy32 * skala]`
- `zmienna[rejestr-indeksowy32 * skala + przesunięcie]`
- `zmienna[rejestr-indeksowy32 * skala - przesunięcie]`
- `[rejestr-bazowy32 + rejestr-indeksowy32 * skala]`
- `[rejestr-bazowy32 + rejestr-indeksowy32 * skala + przesunięcie]`
- `[rejestr-bazowy32 + rejestr-indeksowy32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

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

Rejestr-bazowy32 może być reprezentowany przez dowolny z 32-bitowych rejestrów ogólnego przeznaczenia podobnie jak rejestr-indeksowy32 (z puli dostępnych dla tego operandu rejestrów należy jednak wykluczyć rejestr ESP)

W trybie tym adres efektywny obliczany jest przez dodanie wartości rejestru indeksowego pomnożonej przez współczynnik skalowania. Dopiero ta wartość wykorzystywana jest w roli indeksu.

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



Rys 6.4 Schemat adresowania indeksowanego skalowanego

Przykład:

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*4], AL );
```

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

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

### Deklaracja zmiennej tablicowej

Tablica jest agregatem, którego wszystkie elementy mają ten sam typ. Wyboru elementu tablicy dokonuje się określając indeks elementu będący liczbą całkowitą. Stosowany jest operator indeksowania, którego użycie powoduje wybranie z tablicy elementu o zadanym indeksie, np. odwołanie w postaci A[i] powoduje wybranie z tablicy A i-tego elementu. Tablice domyślnie indeksowane są od zera.

Dla zmiennej tablicowej określa się adres bazowy. Jest to adres, pod którym znajduje się w pamięci pierwszy element tablicy. Jest to zawsze najmniejszy co do wartości adres w tablicy. Drugi z elementów tablicy sąsiaduje w pamięci z elementem pierwszym, trzeci z drugim i tak dalej.



Rys 6.5 Schemat zapisu w pamięci zmiennej tablicowej

Deklaracja zmiennej tablicowej różni się od deklaracji zwykłej zmiennej tym, że po podaniu typu zmiennych w tablicy w nawiasie kwadratowym określa się ilość elementów tablicy. Deklarując tablicę należy zarezerwować dla jej elementów pewien obszar pamięci. Aby

przydzielić do tablicy pamięć dla  $n$  elementów, należy w jednej z sekcji deklaracji zmiennych umieścić następującą deklarację:

```
NazwaTablicy: TypElementu[n];
```

Przyrostek  $[n]$  instruuje kompilator, że przydzielona do tablicy pamięć powinna pomieścić  $n$  obiektów typu `TypElementu`.

Na przykład:

```
static
```

```
tab8:      int8[10];  
tab32:     int32[10];
```

W sekcji statycznej zostały zadeklarowane zmienne tablicowe o nazwie `tab8` i `tab32` o rozmiarze 10 elementów typu `int8` oraz `int32`.

Deklaracja wartości początkowych tablicy może zostać przeprowadzona następująco:

```
static
```

```
tab8:      int8[10]:=[0,1,2,3,4,5,6,7,8,9];  
tab32:     int32[10]:=[0,1,2,3,4,5,6,7,8,9];
```

W przypadku inicjalizacji tablicy tą samą wartością, przy dużej liczbie elementów tablicy, proces inicjalizacji można skrócić:

```
tab: int32[1000] := 1000 dup [1];
```

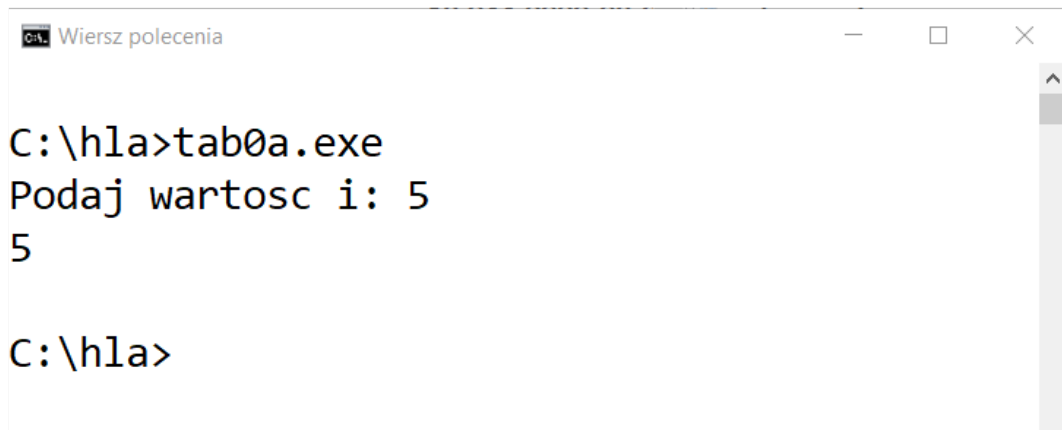
Za pośrednictwem operatora `dup` można również inicjalizować tablice seriami wartości:

```
tab: int32[16] := 4 dup [1, 2, 3, 4];
```

## Polecenie 2

Napisz program, w którym zadeklarowana zostanie tablica jednowymiarowa o określonej liczbie elementów typu `int8`. Wykorzystując adresowanie indeksowe program powinien wyświetlić zadany element tablicy.

Przykładowe działanie programu z zadeklarowaną tablicą o elementach od 1 do 10:

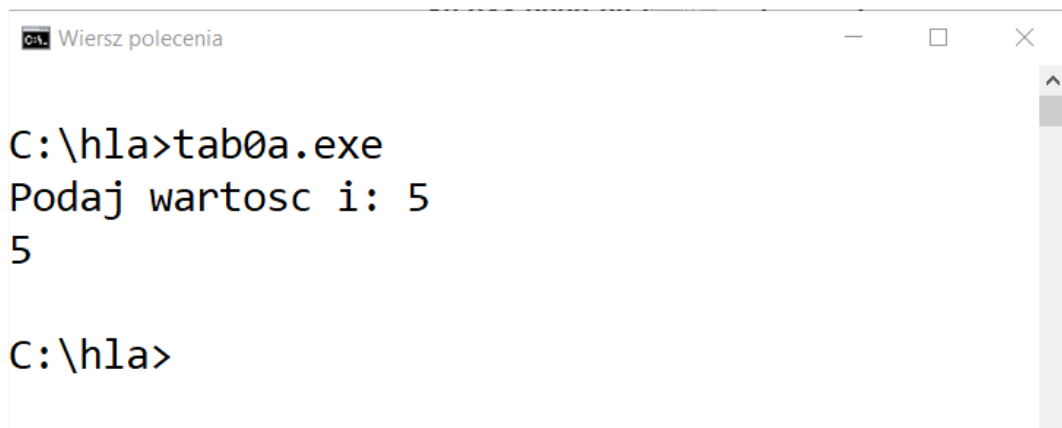


```
C:\hla>tab0a.exe
Podaj wartosc i: 5
5
C:\hla>
```

### **Polecenie 3**

Zmodyfikuj program z polecenia 1 tak by zadeklarowana tablica zawierała elementy o szerokości podwójnego słowa.

Przykładowe działanie programu:



```
C:\hla>tab0a.exe
Podaj wartosc i: 5
5
C:\hla>
```

### **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 int16 zmienna8bitowa), ax );
```

Wykorzystując koercję typów należy zwrócić uwagę na wielkość rzutowanych elementów. Koercja szerszego elementu do mniejszego może spowodować błędy.

#### **Polecenie 4**

Napisz program, w którym zostaną zadeklarowane dwie tablice jednowymiarowe o elementach typu int8 oraz int32. Wyświetlić obie tablice. Następnie program powinien skopiować zadany przez użytkownika element tablicy z elementami int8 do tablicy z danymi typu int32, oraz wyświetlić zawartość tej tablicy.

Przykładowe działanie programu;

```
Wiersz polecenia
C:\hla>tab0.exe
Tablica elementow int8:
0  1  2  3  4  5  6  7  8  9
Tablica elementow int32:
1  1  1  1  1  1  1  1  1  1
ktory element kopiowac: 4

Tablica elementow int32 po kopiowaniu elementu z tablicy int8:
1  1  1  1  4  1  1  1  1  1
C:\hla>
```

## **LABORATORIUM 7. DEKLARACJE I ODWOŁANIA DO TABLIC WIELOWYMIAROWYCH, WYKORZYSTANIE TRYBÓW ADRESOWANIA**

Cel laboratorium:

Celem zajęć jest zapoznanie z operacjami z wykorzystaniem tablic wielowymiarowych

Zakres tematyczny zajęć:

- adresowanie elementów tablic wielowymiarowych
- operacje arytmetyczne

### **Zadanie 7.1 Operacje arytmetyczne – ciąg dalszy**

#### **Mnożenie**

Poznane do tej pory instrukcje dodawania i odejmowania są instrukcjami dwuargumentowymi. Instrukcja mnożenia jest wyjątkiem i dla rodziny procesorów x86 jest instrukcją jednoargumentową.

Oczywistym jest fakt, operacja mnożenia wymaga użycia dwóch argumentów. Przyjęto rozwiązanie, które polega na tym, że argumentem instrukcji mnożenia może być rejestr lub zmienna w pamięci. Drugi z operandów powinien być umieszczony w akumulatorze i stanowi jednocześnie rolę operandu docelowego, tzn. w nim zapisywany jest wynik.

Istnieją dwa polenienia niskopoziomowe realizujące mnożenie liczb, służące do mnożenia liczb ze znakiem i bez znaku:

Mnożenie liczb bez znaku:

```
mul(rejestr8); //operand docelowy - ax
mul(rejestr16); //operand docelowy - dx:ax
mul(rejestr32); //operand docelowy - eax:edx
```

```
mul(zmienna8); //operand docelowy -ax
mul(zmienna16); //operand docelowy - dx:ax
mul(zmienna32); //operand docelowy - eax:edx
```

Mnożenie liczb ze znakiem:

```
imul(rejestr8); //operand docelowy -ax
imul(rejestr16); //operand docelowy - dx:ax
imul(rejestr32); //operand docelowy - eax:edx
```

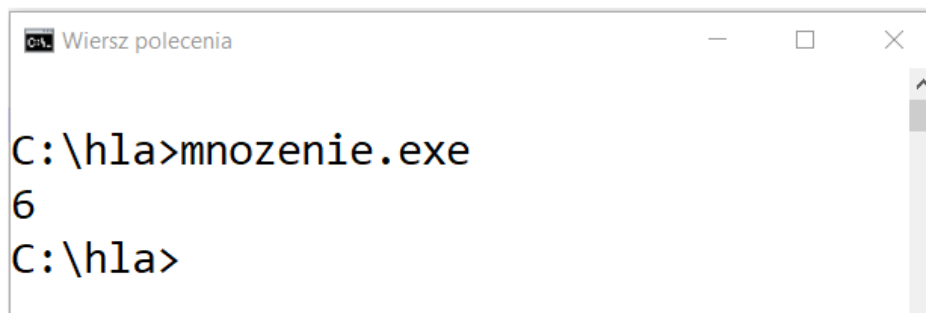
```
imul(zmienna8); //operand docelowy - ax
imul(zmienna16); //operand docelowy - dx:ax
imul(zmienna32); //operand docelowy - eax:edx
```



Wykonanie kodu:

```
mov(3,bl);  
mov(2,ax);  
mul(bl);  
stdout.puti16(ax);
```

spowoduje wykonanie operacji mnożenia wartości z rejestru *bl* i wartości z rejestru *ax* oraz zapisanie wyniku w rejestrze *ax*:

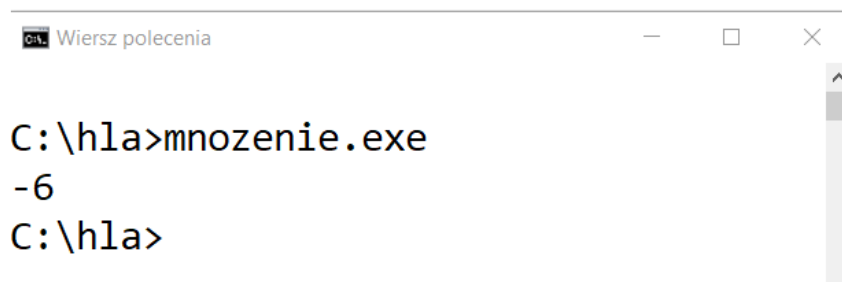


```
C:\hla>mnozenie.exe  
6  
C:\hla>
```

Do mnożenia liczb ze znakiem należy użyć polecenia *imul*:

```
mov(-2,ax);  
mov(3,zmienna8);  
imul(zmienna8);  
stdout.puti16(ax);
```

Działanie programu:



```
C:\hla>mnozenie.exe  
-6  
C:\hla>
```

Zapis *dx:ax* oznacza, że wynik zostanie zapisany w rejestrach *dx* i *ax*.

Dla uproszczenia zapisu operacji mnożenia a HLA dostępna jest dwuoperatorowa instrukcja mnożenia całkowitego *intmul*

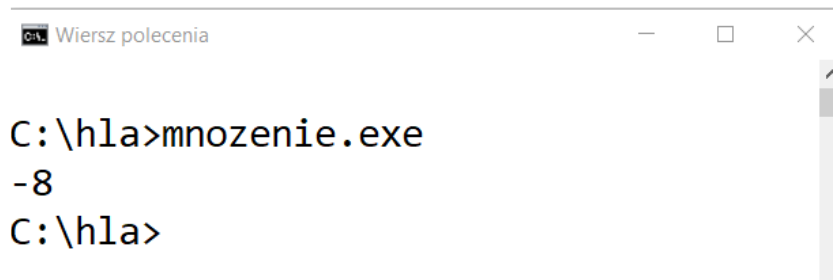
```
Intmul(operand1, operand2);
```

Operandem źródłowym może być stała, zmienna lub rejestr, natomiast operandem docelowym rejestr. Przykład:



```
mov(2, eax);  
intmul(4, eax);  
stdout.puti32(eax);
```

Działanie:



```
C:\hla>mnozenie.exe  
-8  
C:\hla>
```

### **Polecenie 1**

Napisz program, w którym zastosowane zostaną powyższe operacje mnożenia.

### **Dzielenie**

Instrukcje maszynowe procesorów x86 przewidują operacje dzielenia operandów 64-bitowego przez operand 32-bitowy, operandu 32-bitowego przez operand 16-bitowy oraz operandu 16-bitowego przez operand 8-bitowy.

```
div(rejestr8); //operand docelowy - al:ah  
div(rejestr16); //operand docelowy - ax:dx  
div(rejestr32); //operand docelowy - eax:edx  
  
div(zmienna8); //operand docelowy - al:ah  
div(zmienna16); //operand docelowy - ax:dx  
div(zmienna32); //operand docelowy - eax:edx
```

Instrukcja *div* oblicza całkowity iloraz bez znaku. Wykonanie kodu:

```
mov(7, ax);  
mov(2, bl);  
div(bl);  
stdout.put("iloraz calkowity:    ");  
stdout.puti8(al);  
stdout.newln();  
stdout.put("reszta calkowita:    ");  
stdout.puti8(ah);
```



spowoduje podzielenie wartości z rejestru *ax* przez wartość z rejestru *bx*. Iloraz całkowity umieszczany jest w rejestrze *ax*, a całkowita reszta z dzielenia w rejestrze *ah*. W przypadku, gdy operand jest wartością 16-bitową, wówczas iloraz całkowity umieszczany jest w rejestrze *ax*, a reszta całkowita w rejestrze *dx*. W przypadku dzielenia przez operand 32-bitowy, iloraz całkowity umieszczany jest w rejestrze *eax*, a reszta całkowita w rejestrze *edx*.

Działanie programu:

```
Wiersz polecenia

C:\hla>dzielenie.exe
iloraz calkowity:    3
reszta calkowita:   1
C:\hla>
```

HLA tak jak w przypadku operacji mnożenia udostępnia instrukcję, która umożliwia operację dzielenia z wykorzystaniem instrukcji dwuargumentowej:

```
div(rejestr8,ax); //operand docelowy - al:ah
div(rejestr16,dx:ax); //operand docelowy - ax:dx
div(rejestr32,edx:eax); //operand docelowy - eax:edx
```

```
div(zmienna8,ax); //operand docelowy - al:ah
div(zmienna16,dx:ax); //operand docelowy - ax:dx
div(zmienna32,edx:eax); //operand docelowy - eax:edx
```

Instrukcja ta umożliwia również dzielenie przez stałą użytą jako pierwszy operand:

```
div(12,ax);
```

Przykład:

```
mov(7,dx:ax);
mov(2,bx);
div(bx,dx:ax);
stdout.put("iloraz calkowity:    ");
stdout.putil6(ax);
stdout.newln();
stdout.put("reszta calkowita:    ");
stdout.putil6(dx);
```

Działanie programu:

```
Wiersz polecenia

C:\hla>dzielenie.exe
iloraz calkowity:    3
reszta calkowita:   1
C:\hla>
```

Dzielenie liczb ze znakiem przeprowadza się za pomocą instrukcji:

```
idiv(rejestr8); //operand docelowy - al:ah
idiv(rejestr16); //operand docelowy - ax:dx
idiv(rejestr32); //operand docelowy - eax:edx

idiv(zmienna8); //operand docelowy - al:ah
idiv(zmienna16); //operand docelowy - ax:dx
idiv(zmienna32); //operand docelowy - eax:edx

idiv(rejestr8,ax); //operand docelowy - al:ah
idiv(rejestr16,dx:ax); //operand docelowy - ax:dx
idiv(rejestr32,edx:eax); //operand docelowy - eax:edx

idiv(zmienna8,ax); //operand docelowy - al:ah
idiv(zmienna16,dx:ax); //operand docelowy - ax:dx
idiv(zmienna32,edx:eax); //operand docelowy - eax:edx
```

Działanie polecenia *idiv* jest analogiczne do działania polecenia *div*.

## Polecenie 2

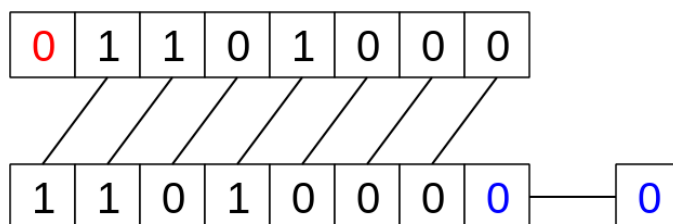
Napisz program, w którym zastosowane zostaną powyższe operacje dzielenia.

## Przesunięcie bitowe

Przesunięcie bitowe jest operacją na liczbach w systemie dwójkowym polegającą na przesunięciu wszystkich cyfr binarnych o określoną wartość w lewo lub prawo.

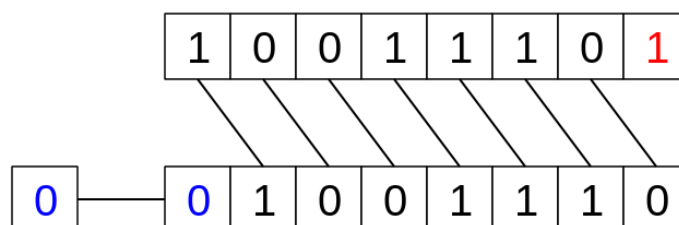
Rodzaje przesunięcia bitowego:

**Przesunięcie w lewo** polega na przesunięciu bitów w lewą stronę, dopisaniu na najmłodsze pozycje wartości zero, natomiast najstarsze bity są tracone. Na przykład:



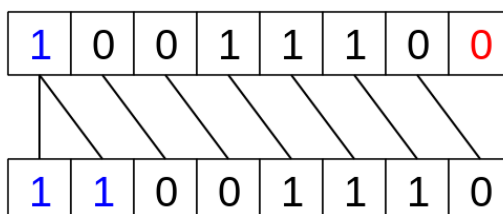
Rys 7.1 Schemat przesunięcia bitowego w lewo

**Przesunięcie w prawo** polega na przesunięciu bitów w prawą stronę, dopisaniu na najstarsze pozycje wartości zero, natomiast najmłodsze bity są tracone. Na przykład:



Rys 7.2 Schemat przesunięcia bitowego w prawo

**Przesunięcie arytmetyczne w prawo** polega na przesunięciu bitów w prawą stronę, powieleniu bitów na najstarszych pozycjach, natomiast najmłodsze bity są tracone. Na przykład:



Rys 7.3 Schemat przesunięcia bitowego arytmetycznego w prawo

Przesunięcie bitowe w prawo wykorzystywane jest w przypadku operacji na liczbach ujemnych zapisanych w kodzie uzupełnienia do dwóch.

Operację mnożenia przez 2, 4, 8... można wykonać za pomocą operacji przesunięcia bitowego w lewo. Dostępne jest polecenie niskopoziomowe *shl*:

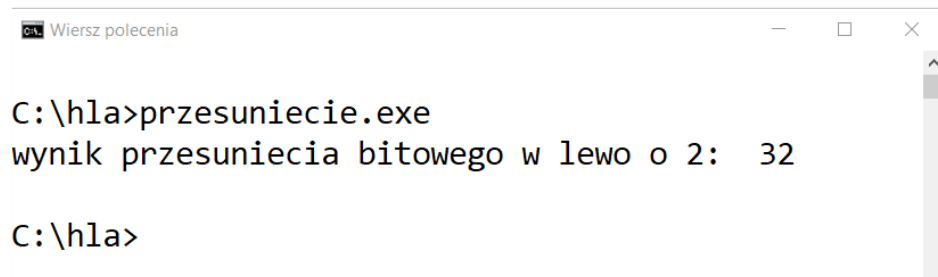
```
shl (operand1, operand2) ;
```

Operandem 1 może być stała, zmienna lub rejestr. Rolę operandu docelowego może pełnić zmienna w pamięci bądź rejestr.

Przykład:

```
mov(8, zmienna32);  
shl(2, zmienna32);  
stdout.put("wynik przesunieciecia bitowego w lewo o 2:  ");  
stdout.put(zmienna32);
```

Przesunięcie bitowe w lewo o dwie pozycje liczby 8:



```
C:\hla>przesuniecie.exe  
wynik przesunieciecia bitowego w lewo o 2: 32  
  
C:\hla>
```

Operację dzielenia przez 2,4,8... można wykonać za pomocą operacji przesunięcia bitowego w prawo. Dostępne jest polecenie niskopoziomowe *shr*:

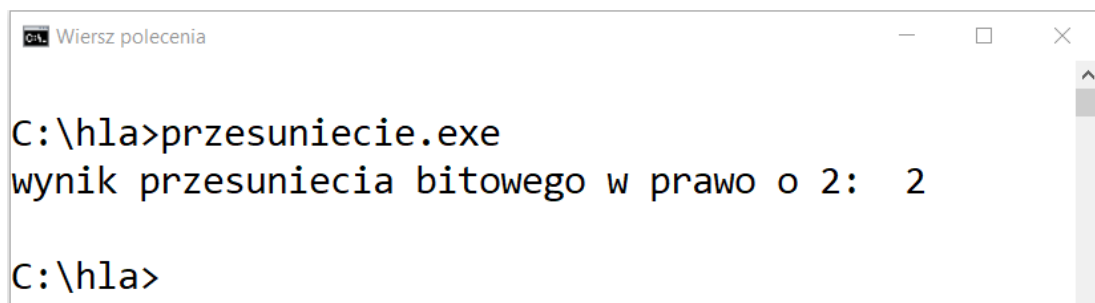
```
shr(operand1, operand2);
```

Operandem 1 może być stała, zmienna lub rejestr. Rolę operandu docelowego może pełnić zmienna w pamięci bądź rejestr. Działanie należy przeprowadzić na liczbie dodatniej.

Przykład programu dla dzielenia liczby 8 przez 4:

```
mov(8, eax);  
shr(2, eax);  
stdout.put("wynik przesunieciecia bitowego w prawo o 2:  ");  
stdout.puti32(eax);
```

Działanie:



```
C:\hla>przesuniecie.exe  
wynik przesunieciecia bitowego w prawo o 2: 2  
  
C:\hla>
```

Operację dzielenia liczby ujemnej można wykonać za pomocą operacji przesunięcia bitowego w prawo. Dostępne jest polecenie niskopoziomowe *sar*:

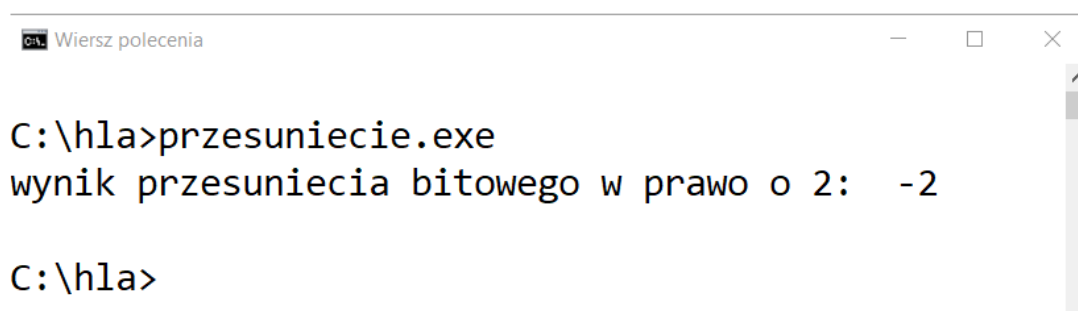
```
sar (operand1, operand2) ;
```

Operandem 1 może być stała, zmienna lub rejestr. Rolę operandu docelowego może pełnić zmienna w pamięci bądź rejestr.

Przykład programu dla dzielenia liczby -8 przez 4:

```
mov(-8, zmienna32) ;  
sar(2, zmienna32) ;  
stdout.put("wynik przesunieciecia bitowego w prawo o 2: ");  
stdout.put(zmienna32) ;
```

Działanie programu:



```
C:\hla>przesuniecie.exe  
wynik przesunieciecia bitowego w prawo o 2: -2  
  
C:\hla>
```

### **Polecenie 3**

Napisz program, w którym zastosowane zostaną powyższe operacje przesunięcia bitowego.

## **Zadanie 7.2 Adresowanie elementów tablic wielowymiarowych**

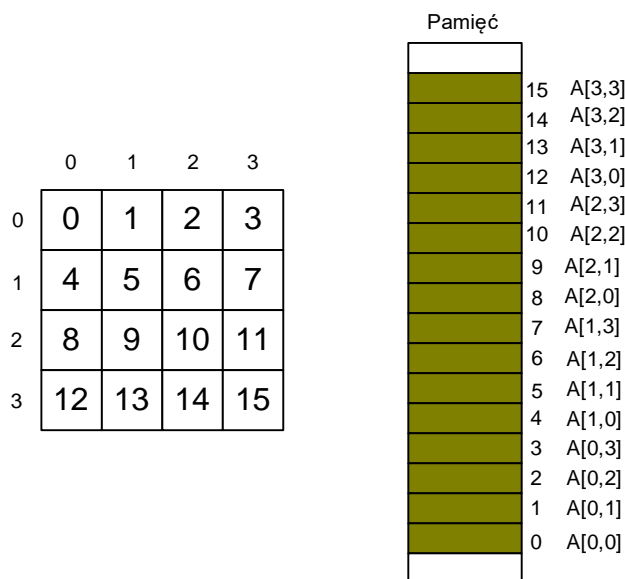
Dla rodziny procesorów x86 nie został przewidziany specjalny tryb adresowania tablic wielowymiarowych. Adresowanie elementów takich tablic należy konstruować samodzielnie.

### **Implementacja tablicy wielowymiarowej**

Elementy tablicy wielowymiarowej umieszczane są w pamięci, która jest tablicą jednowymiarową. Istnieją dwa sposoby układania elementów tablicy w pamięci:

- wierszowe
- kolumnowe

Dla układu wierszowego pod kolejnymi adresami pamięci umieszczane są elementy tablicy jakie występują w kolejnych jej wierszach:



Rys 7.4 Schemat zapisu w pamięci tablicy dwuwymiarowej o wierszowym układzie elementów

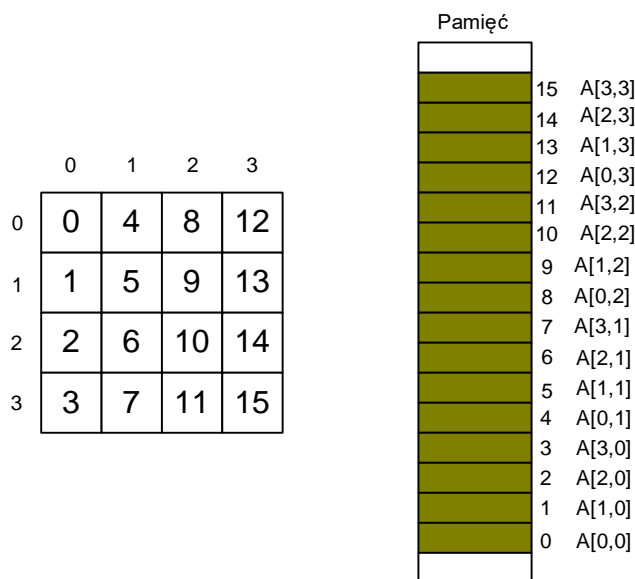
Adres bazowy tablicy to adres jej zerowego elementu  $A[0,0]$ . Odwołanie do elementu tablicy polega na określeniu przesunięcia adresu elementu tablicy względem adresu bazowego. Zależy ono od położenia elementu w tablicy oraz rozmiaru elementów.

Określenie adresu elementu dla tablicy dwuwymiarowej w układzie wierszowym:

Adres elementu = adres bazowy + (index\_kolumny \* rozmiar\_wiersza + index\_wiersza) \* rozmiar\_elementu

Adres bazowy określany jest poprzez podanie nazwy zmiennej.

Dla układu kolumnowego pod kolejnymi adresami pamięci umieszczane są elementy tablicy jakie występują w kolejnych jej wierszach:



Rys 7.5 Schemat zapisu w pamięci tablicy dwuwymiarowej o kolumnowym układzie elementów



**Fundusze Europejskie**  
Wiedza Edukacja Rozwój



**Rzeczpospolita Polska**

**Unia Europejska**  
Europejski Fundusz Społeczny



Określenie adresu elementu dla tablicy dwuwymiarowej w układzie wierszowym:

```
Adres elementu = adres_bazowy + (index_wiersza *  
rozmiar_kolumny + index_kolumny) * rozmiar_elementu
```

### **Deklarowanie tablicy dwuwymiarowej:**

```
static  
    tab: int32[rozmiar_kolumn,rozmiar_wierszy]:=[  
        0,1,2,3,  
        4,5,6,7,  
        8,9,10,11,  
        12,13,14,15  
    ];
```

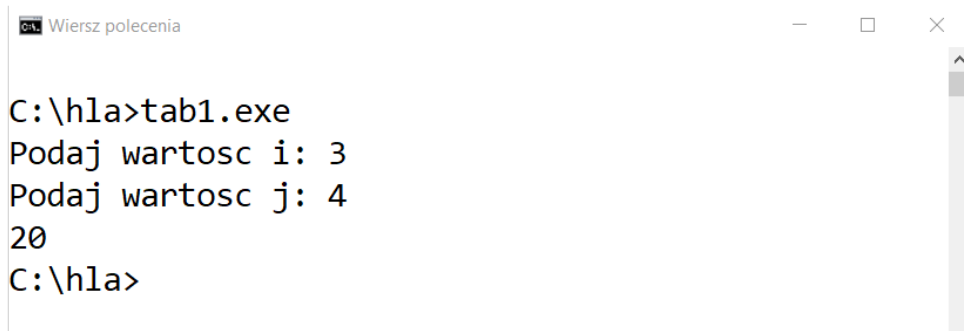
### **Polecenie 1**

Przeanalizuj działanie programu wykorzystującego odwołanie do elementu tablicy dwuwymiarowej w układzie wierszowym:

```
#include ("stdlib.hhf");  
  
static  
    i:    int32;  
    j:    int32;  
    tab: int32[4,8]:=[  
        1,2,3,4,5,6,7,8,  
        9,10,11,12,13,14,15,16,  
        17,18,19,20,21,22,23,24,  
        25,26,27,28,29,30,31,32  
    ];  
  
begin tab1;  
    stdout.put( "Podaj wartosc i: " );  
    stdin.get(i);  
    stdout.put("Podaj wartosc j: " );  
    stdin.get(j);  
    dec(i);  
    dec(j);  
    mov(i,ebx);  
    intmul(8,ebx);  
    add(j,ebx);  
    stdout.put(tab[ebx*4]);  
end tab1;
```

Zastosowanie polecenia niskopoziomowego *dec* pozwala na zmniejszenie wprowadzonych indeksów wiersza i kolumny numerowanych od jednego na numerację od zera, potrzebną do prawidłowego obliczenia przesunięcia elementu tablicy względem adresu bazowego.

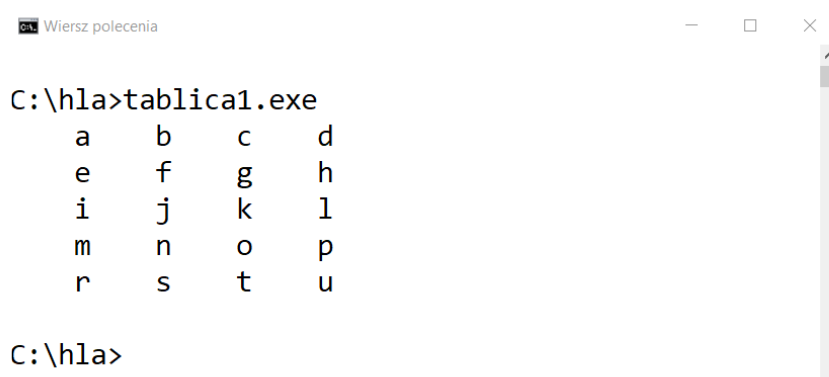
Działanie programu:



```
C:\hla>tab1.exe
Podaj wartosc i: 3
Podaj wartosc j: 4
20
C:\hla>
```

## Polecenie 2

Napisz program, w którym zostanie zadeklarowana, a następnie wyświetlona tablica znaków od **a** do **u** w następującym układzie:



```
C:\hla>tablica1.exe
  a    b    c    d
  e    f    g    h
  i    j    k    l
  m    n    o    p
  r    s    t    u

C:\hla>
```

## Polecenie 3

Zmodyfikuj program z polecenia 3 tak by tablica znaków uzupełniana była samoczynnie w trakcie działania programu. Następnie tablica powinna zostać wyświetlona.

## Tablice wielowymiarowe

Określenie przesunięcia adresu elementu tablicy względem adresu bazowego dla tablicy trzymiwymiarowej o wierszowym układzie elementów:

$$\text{Adres elementu} = \text{adres bazowy} + ((\text{index\_warstwy} * \text{rozmiar kolumny} + \text{index\_kolumny}) * \text{rozmiar wiersza} + \text{index\_wiersza}) * \text{rozmiar\_elementu}$$

Określenie przesunięcia adresu elementu tablicy względem adresu bazowego dla tablicy czterowymiarowej o wierszowym układzie elementów:



**Fundusze Europejskie**  
Wiedza Edukacja Rozwój



**Rzeczpospolita  
Polska**

**Unia Europejska**  
Europejski Fundusz Społeczny





```
Adres elementu = adres bazowy + ((index * rozmiar warstwy +  
index_warstwy) * (rozmiar kolumny + index_kolumny) *  
rozmiar_wiersza + index_wiersza) * rozmiar_elementu
```

Określenie przesunięcia adresu elementu tablicy względem adresu bazowego dla tablicy trzywymiarowej o kolumnowym układzie elementów:

```
Adres elementu = adres bazowy + ((index_wiersza * rozmiar  
kolumny + index_kolumny) * rozmiar_warstwy + index_warstwy) *  
rozmiar_elementu
```

Określenie przesunięcia adresu elementu tablicy względem adresu bazowego dla tablicy czterowymiarowej o kolumnowym układzie elementów:

```
Adres elementu = adres bazowy + ((index_wiersza *  
rozmiar_kolumny + index_kolumny) * (rozmiar_warstwy +  
index_warstwy) * rozmiar + index) * rozmiar_elementu
```

#### **Polecenie 4**

Napisz program, w którym zostanie zadeklarowana tablica trzymiarowa o następującym układzie elementów:

```
Tab3w:      int32[3,4,5]:=[  
  
            111 ,112 ,113 ,114 ,115,  
            121 ,122 ,123 ,124 ,125,  
            131 ,132 ,133 ,134 ,135,  
            141 ,142 ,143 ,144 ,145,  
  
            211 ,212 ,213 ,214 ,215,  
            221 ,222 ,223 ,224 ,225,  
            231 ,232 ,233 ,234 ,235,  
            241 ,242 ,243 ,244 ,245,  
  
            311 ,312 ,313 ,314 ,315,  
            321 ,322 ,323 ,324 ,325,  
            331 ,332 ,333 ,334 ,335,  
            341 ,342 ,343 ,344 ,345,  
  
            ];
```

Po podaniu indeksu warstwy wiersza i kolumny, program powinien wyświetlić zadany element tablicy.

## **LABORATORIUM 8. PROCEDURY**

Cel laboratorium:

Celem zajęć jest zapoznanie z procedurami w programowaniu niskopoziomym

Zakres tematyczny zajęć:

- deklarowanie procedury
- wywołanie procedury
- wyjście z procedury przed zakończeniem jej działania
- rola stosu w przechowywaniu danych
- zmienne lokalne i globalne

### **Zadanie 8.1 Pierwsza procedura**

Procedura to zestaw instrukcji realizujących pewne obliczenie, bądź inną funkcję programu. Procedura to zestaw reguł, którego zastosowanie powinno doprowadzić do uzyskania pewnego wyniku czy efektu. W pewnym miejscu kodu następuje wywołanie procedury, podejmowane jest wykonanie kodu procedury, po czym sterowanie jest przekazywane z powrotem z kodu procedury do programu.

Procedurę wywołuje się za pomocą instrukcji maszynowej *call*:

```
call (nazwa procedury);
```

Procedurę można również wywołać podając jej nazwę. Sposób ten umożliwia wywołanie procedury bez parametrów wejściowych bądź z parametrami wejściowymi:

```
nazwaprocedury();
```

Najprostsza deklaracja procedury przyjmuje postać:

```
procedure nazwa-procedury;  
    //Deklaracje lokalne względem procedury;  
begin nazwa-procedury;  
    //Instrukcje tworzące kod procedury;  
end nazwa-procedury;
```

Deklaracje procedur powinny być umieszczane w części deklaracyjnej programu.

### **Polecenie 1**

Przeanalizuj działanie programu, który:

- uzupełnia wartościami od 1 do 256 tablicę (wielkość elementów - podwójne słowo)
- wyświetla uzupełnioną tablicę
- zawiera procedurę, która:

powoduje wyzerowanie słów, począwszy od numeru elementu tablicy przechowywanego w rejestrze. (zerowanie wykonywane jest z wykorzystaniem rejestrów, bez użycia zmiennych)

- wyświetla tablicę po wywołaniu procedury

```
program z1;
#include ("stdlib.hhf");

static
    tab: int32[256];
    i: int32 := 1;
procedure zero;
begin zero;
    for (mov(ebx,ecx); ecx<256; inc(ecx)) do
        mov(0,tab[ecx*4]);
    endfor;
end zero;          //zerowanie tablicy
begin z1;
    mov(100,ebx);
    for (mov(0,ecx); ecx<256; inc(ecx)) do
        mov(i,tab[ecx*4]);
        inc(i);
    endfor;          //wczytanie wartosci do tablicy

    for (mov(0,ecx);ecx<256;inc(ecx)) do
        stdout.put(tab[ecx*4],' ')
    endfor;          //wyswietlanie tablicy
zero();

    stdout.newln();
    for (mov(0,ecx);ecx<256;inc(ecx)) do
        stdout.put(tab[ecx*4],' ')
    endfor;          //wyswietlanie tablicy po zerowaniu

end z1;
```



Działanie programu:

```
Wiersz polecenia

C:\hla>z1a.exe
1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28
29 30 31 32 33 34 35 36 37 38 39 40 41 42 43 44 45 46 47 48 49 50 51 52 5
3 54 55 56 57 58 59 60 61 62 63 64 65 66 67 68 69 70 71 72 73 74 75 76 77
78 79 80 81 82 83 84 85 86 87 88 89 90 91 92 93 94 95 96 97 98 99 100 101
102 103 104 105 106 107 108 109 110 111 112 113 114 115 116 117 118 119 12
0 121 122 123 124 125 126 127 128 129 130 131 132 133 134 135 136 137 138
139 140 141 142 143 144 145 146 147 148 149 150 151 152 153 154 155 156 15
7 158 159 160 161 162 163 164 165 166 167 168 169 170 171 172 173 174 175
176 177 178 179 180 181 182 183 184 185 186 187 188 189 190 191 192 193 19
4 195 196 197 198 199 200 201 202 203 204 205 206 207 208 209 210 211 212
213 214 215 216 217 218 219 220 221 222 223 224 225 226 227 228 229 230 23
1 232 233 234 235 236 237 238 239 240 241 242 243 244 245 246 247 248 249
250 251 252 253 254 255 256
1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28
29 30 31 32 33 34 35 36 37 38 39 40 41 42 43 44 45 46 47 48 49 50 51 52 5
3 54 55 56 57 58 59 60 61 62 63 64 65 66 67 68 69 70 71 72 73 74 75 76 77
78 79 80 81 82 83 84 85 86 87 88 89 90 91 92 93 94 95 96 97 98 99 100 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0
C:\hla>
```

## Przedwczesny powrót z procedury

Możliwe jest opuszczenie kodu procedury przed osiągnięciem przez program właściwego końca procedury – opuszczenie takie umożliwiają instrukcje *exit* i *exitif*.

W kodzie źródłowym programu wykorzystuje się je następująco:

```
exit nazwa-procedury;
```

```
exitif( wyrażenie-logiczne ) nazwa-procedury;
```

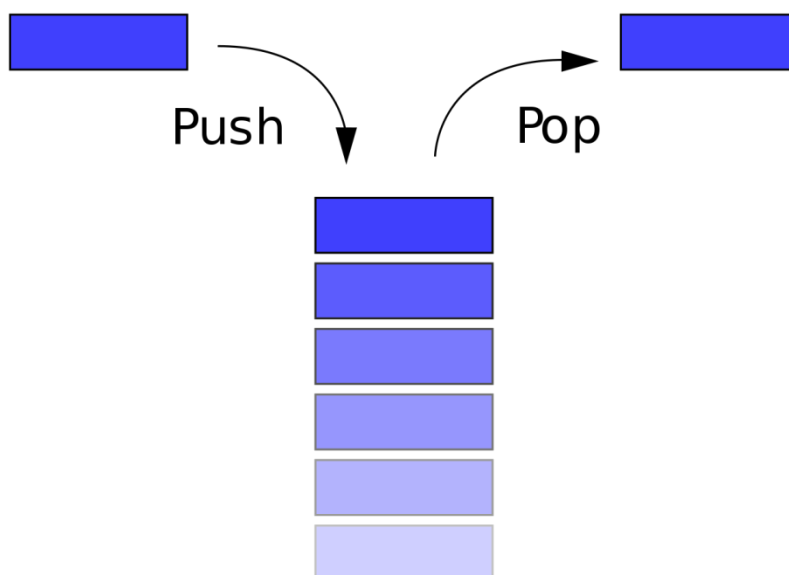
## Polecenie 2

Zmodyfikuj program z polecenia 1 tak by wykorzystywał instrukcję przerwania wykonania kodu procedury.

## Zadanie 8.2 Wykorzystanie stosu w procedurach

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).



Rys 8.1 Schemat zapisu na stosie

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.

Odkładanie danych na stos realizowane jest za pomocą instrukcji niskopoziomowej *push*:

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

Do wykorzystania są również instrukcje *pushw* i *pushd*, których operandami są zawsze stałe o rozmiarze odpowiednio słowa bądź podwójnego słowa.

Działanie instrukcji *push* można rozpisać następującym pseudokodem:

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

[ESP] := wartość-operandu

Jeżeli rejestr ESP zawiera wartość \$00FF\_FFE8 to wykonanie instrukcji *push( EAX )*; spowoduje ustawienie rejestru ESP na wartość \$00FF\_FFE4 i skopiowanie bieżącej wartości rejestru EAX pod adres \$00FF\_FFE4.

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.

Zdejmowanie danych realizowane jest za pomocą instrukcji *pop*:

```
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.

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 polecenia *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.

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 należy stosować jako kolejkę 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.

Dodatkowe 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.

Instrukcje *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. 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**

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.

Przykład:

Ze szczytu stosu należy usunąć dwie wartości o rozmiarze podwójnego słowa. Efekt usunięcia ich ze stosu można osiągnąć dodając do wskaźnika stosu liczbę "osiem":



**Fundusze Europejskie**  
Wiedza Edukacja Rozwój



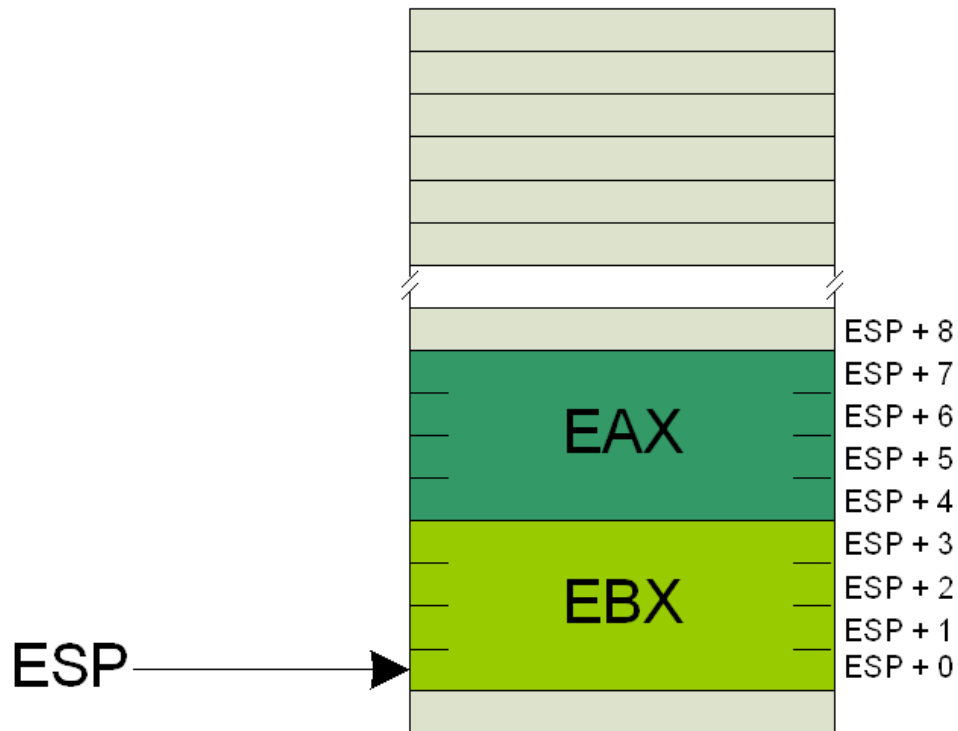
**Rzeczpospolita  
Polska**

**Unia Europejska**  
Europejski Fundusz Społeczny



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

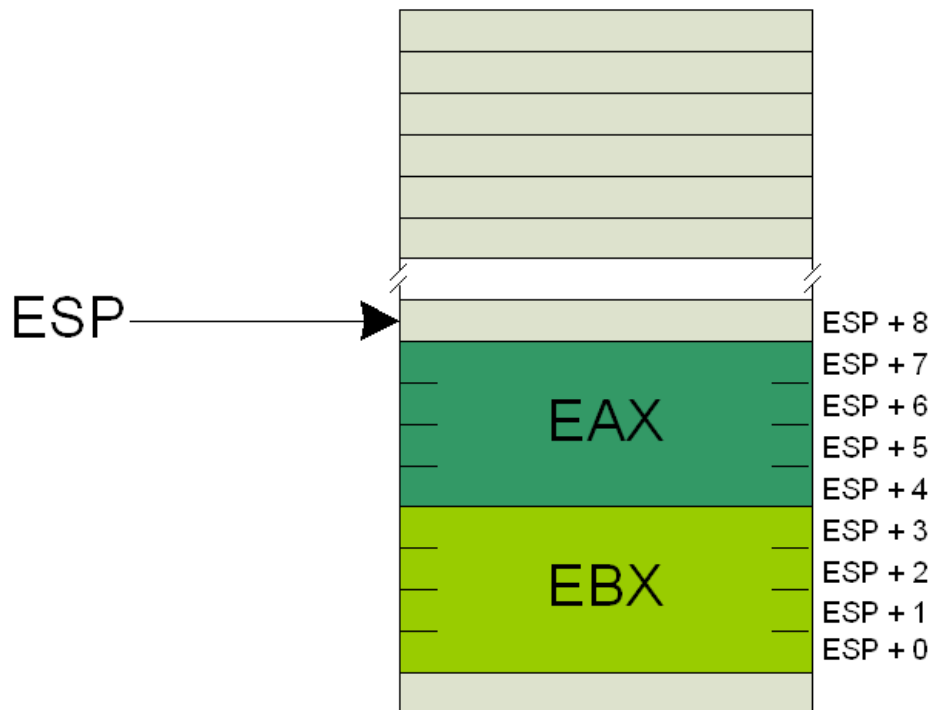
Obraz pamięci stosu przed wykonaniem instrukcji *add( 8, ESP )*:



*Rys 8.2 Schemat zapisu na stosie*



Obraz pamięci stosu po wykonaniu instrukcji `add( 8, ESP )`:



Rys 8.3 Schemat zapisu na stosie

Odpowiedzialność za zachowanie wartości modyfikowanych rejestrów może przyjąć:

- wywołujący (tak nazywany jest kontekst, z poziomu którego nastąpiło wywołanie procedury),
- wywołany (czyli sam kod procedury).

### Polecenie 1

Napisać program, który

- wypisuje na ekranie 20 wierszy, w których umieszcza 40 spacji oraz pojedynczy znak gwiazdki na końcu każdego wiersza.
- procedura wypisuje spacje i gwiazdkę, program główny wypisuje wiersze.
- do wykorzystania jest tylko rejestr ECX !

Działanie programu:



```
Wiersz polecenia
C:\hla>wiersze.exe
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
C:\hla>
```

## Zmienne lokalne

Są to zmienne dostępne wyłącznie w ramach kodu procedury, nie będąc dostępnymi z poziomu kodu wywołującego procedurę.

Deklaracje zmiennych lokalnych są identyczne z deklaracjami zmiennych programu głównego – sekcję deklaracji takich zmiennych osadza się w części deklaracyjnej procedury (a nie programu).

W części deklaracyjnej procedury można deklarować dowolne elementy, których deklarowanie jest dozwolone w części deklaracyjnej programu głównego: typy, stałe, zmienne, a nawet inne procedury. W sekcji deklaracyjnej procedury tworzymy sekcję *var*:

```
procedure zmlokalne;
  var
    i: int32;
    j: int32:=36;
```

Zmienne lokalne charakteryzują się dwoma atrybutami odróżniającymi je od zmiennych programu głównego (zmiennych globalnych):

- zasięgiem leksykalnym, który określa on widoczność, (a tym samym możliwość wykorzystania) w kodzie źródłowym programu.
- czasem życia, który określa momenty, w których dla zmiennej przydzielana jest pamięć i w których pamięć ta jest zwalniana – zmienna może przechowywać wartości jedynie pomiędzy tymi momentami.

W procedurach można odwoływać się do zmiennych globalnych.

Deklaracje procedur mogą występować w dowolnym miejscu części deklaracyjnej programu, a więc potencjalnie również przed miejscem zadeklarowania wykorzystywanej w procedurze zmiennej czy zmiennych.

W kodzie procedury można odwołać się do dowolnego obiektu deklarowanego w sekcji *static*, dostęp do takich obiektów realizowany jest identycznie jak z poziomu kodu programu głównego, a więc sprowadza się do określenia nazwy obiektu.

Zmienne lokalne, deklarowane wewnątrz deklaracji procedury są dostępne wyłącznie z poziomu tej procedury.

Czas życia zmiennych jest okresem pomiędzy przydzieleniem pamięci dla zmiennej, a tej samej pamięci zwolnieniem. Czas życia zmiennej jest przy tym atrybutem dynamicznym (kontrolowanym w fazie wykonania programu) w przeciwieństwie do zasięgu zmiennej, który jest atrybutem statycznym (kontrolowanym w czasie kompilacji).

Dla zmiennych lokalnych nie obowiązuje reguła niepowtarzalności identyfikatorów (nazw zmiennych):

```
program zmlokalne;
#include ("stdlib.hhf");

static
    i: int32:=10;
    j: int32:=20;
procedure pierwsza;
    var
        i: int32;
        j: int32;
```

Niepowtarzalne muszą być wszystkie identyfikatory wewnątrz danego zasięgu.

## **Polecenie 2**

Napisz program, w którym:

- Zadeklarowane zostaną dwie zmienne globalne i przyporządkowane zostaną im wartości 100 oraz 200.
- Zadeklarowana zostanie procedura, w której zadeklarowane zostaną zmienne lokalne o takich samych nazwach



- W sekcji wykonawczej procedury jednej zmiennej powinna zostać przyporządkowana wartość 10. Następnie procedura powinna wyświetlać wartość pierwszej zmiennej lokalnej od 10 do 0 i drugiej zmiennej lokalnej od 0 do 10.
- W programie głównym powinna zostać wywołana procedura a następnie wyświetlone zmienne globalne.

Działanie programu:

 Wiersz polecenia

```
C:\hla>zmlokalne.exe
Operacje na zmiennych lokalnych:
i=0    j=10
i=1    j=9
i=2    j=8
i=3    j=7
i=4    j=6
i=5    j=5
i=6    j=4
i=7    j=3
i=8    j=2
i=9    j=1
Zmienne globalne:
i=100  j=200

C:\hla>
```



## **LABORATORIUM 9. WYWOŁANIE PARAMETRYCZNE I FUNKCJE**

Cel laboratorium:

Celem zajęć jest zapoznanie z wywołaniami parametrycznymi i funkcjami w programowaniu niskopoziomowym

Zakres tematyczny zajęć:

- Wywołanie parametryczne
- Dyrektywa #include
- Funkcje w HLA
- Atrybut @returns

### **Zadanie 9.1 Wywołanie parametryczne**

W przypadku procedur sparametryzowanych, wykonanie kodu procedury uzależnione jest od określenia pewnych danych wejściowych.

Istnieją dwa sposoby przekazywania do procedury wartości reprezentujących poszczególne parametry:

- przekazywanie przez wartość
- przekazywanie przez adres

Argumenty przekazywane przez wartość pełnią zawsze rolę parametrów jednokierunkowych – konkretnie wejściowych.

Jeśli zmienna jest do procedury przekazywana przez wartość, to kod procedury nie może modyfikować tej zmiennej.

W sekcji deklaracji umieszczamy:

```
Procedure nazwaprocedury(zmienna: typ_zmiennej);
```

Wywołanie procedury:

```
nazwa_procedury( stała );  
nazwa_procedury ( rejestr32 );  
nazwa_procedury ( zmienna );
```

Przykłady poprawnych wywołań procedury pierwszaprocc:

```
pierwszaprocc( 40 );  
pierwszaprocc( eax );  
pierwszaprocc( nazwa_zmiennej );
```

Zwane również przekazywaniem przez referencję polega na przekazaniu – w miejsce wartości – adresu obiektu.

Do procedury nie są przekazywane dane, a jedynie wskaźnik na te dane.



Aby dla danego parametru określić tryb przekazywania przez adres, należy deklarację owego parametru poprzedzić słowem `var`:

```
procedure nowaproc( var zmienna: int32 );
```

Wywołanie procedury z przekazaniem argumentu przez referencję nie różni się od analogicznego wywołania z przekazaniem argumentu przez wartość, tyle że argument musi być określony jako adres w pamięci – argumentem nie może być ani stała, ani rejestr.

Wywołanie procedury:

```
procedural( i32 ); //zmienna i32 ma tu typ int32  
procedural(type int32 [ebx]);
```

Przekazywanie przez adres jest zwykle mniej efektywne niż przekazywanie przez wartość, ponieważ w kodzie procedury należy uwzględnić konieczność wyłuskiwania parametru w każdym odwołaniu do przekazanej wartości.

Pośrednie odwołanie wymaga zwykle co najmniej dwóch instrukcji maszynowych, jednak w przypadku większych struktur danych efektywność przekazywania argumentów przez adres rośnie ponieważ alternatywą jest potencjalnie czasochłonne kopiowanie wszystkich bajtów takich struktur do pamięci parametrów procedury.

W przypadku wielu parametrów wywołania wykorzystywany jest zapis pozycyjny parametrów.

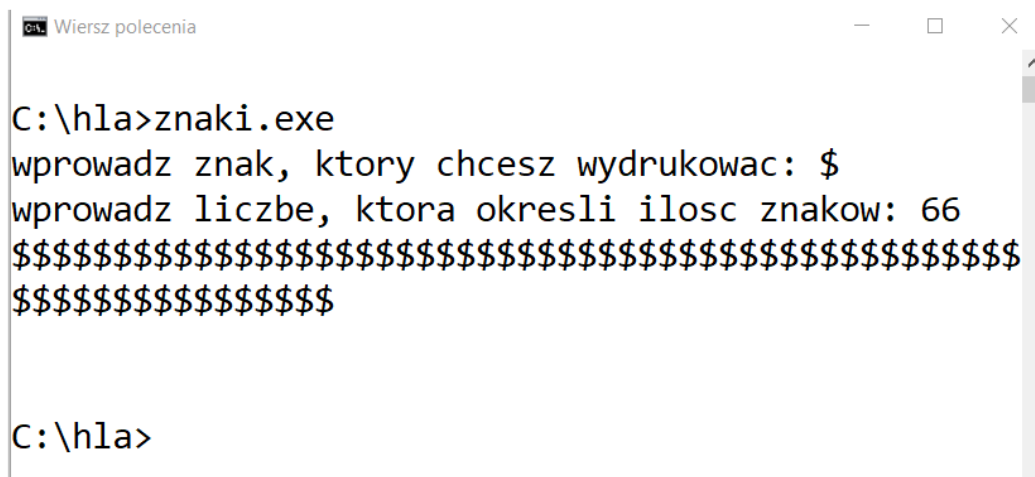
- Przy deklaracji procedury parametry oddzielane są znakiem **;**
- Przy wywołaniu procedury parametry oddzielane są znakiem **,**

### **Polecenie 1 Program znaki**

Napisz program, w którym wykorzystana zostanie procedura o dwóch parametrach wywołania:

- Stworzyć procedurę, która drukuje na ekranie określoną liczbę znaków
- Procedura powinna posiadać dwa parametry wywołania: liczbę znaków oraz rodzaj (wygląd) znaku
- Liczba znaków i wygląd znaku wprowadzane są z klawiatury.

Przykład działania programu:



```
C:\hla>znaki.exe
wprowadz znak, ktory chcesz wydrukowac: $
wprowadz liczbe, ktora okresli ilosc znakow: 66
$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$
$$$$$$$$$$$$$$$$$$$$
C:\hla>
```

## Zadanie 9.2 Dyrektywa #include

Dyrektywa *#include* sygnalizuje konieczność wstawienia w miejscu dyrektywy zawartości pliku określonego argumentem dyrektywy.

W tak wstawianym do kodu źródłowego pliku można na przykład zgrupować definicje wykorzystywanych w programie stałych, procedur czy innych obiektów programu. Definicje te można następnie wykorzystywać w wielu różnych programach.

Dyrektywy *#include* mogą być zagnieżdżane również w plikach włączanych, a dalej w plikach włączanych do plików włączanych i tak dalej. Włączenie jednego pliku do kompilacji może pociągnąć za sobą konieczność włączenia do niej całego zestawu plików.

Znakomitym przykładem zagnieżdżania dyrektyw *#include* jest włączany często do programów w języku HLA plik `stdlib.hhf`. Plik ów nie jest niczym więcej jak tylko listą kolejnych plików do włączenia. Włączając do pliku kodu źródłowego plik `stdlib.hhf`, automatycznie włączamy do kodu wszystkie moduły biblioteki HLA.

Przykład wykorzystania deklaracji dyrektywy *#include* w programie `znaki` z polecenia 1:

```
program znaki;
#include( "stdlib.hhf");
#include ("drukznakow.hhf");
```

Plik `drukznakow.hhf` należy umieścić w katalogu głównym HLA.

## Polecenie 2

Napisz program kalkulator (dodawanie, odejmowanie, mnożenie i dzielenie) korzystając z procedur dla każdej operacji. Do rozwiązania wykorzystaj dyrektywę *#include*.



### Zadanie 9.3 Funkcje

Funkcje to procedury, które zwracają wyniki. Efektem wykonania kodu procedury jest realizacja konkretnego zadania, w ramach funkcji podobna sekwencja instrukcji maszynowych służy do ustalenia pewnego wyniku (wartości funkcji), który jest następnie zwracany do wywołującego.

Procedura staje się funkcją, kiedy programista zdecyduje o przekazaniu pewnej określonej i mającej jakieś umowne znaczenie wartości poza kod procedury.

Sposób deklaracji funkcji nie różni się niczym od deklaracji procedury – składniowo są to obiekty identyczne.

Wartości funkcji najczęściej przekazywane są do wywołującego za pośrednictwem rejestrów ogólnego przeznaczenia. Obowiązuje ogólnie przyjęta konwencja w przypadku assemblerów dla procesorów rodziny 80x86, zakładająca zwracanie 8-bitowych, 16-bitowych i 32-bitowych wartości całkowitych za pośrednictwem rejestrów AL, AX i EAX (odpowiednio).

Bardzo często funkcje zwracają wartości 64-bitowe w parze rejestrów EDX:EAX (na przykład funkcja `stdin.geti64` biblioteki standardowej języka HLA zwraca w tej parze rejestrów wczytaną z wejścia 64-bitową wartość całkowitą).

#### Atrybut `@returns`

Możliwe jest określenie w deklaracji procedury specjalnego atrybutu, który określać będzie napis interpretowany przez kompilator jako „wartość zwracana” instrukcji wywołania danej procedury, kiedy wywołanie to zostanie w wyniku złożenia osadzone w miejscu operandu innej instrukcji. Składnia deklaracji procedury rozszerzona o atrybut `@ returns` prezentuje się następująco:

```
procedure nazwa-procedury ( opcjonalna-lista-parametrów );
@returns( napis );
    //Deklaracje lokalne względem procedury;
begin nazwa-procedury;
    //Instrukcje tworzące kod procedury;
end nazwa-procedury;
```

Atrybut `@returns` wymaga określenia pomiędzy znakami nawiasów pojedynczego literału łańcuchowego; assembler będzie podstawiał ów literał wszędzie tam, gdzie instrukcja wywołania procedury zostanie wykorzystana w roli operandu innej instrukcji.

Zwykle *napis* określa nazwę jednego z rejestrów ogólnego przeznaczenia, dopuszczalne jest jednak określenie zupełnie dowolnego literału łańcuchowego, który będzie nadawał się do wykorzystania w roli operandu instrukcji, na przykład *napis* może określać nazwę zmiennej albo stałą reprezentującą konkretny adres w pamięci.

#### Polecenie 1

Stworzyć funkcję logiczną, zwracającą wartość logiczną „prawda” (1) bądź „fałsz” (0) za pośrednictwem rejestru EAX.

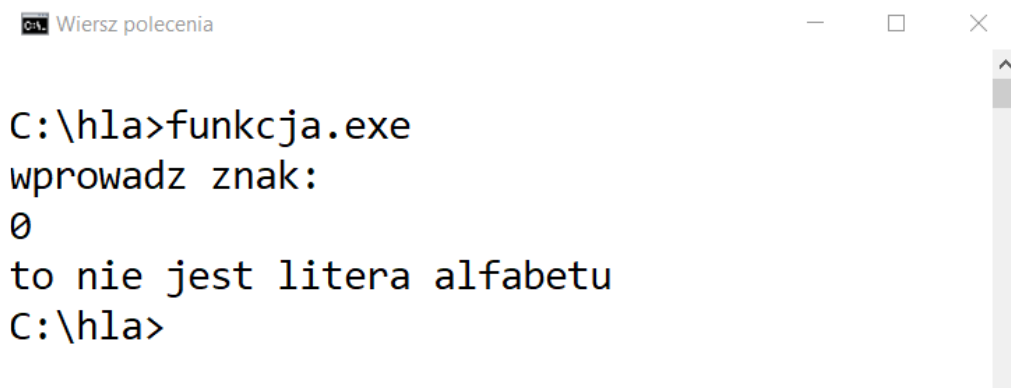
- Wartość 1 sygnalizuje, że określona w wywołaniu wartość znakowa reprezentuje literę alfabetu.



- Funkcję wywołać w programie, który wyświetli komentarz stosowny do tego, czy wprowadzona została litera alfabetu czy też inny znak.
- Jako argument warunku w programie głównym użyć nazwy procedury wg. przykładu:

```
if (nazwa_procedury(parametr)) then
    //instrukcje
else
    //instrukcje
endif;
```

Działanie programu:



```
C:\hla>funkcja.exe
wprowadz znak:
0
to nie jest litera alfabetu
C:\hla>
```

## **LABORATORIUM 10. TWORZENIE OKIEN I KOMUNIKATÓW PROGRAMOWANIE RÓWNOLEGŁE Z WYKORZYSTANIEM PAMIĘCI WSPÓŁDZIELONEJ I ROZPROSZONEJ. PROGRAMOWANIE SYSTEMÓW WIELOPROCESOROWYCH**

Cel laboratorium:

Celem zajęć jest opanowanie tworzenia okien i komunikatów w Windows

Zakres tematyczny zajęć:

- Pakiet masm32
- Windows API
- Platforma SDK, MSDN
- Komunikaty
- Pierwsze okna
- Programowanie równoległe
- Systemy wieloprocesorowe

### **Funkcje API**

Systemy Windows (łącznie z 3.1) udostępniają tzw. funkcje API (Application Programming Interface). Jest to zestaw procedur oferowanych programiście, służących do budowy aplikacji w danym systemie operacyjnym.

API to wszystko to, co dany system oferuje programiście (zbiór funkcji oferowanych przez system). W różnych wersjach Windows działają różne wersje API. Użycie rozszerzonych funkcji Windows API (z nazwami zakończonymi na Ex) jest pożądane wszędzie tam, gdzie mogą one zastąpić swoje podstawowe wersje (basic).

Potrzebne materiały

- Macro Assembler 32 pod Windows – dostępny np. na podanych stronach [www](http://www.microsoft.com), Windows,
- Win32 Programmer's Reference – plik pomocy zawierający opisy wszystkich funkcji API, dołączany do wielu pakietów programistycznych Delphi 5 (C++), dostępny również na stronach Microsoft'u,
- Edytor tekstu (notatnik, Quick Editor dołączony do masm lub inny).

Dokumentację funkcji Win API znajduje się w pakiecie Platform SDK (Software Development Kit) na stronach Microsoft:

<http://www.microsoft.com/msdownload/platformsdk/sdkupdate/>  
<http://msdn.microsoft.com>

Stałe elementy programu:

.386

;zestaw wykorzystywanych instrukcji



**Fundusze Europejskie**  
Wiedza Edukacja Rozwój



**Rzeczpospolita  
Polska**

**Unia Europejska**  
Europejski Fundusz Społeczny



```
.model flat, stdcall          ;model pamięci stosowany w
programie
option casemap:none          ;rozróżnianie wielkości liter
include \masm32\include\windows.inc    ;definicje stałych API
;instrukcje wczytania bibliotek
.const                          ;stałe
.data                          ;zmienne inicjalizowane (nadajemy wartości
początkowe)
.data?                         ;zmienne niezdefiniowane (podajemy tylko
typ, nie posiadają wartości początkowej)

.code
nasz_program: ;kod programu

end nasz_program
```

Niezbędne biblioteki, które zostaną wykorzystane w pierwszych przykładach znajdują się w bibliotekach kernel32 i user32. Wczytanie bibliotek realizowane jest następująco:

```
include \masm32\include\user32.inc
include \masm32\include\kernel32.inc
includelib \masm32\lib\user32.lib
includelib \masm32\lib\kernel32.lib
```

Możliwe jest dynamicznie dołączane biblioteki (ang. dynamically linked libraries, w skrócie DLL), zwane też bibliotekami DLL lub po prostu DLL'ami, są skompilowanymi modułami, zawierającymi kod (funkcje, zmienne, klasy itd.), który może być wykorzystywany przez wiele programów jednocześnie. Kod ten istnieje przy tym tylko w jednej kopii - zarówno na dysku, jak i w pamięci operacyjnej.

Biblioteki takie istnieją w postaci plików z rozszerzeniem .dll i są zwykle umieszczone w katalogu systemowym, względnie w folderach wykorzystujących je aplikacji. Udostępniają one (eksportują) zbiory symboli, które mogą być użyte (zaimportowane) w programach pracujących w Windows.

Z punktu widzenia programisty C++ korzystanie z kodu zawartego w bibliotekach DLL nie różni się wiele od stosowania zasobów Biblioteki Standardowej. Różnica polega na tym, że biblioteki DLL nie są statycznie dołączane do pliku wykonywalnego aplikacji, lecz linkowane dynamicznie (stąd ich nazwa) w czasie działania programu. W pliku EXE muszą się jedynie znaleźć informacje o nazwach wykorzystywanych bibliotek oraz o symbolach, które są z nich importowane. Dane te są automatycznie zapisywane przez kompilator jako tzw. tabele importu.

Wyodrębnienie kluczowego kodu systemu Windows w postaci bibliotek DLL likwiduje zatem wszystkie dolegliwości związane z jego wykorzystaniem w aplikacjach. Mechanizm dynamicznych bibliotek pozwala ponadto na tworzenie innych, własnych skarbnic kodu, które mogą być współużytkowane przez wiele programów. Windows API jest zawarte w bibliotekach DLL. Wraz z kolejnymi wersjami systemu bibliotek tych przybywało - pojawiły się moduły odpowiedzialne za multimedia, komunikację sieciową, internet i jeszcze wiele innych.

Najważniejsze trzy z nich były jednak obecne od samego początku i to one tworzą zasadniczą część interfejsu programistycznego Windows. Są to:

- kernel32.dll - w niej zawarte są funkcje sterujące jądrem (ang. kernel) systemu, zarządzające pamięcią, procesami, wątkami i innymi niskopoziomowymi sprawami, które są kluczowe dla funkcjonowania systemu operacyjnego.
- user32.dll - odpowiada za graficzny interfejs użytkownika, czyli za okna - ich wyświetlanie i interaktywność.
- gdi32.dll - jest to elastyczna biblioteka graficzna, pozwalająca rysować skomplikowane kształty, bitmapy oraz tekst na dowolnym rodzaju urządzeń wyjściowych. Zapoznamy się z nią w rozdziale 3, Windows GDI.

Sekcja kodu rozpoczyna się etykietą (nazwą programu), zakończoną znakiem ":"

```
pierwszy:
```

Koniec sekcji kodu sygnalizowany jest poleceniem

```
end <nazwa_programu>
```

Do wyświetlania komunikatów wykorzystywana jest funkcja API o nazwie MessageBox.

Parametry MessageBox:

- uchwyt okna nadrzędnego (w przypadku braku – 0)
- adres w pamięci tekstu, który zostanie wyświetlony w głównej części okna
- adres tekstu, który pojawi się w pasku tytułowym
- styl komunikatu (możliwość wybrania przycisków i ikony, które pojawią się w oknie).

wywołanie funkcji wyświetlającej okno z komunikatem:

```
invoke MessageBox,0,addr tekst_komunikatu,addr  
tekst_tytulu,MB_OK
```

Wywołanie:

w miejscu przeznaczonym na kod, parametry funkcji podajemy po przecinku aby kompilator zamienił całe wyrażenie na adres, a nie zawartość zmiennej należy użyć polecenia "addr"

tekst\_tytulu – zmienna w pamięci (określonego typu)

tekst\_komunikatu – zmienna w pamięci (określonego typu)

MB\_OK – przycisk <OK>

### **Okno komunikatu - pozostałe dane**

definicje zmiennych tekst\_tytulu oraz tekst\_komunikatu:

```
tekst_tytulu db "Tytul pierwszego komunikatu",0  
tekst_komunikatu db "Tresc okna pierwszego komunikatu.",0
```

zmienne definiujemy w sekcji data

db – oznacza typ zmiennej (jeden bajt)

inne typy:     dw     word – dwa bajty  
              dd     double word – cztery bajty  
              dq     quad word – osiem bajtów

bajt 0 kończy każdy ciąg tekstowy

### **Zakończenie sekcji kodu**

Aby program nie wywoływał nieprawidłowej operacji, należy go zakończyć. Służy do tego funkcja `ExitProcess`:

```
invoke ExitProcess,0
```

funkcja powinna zostać wywołana na końcu sekcji kodu.

### **Wybrane definicje przycisków**

- `MB_ABORTRETRYIGNORE` – przerwij, ponów próbę, zignoruj
- `MB_OKCANCEL` – ok, anuluj
- `MB_RETRYCANCEL` – ponów próbę, anuluj
- `MB_YESNO` – tak, nie
- `MB_YESNOCANCEL` – tak, nie, anuluj

### **Wybrane definicje ikon**

- `MB_ICONEXCLAMATION`     - wykrzyknik
- `MB_ICONWARNING`        - wykrzyknik
- `MB_ICONINFORMATION` - mała litera "i"
- `MB_ICONASTERISK`        - mała litera "i"
- `MB_ICONQUESTION`       - znak zapytania
- `MB_ICONSTOP`            - znak zapytania
- `MB_ICONERROR`           - znak zapytania
- `MB_ICONHAND`           - krzyżyk

wyświetlenie ikony – po słowie `or` (podanym w parametrach wywołania procedury `MessageBox`)

### **Monitorowanie naciśnięcia klawisza/przycisku**



**Fundusze Europejskie**  
Wiedza Edukacja Rozwój



**Rzeczpospolita  
Polska**

**Unia Europejska**  
Europejski Fundusz Społeczny



Informacja o wybranym przycisku jest przeznaczona dla programu, a otrzymuje on ją poprzez wynik funkcji MessageBox(). Jest to liczba typu int, która przyjmuje wartość jednej z następujących stałych: IDABORT, IDCANCEL, IDIGNORE, IDNO, IDOK, IDRETRY, IDYES.

Funkcje API zawsze zwracają wartości w rejestrze EAX. Aby sprawdzić czy został wciśnięty TAK, należy porównać rejestr eax ze stałą IDYES.

Jeżeli okno komunikatu nie zostało utworzone (np. błąd programu, brak pamięci), funkcja MessageBox zwraca w eax wartość 0. Jeżeli wywołanie funkcji MessageBox powiedzie się, do eax zwracane zostają następujące wielkości:

IDABORT	wybrany przycisk "Zamknij"
IDCANCEL	wybrany przycisk "Anuluj"
IDIGNORE	wybrany przycisk "Ignoruj"
IDNO	wybrany przycisk "Nie"
IDOK	wybrany przycisk "Tak"
IDRETRY	wybrany przycisk "Ponów"
IDYES	wybrany przycisk "Tak"

Składnia warunku "if"

```
.IF <warunek>
    polecenia
.ELSEIF <warunek>
    polecenia
.ELSE
    polecenia
.ENDIF
```

## Warunki

Istnieje możliwość wykorzystania następujących operatorów:

==	równy
>	wiekszy
>=	wiekszy lub równy
<	mniejszy
<=	mniejszy lub równy
!=	różny

## Spójniki logiczne

	alternatywa
&&	koniunkcja
!	negacja



**Fundusze Europejskie**  
Wiedza Edukacja Rozwój



**Rzeczpospolita  
Polska**

**Unia Europejska**  
Europejski Fundusz Społeczny



## **Polecenie 1**

Napisz program, którego zadaniem będzie wyświetlenie komunikatu wg założeń:

Program-komunikat: **Sonda studencka**

Pytanie: **Czy zrozumiałeś zasady tworzenia komunikatów? [ikona – znak zapytania]**

Możliwe odpowiedzi: **Tak, Nie, Anuluj**

Po wybraniu Tak: **Możemy zatem przejść do następnej części [ikona wykrzyknik]**

Po wybraniu Nie: **Szkoda, ćwicz więcej [ikona wykrzyknik]**

Po wybraniu Anuluj: **Nie odpowiedziałeś na pytanie [ikona krzyżyk]**

## **Okna Windows**

Do tworzenia okien w Windows wykorzystuje się m.in. funkcję API CreateWindowEx. Funkcja potrzebuje szeregu parametrów, których uzyskanie jest możliwe z innych funkcji. Znajdują się one w bibliotekach systemowych: user32 oraz kernel32.

Okna tworzą hierarchię: pewne okno może być nadrzędnym dla innego, podrzędnego. Na szczycie tej hierarchii widnieje pulpit - okno, które istnieje przez cały czas działania systemu. Bezpośrednio podległe są mu okna poszczególnych aplikacji (lub inne okna systemowe), zaś dalej hierarchia może sięgać aż do pojedynczych kontrolki (przycisków itd.).

Żadne okno w systemie Windows nie istnieje jednak samo dla siebie. Zawsze musi być ono związane z jakimś programem, a dokładniej z jego instancją.

Instancją programu (ang. application instance) nazywamy pojedynczy egzemplarz uruchomionej aplikacji.

## **Parametry funkcji CreateWindowEx**

- LPVOID lpParam
- HINSTANCE hInstance
- HMENU hMenu
- HWND hWndParent
- int nHeight
- int nWidth
- int y
- int x
- DWORD dwStyle



**Fundusze Europejskie**  
Wiedza Edukacja Rozwój



**Rzeczpospolita  
Polska**

**Unia Europejska**  
Europejski Fundusz Społeczny



- LPCTSTR lpWindowName
- LPCTSTR lpClassName
- DWORD dwExStyle

LPVOID lpParam - Ewentualny dodatkowy parametr, przekazywany do okna w chwili jego stworzenia. Zwykle nie ma takiej potrzeby, więc wpisujemy tu 0.

HINSTANCE hInstance

Uchwyt instancji (ang. handle to application instance) jest to liczba dzięki której Windows jest w stanie rozróżnić wszystkie uruchomione kopie tego samego programu. Jeżeli uruchomimy ten sam program dwa razy, Windows przydzieli każdej jego kopii osobny uchwyt. Uchwyt instancji dla naszego programu pobieramy za pomocą funkcji `GetModuleHandle`, która zwraca go w rejestrze `eax` (wszystkie API zwracają wynik swoich działań w `eax`).

```
.code
start:
    invoke     GetModuleHandle, 0
    mov     hInstance, eax
.data?
hInstance HINSTANCE ?
```

HINSTANCE jest to specjalny typ zmiennej służący właśnie do przechowywania uchwytu instancji.

HMENU hMenu - Uchwyt menu – w przypadku braku menu, wpisujemy wartość 0

HWND hWndParent - Uchwyt okna nadrzędnego względem tego tworzonego przez nas. W przypadku głównych okien aplikacji (ang. top-level) podajemy tu `NULL`. 0 w przypadku gdy tworzone okno będzie oknem głównym.

int nHeight - Wysokość okna, jaką umieszczamy tutaj, również można zostawić do ustalenia dla systemu operacyjnego przy pomocy stałej `CW_USEDEFAULT`.

int nWidth - W tym parametrze podajemy szerokość okna, którą przy pomocy `CW_USEDEFAULT` także może być wybrana domyślnie.

int x - Wpisujemy tutaj współrzędną poziomą okna lub `CW_USEDEFAULT` - wówczas jego pozycja zostanie ustalona domyślnie.

DWORD dwStyle - Jest to styl okna, w największym stopniu determinujący jego wygląd i zachowanie. Parametr ten jest kombinacją flag bitowych.

Stała `WS_OVERLAPPEDWINDOW`, powoduje stworzenie zwykłego okna z paskiem i przyciskami tytułu oraz skalowalnym obramowaniem. Jest to jednocześnie jeden z częściowo stosowanych stylów okna.



LPCTSTR lpWindowName - W tym parametrze wpisujemy tytuł okna - jest to jednocześnie tekst pojawiający się na jego pasku tytułu.

LPCTSTR lpClassName - Tutaj należy podać nazwę klasy, której przynależne będzie tworzone okno. Najczęściej jest to nasza własna klasa, zarejestrowana chwilę wcześniej; wartość tego parametru powinna być zatem taka sama, jak pola lpzClassName w strukturze WNDCLASSEX.

DWORD dwExStyle - Parametr ten jest kombinacją flag bitowych, stanowiącą tzw. rozszerzony styl okna (ang. extended window style). Styl określa raczej zaawansowane aspekty okna i dlatego zwykle wpisujemy w tym miejscu 0 (NULL). W takim przypadku możemy używać funkcji CreateWindow(), która od omawianej różni się tylko tym, iż w ogóle nie posiada parametru dwExStyle.

### **Tworzenie okna – klasa okna**

Utworzenie okna tego rodzaju, mogącego np. stanowić główną bazę jakiejś aplikacji, przebiega w dwóch etapach.

Najpierw musimy zarejestrować w Windows klasę okna, a następnie stworzyć jej egzemplarz. Obie te czynności mogą zostać przeprowadzone w funkcji WinMain().

Każde okno w systemie Windows należy do jakiejś klasy. Klasa okna (ang. window class) jest czymś w rodzaju wzorca, na podstawie którego tworzone są kolejne kopie okien. Wszystkie te okna, należące do jednej klasy, mają z początku pewne cechy wspólne oraz pewne odrębne, charakterystyczne tylko dla nich.

Najważniejszą właściwością klasy jest nazwa. W systemie Windows każda klasa okna musi posiadać swoją unikalną nazwę, poprzez którą można ją identyfikować. Podajemy to miano, gdy chcemy utworzyć okno na podstawie klasy.

Drugą bardzo ważną sprawą jest procedura zdarzeniowa, zajmująca się przetwarzaniem zdarzeń systemowych. Windows jest tak skonstruowany, że owa procedura jest związana właśnie z klasą okna - wynika stąd, że:

Wszystkie okna należące do jednej klasy reagują na zdarzenia przy pomocy tej samej procedury zdarzeniowej.

Pozostałe cechy klasy to np. tło, jakim jest wypełniane wnętrze okna (tzw. obszar klienta), ikonka, która pojawia się w jego lewym górnym rogu, wygląd kursora przemieszczającego się nad oknem, a także kilka innych opcji.

Wszystkie potrzebne informacje o klasie okna umieszczamy w specjalnej strukturze, a następnie przy pomocy odpowiedniej funkcji Windows API rejestrujemy klasę w systemie. Jeżeli operacja ta zakończy się sukcesem, możemy już przystąpić do utworzenia właściwego okna (lub okien) na podstawie zarejestrowanej klasy.

### **Rejestrowanie klasy okna**

Klasa okna musi zostać zarejestrowana w systemie, aby stała się dostępna dla aplikacji. Ponieważ zarejestrowanie klasy zajmie nam trochę miejsca, istnieje możliwość utworzenia procedury, w której zarejestrowana zostanie klasa.

Nowa procedura nazywać się będzie WinMain. Zanim ją utworzymy, dopiszmy kod który ją wywoła. Do wywoływania podprogramów służy instrukcja call po której podajemy adres, gdzie rozpoczyna się dana procedura (nie musimy ręcznie obliczać tego adresu, kompilator zrobi to za nas jeżeli wpiszemy po prostu nazwę wywoływanej procedury).

Definiowanie klasy okna:

Wywołujemy procedurę WinMain (treść procedury zapiszemy w dalszej części kodu)

```
.code  
call WinMain
```

Kończymy program:

```
.code  
invoke ExitProcess,0
```

Procedura WinMain

```
.code  
WinMain proc  
    LOCAL KlasaOkna:WNDCLASSEX
```

Zmienna lokalna (używana tylko w obrębie danej procedury), w której będziemy umieszczać wszystkie dane niezbędne do zarejestrowania klasy. Tą zmienną podamy następnie jako parametr dla funkcji RegisterClassEx.

zapisanie wartości pól

```
mov KlasaOkna.cbSize, sizeof(WNDCLASSEX)
```

rozmiar struktury WNDCLASSEX

```
mov KlasaOkna.style, CS_HREDRAW or CS_VREDRAW
```

styl klasy, możliwość podania jednej lub kilku wartości połączonych operatorem or. CS\_HREDRAW – okno będzie odmalowane po zmianie jego szerokości, CS\_VREDRAW – okno będzie odmalowane po zmianie jego długości.

```
mov KlasaOkna.lpfnWndProc, offset WndProc
```

adres procedury obsługi zdarzeń (WndProc), będzie zdefiniowana w dalszej części

```
push hInstance  
pop KlasaOkna.hInstance
```

uchwyt do instancji naszego programu



```
mov KlasaOkna.hbrBackground,COLOR_WINDOW+1  
  
mov KlasaOkna.lpszClassName,offset Klasa
```

Adres zmiennej zawierającej nazwę klasy. Po tej nazwie klasa jest rozpoznawana, m.in. w funkcji CreateWindowEx. Zmienna Klasa jeszcze nie istnieje, musimy więc utworzyć ją w sekcji data (zawartość zmiennej może być dowolna, nie może jednak określać klasy, która już występuje w systemie):

```
.data  
Klasa db "WinClass",0  
  
invoke LoadIcon,0,IDI_APPLICATION  
mov KlasaOkna.hIcon,eax  
mov KlasaOkna.hIconSm,eax
```

Wczytanie ikony z zasobów systemowych i ustawienie jej jako ikony okna alternatywne ikony:

- IDI\_ASTERISK                      0 gdyż nie posługujemy się własnym obrazkiem
- IDI\_EXCLAMATION
- IDI\_HAND
- IDI\_QUESTION
- IDI\_WINLOGO

```
invoke LoadCursor,0,IDC_ARROW  
mov KlasaOkna.hCursor,eax
```

Nazwy alternatywnych kursorów:

- IDC\_APPSTARTING
- IDC\_ARROW
- IDC\_CROSS
- IDC\_IBEAM
- IDC\_ICON
- IDC\_NO
- IDC\_SIZE
- IDC\_SIZEALL
- IDC\_SIZENESW
- IDC\_SIZENS
- IDC\_SIZENWSE
- IDC\_SIZEWE
- IDC\_UPNARROW
- IDC\_WAIT

Jeżeli mamy już gotowe wszystkie informacje o klasie okna, przychodzi czas na jej zarejestrowanie. Operacja ogranicza się do wywołania jednej funkcji:



```
invoke RegisterClassEx,addr KlasaOkna
```

### **Tworzenie okna - Procedura CreateWindowEx**

```
invoke      CreateWindowEx,  
    0,                ;styl rozszerzony  
    addr Klasa,        ;nazwa klasy  
    addr TytulOkna,    ;tytuł  
    WS_OVERLAPPEDWINDOW or WS_VISIBLE, ;styl  
    100,               ;współrzędna x      (CW_USEDEFAULT)  
    100,               ;współrzędna y      (CW_USEDEFAULT)  
    320,               ;szerokość          (CW_USEDEFAULT)  
    200,               ;wysokość           (CW_USEDEFAULT)  
    0,                ;uchwyt okna nadrzędnego  
    0,                ;uchwyt menu  
    hInstance,        ;uchwyt instancji  
    0,                ;dodatkowe dane/opcje
```

Brakuje jedynie tytułu okna. Jest to napis, który zostanie wyświetlony na pasku tytułowym u góry okna. Zmienną TytulOkna deklarujemy w sekcji data:

```
.data  
TytulOkna db "Moje pierwsze wymęczone okno",0
```

### **Pętla komunikatów**

Pętla komunikatów (ang. message loop) odpowiada za odbieranie od systemu Windows komunikatów o zdarzeniach i przesyłanie ich do docelowych okien aplikacji. Pętla ta wykonuje się przez cały czas trwania programu i odpowiada za jego właściwą interakcję z otoczeniem.

Kod pętli komunikatów może przedstawiać się następująco:

```
.WHILE TRUE  
    invoke      GetMessage,addr msg,0,0,0  
    .BREAK     .IF (!eax)  
    invoke TranslateMessage,addr msg  
    invoke DispatchMessage,addr msg  
.ENDW  
ret
```

Pętla komunikatów działa dopóty, dopóki program powinien zostać zakończony. Przez cały ten czas wykonuje przy tym bardzo pożyteczną pracę: pobiera nadchodzące informacje o zdarzeniach z kolejki komunikatów (ang. message queue) Windows, a następnie wysyła je do właściwych im okien. Kolejka komunikatów jest zaś wewnętrzną strukturą danych systemu operacyjnego, istniejącą dla każdej uruchomionej w nim aplikacji. Na jeden koniec tej kolejki trafiają wszystkie komunikaty o zdarzeniach, jakie pochodzą ze wszystkich możliwych źródeł w systemie; z drugiego jej końca program pobiera te komunikaty i reaguje na nie zgodnie z

intencją programisty. W ten sposób nadchodzące zdarzenia są przetwarzane w kolejności pojawiania się, a żadne z nich nie zostaje „zgubione”.

Za pobieranie komunikatów od systemu odpowiedzialna jest funkcja GetMessage(). Umieszcza ona uzyskany komunikat w strukturze specjalnego typu MSG, zawierającej między innymi cztery pola odpowiadające parametrom procedury zdarzeniowej. Adres tej struktury (u nas nazywa się ona msgKomunikat)

W polu deklaracji procedury WinMain dodajemy:

```
LOCAL msgKomunikat:MSG
```

Podajemy w pierwszym parametrze funkcji GetMessage(); pozostałe trzy parametry są w większości przypadków wypełniane zerami.

Wartość zwrócona przez GetMessage() jest także bardzo ważna, skoro używamy jej jako warunku pętli while - pętli komunikatów. Omawiana funkcja zwraca bowiem zero (co przerywa pętlę), gdy odebrany komunikat jest WM\_QUIT.

### **Procedura zdarzeniowa**

Funkcja nazywana jest: WindowEventProc. Nazwa procedury zdarzeniowej nie ma tak naprawdę żadnego znaczenia i może być obrana dowolnie - najlepiej z korzyścią dla programisty.

Najczęstszymi nazwami są aczkolwiek WindowProc, WndProc, EventProc czy MsgProc, jako że dobrze ilustrują one czynność, którą ta procedura wykonuje.

Do procedury zdarzeniowej trafiają informacje na temat wszelkich zaistniałych w systemie zdarzeń, które dotyczą „obsługiwanego” przez procedurę okna.

Zgodnie z zasadami modelu zdarzeniowego, gdy wystąpi jakaś potencjalnie interesująca sytuacja (np. kliknięcie myszą, przyciśnięcie klawisza), system operacyjny operacyjny wywołuje procedurę zdarzeniową i podaje jej przy tym właściwe dane o zaistniałym zdarzeniu.

Rolą programisty piszącego treść tej procedury jest odebranie owych danych i posłużenie się nimi w należyty sposób we własnej aplikacji.

### **Procedura zdarzeniowa WndProc**

```
WndProc proc hWnd:HWND, uMsg:UINT, wParam:WPARAM, lParam:LPARAM
```

Procedura jest wywoływana w momencie wystąpienia jakiegoś zdarzenia. Może to być np. ruch myszą, kliknięcie przycisku, zamknięcie systemu itp. Za słowem proc znajdują się zmienne (i ich typy), które "dostajemy" od systemu w momencie wywołania procedury.

hWnd jest to uchwyt naszego okna. Podobnie jak uchwyt instancji, uchwyt okna jest to liczba której Windows używa do identyfikacji okna. Jest ona różna dla każdego z nich. Zmienna uMsg identyfikuje zdarzenie, które spowodowało wywołanie procedury. Przyjmuje ona różną wartość w zależności od niego, np. w momencie kliknięcia przycisku uMsg jest równe WM\_COMMAND, a niszczenia okna - WM\_DESTROY. Pozwala ona odpowiednio zareagować programiście na zaistniałe zdarzenie.

wParam i lParam są to parametry dla uMsg, a ich wartość również zależy od zdarzenia np. w momencie kliknięcia myszą w oknie zawierają one aktualne współrzędne kursora.

```
.IF uMsg==WM_DESTROY
    invoke    PostQuitMessage,0
```

W momencie zniszczenia okna wyślij message WM\_QUIT, które spowoduje zamknięcie okna i wyjście z procedury WinMain.

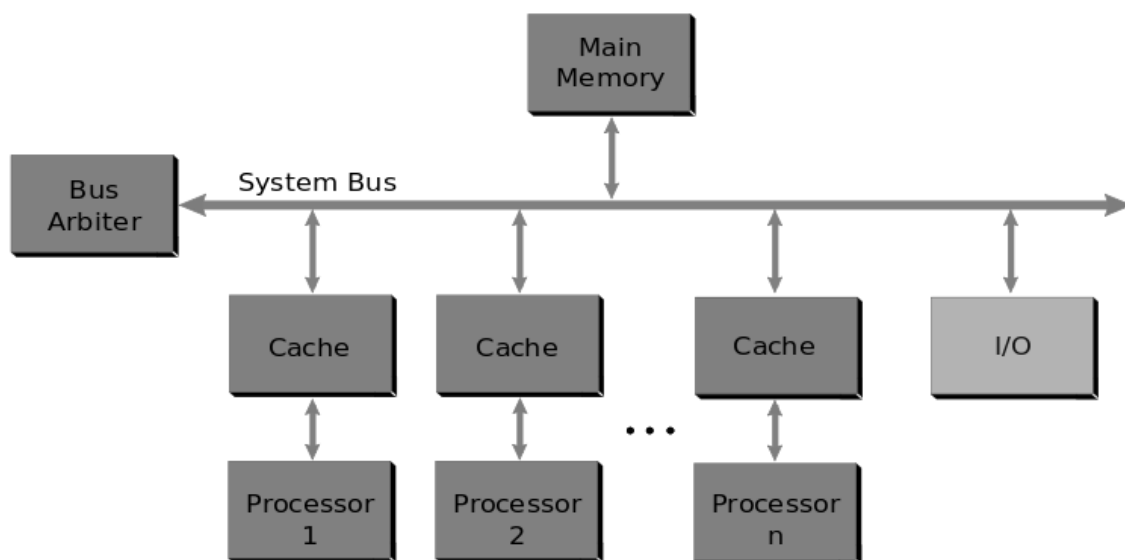
```
.ELSE
    invoke    DefWindowProc,hWnd,uMsg,wParam,lParam
    ret
.ENDIF
```

W innym przypadku wywołaj standardową procedurę obsługi okna.

### **Programowanie równoległe z wykorzystaniem pamięci współdzielonej i rozproszonej. Programowanie systemów wieloprocessorowych.**

Systemy wieloprocessorowe używają dwóch lub więcej jednostek centralnych (CPU) w ramach jednego systemu komputerowego. Termin odnosi się również do zdolności systemu do obsługi więcej niż jednego procesora lub do przydzielania zadań między nimi.

Wieloprocessorowy system komputerowy składa się z dwóch lub więcej jednostek przetwarzania (wielu procesorów), z których każda dzieli pamięć główną i urządzenia peryferyjne w celu jednoczesnego przetwarzania programów. Procesory mogą współdzielić część lub całość pamięci systemu i urządzeń we / wy.



Rys. Schemat systemu wieloprocessorowego



Na poziomie systemu operacyjnego termin ten używany jest w odniesieniu do wykonywania wielu współbieżnych procesów w systemie, przy czym każdy proces działa na osobnym procesorze lub rdzeniu, w przeciwieństwie do pojedynczego procesu w dowolnym momencie. Procesory mogą przy tym korzystać z obszarów pamięci współdzielonej bądź pamięci rozproszonej przyporządkowanej każdemu z nich. W przypadku użycia tej definicji systemy wieloprocessorowe sprowadzają się do wielozadaniowości, która może wykorzystywać tylko jeden procesor, ale przełączać go w przedziałach czasowych między zadaniami. Jednak przetwarzanie wieloprocessorowe oznacza prawdziwe równoległe wykonywanie wielu procesów przy użyciu więcej niż jednego procesora.

Można wyróżnić następujące sposoby wykonania programów:

- sekwencyjne– każda kolejna operacja wykonywana jest po zakończeniu poprzedniej
- równoległe– więcej niż jedna operacja wykonywana jest w tym samym czasie – wymaga więcej, niż jednego procesora
- przeplatane– wykonanie programu odbywa się fragmentami na przemian z fragmentami innych uruchomionych programów
- współbieżne– uogólnione pojęcie obejmujące zarówno wykonanie równoległe, jak i przeplatane

Rozkazy maszynowe wykonywane są przez procesor sekwencyjnie. Podczas wykonywania programu rozkazy maszynowe odwołują się do danych przechowywanych w pamięci głównej lub pomocniczej.

Sekwencyjnie wykonywany zbiór rozkazów nazywamy wątkiem. Przetwarzanie równoległe następuje wtedy, kiedy przynajmniej niektóre z rozkazów (wątków) podczas wykonywania programu realizowane są w tym samym czasie.

Przeanalizuj działanie programu w języku C:

```
#include<stdlib.h>
#include<stdio.h>
#include<unistd.h>
int zmienna_globalna=0;
main(){
int pid, wynik;
pid = fork();
if(pid==0){
zmienna_globalna++;
printf("Proces potomny: zmienna globalna = %d\n",
zmienna_globalna);
wynik = execl("/bin/ls", "ls", ".", (char *) 0);
if(wynik == -1) printf("Proces potomny nie wykonał polecenia
ls\n");
} else {
wait(NULL);
printf("Proces nadrzędny: proces potomny zakończył
działanie\n");
printf("Proces nadrzędny: zmienna globalna = %d\n",
zmienna_globalna);
}
```



}

Kod wykorzystuje procedurę *fork*. Po kompilacji kodu, uruchomienie programu związane jest z utworzeniem procesu, którego wątek główny, i w tym momencie jedyny, rozpoczyna realizację funkcji *main* kodu. Proces ten jest nazywany procesem nadrzędnym lub procesem macierzystym. W momencie wywołania funkcji *fork* system operacyjny tworzy nowy proces, proces potomny. W efekcie działania funkcji *fork* w systemie pojawiają się dwa procesy o różnych identyfikatorach, różnych przestrzeniach adresowych, realizujące ten sam kod i współdzielące szereg zasobów.



**Fundusze Europejskie**  
Wiedza Edukacja Rozwój



**Rzeczpospolita  
Polska**

**Unia Europejska**  
Europejski Fundusz Społeczny







Materiały zostały opracowane w ramach projektu  
*„Zintegrowany Program Rozwoju Politechniki Lubelskiej – część druga”*,  
umowa nr **POWR.03.05.00-00-Z060/18-00**  
w ramach Programu Operacyjnego Wiedza Edukacja Rozwój 2014-2020  
współfinansowanego ze środków Europejskiego Funduszu Społecznego