Kamil Skarżyński

Programowanie niskopoziomowe
Instrukcje arytmetyczne i logiczne
Laboratorium 04

1. Wprowadzenie

Podczas laboratoriów zapoznamy się z:
1. Procedurą konwertującą znaki ASCII do liczby,
2. Operacjami arytmetycznymi na liczbach ujemnych,
3. Procedurą konwertującą znaki ASCII do formy binarnej,
4. Operacjami przesunięć bitowych.

2. Zadania

1. Utwórz program umożliwiający wprowadzenie kolejno 4 liczb(do każdej oddzielna zachęta oraz zczytywanie) a następnie zapisz je odpowiednio do zmiennych. Przy zapisie do zmiennej wykorzystaj procedurę atoi w pliku atoi.asm (znajdziesz tam przykład jej wykorzystania) w celu konwersji znaków ascii do liczby (2 punkty).
2. Oblicz wartość funkcji dla 4 argumentów ze znakiem. Argumenty wprowadzaj pojedyńczo wyświetlając zachęte do wprowadzenia każdego z nich. Wynik zapisz do rejestru EAX (2 punkty),
3. Wyświetl wynik obliczeń w formacie : "Wynik obliczeń dla funkcji y= <wariant>to: <wynik>". Wykorzystaj procedurę wsprintfa (2 punkty).
4. Utwórz program korzystając z kodu dołączonym w pliku scanBin.asm. Oblicz wartość funkcji dla czterech argumentów logicznych według wzoru dla swojego zadania (wg tabeli wariantów). Argumenty wprowadzane są pojedynczo w postaci binarnej (ciąg 8 bitów w formie znaków ascii 0 lub 1)). Wyświetl zachęte do wprowadzenia każdego z ciągów. Wynik wyświetl również w postaci binarnej (2 punkty).
5. Napisz program pobierający liczbę wprowadzoną przez użytkownika (bez znaku), a następnie pomnoży ją przez liczbę określoną w tabeli wariantów i wynik wypisze na konsolę. Nie można korzystać z instrukcji MUL i IMUL, wykorzystaj przesunięcie bitowe (2 punkty).

3. Tabele wariantów

Tabela 1. Tabela wariantów dla zadania 2

Numer	Wariant	Numer	Wariant
1	(a+b)*c/d	16	a/b/(c+d)
2	a-b*c/d	17	(a+b)*c-d
3	a*(b+c)/d	18	a*(b+c)-d
4	a/(b+c)*d	19	a/(b+c-d)
5	(a+b)-c/d	20	a+(b-c)-d
6	(a-b*c)+d	21	a-b-c/d
7	a*(b-c)+d	22	a*(b-c-d)
8	a/b-c+d	23	a*(b-c)-d
9	(a+b)*(c+d)	24	a+b*c-d
10	(a-b)*c+d	25	(a-b)*(c-d)
11	a*b*(c+d)	26	a*b*c-d
12	a/(b*c)+d	27	a/b*c-d
13	a+b/c/d	28	a+b/c-d
14	(a-b)/c+d	29	a-b/c-d
15	a*b/c+d	30	a*b/c-d

Tabela 2. Tabela wariantów dla zadania 5

Numer	Wariant
1	5=(4+1)
2	6=(4+2)
3	7=(4+2+1)
4	9=(8+1)
5	10=(8+2)
6	11=(8+2+1)
7	12=(8+4)
8	13=(8+4+1)
9	14=(8+4+2)
10	17=(16+1)
11	18=(16+2)
12	19=(16+2+1)
13	20=(16+4)
14	21=(16+4+1)
15	22=(16+4+2)

Tabela 3. Tabela wariantów dla zadania 4

Numer	Wariant
1	(a OR b) AND c XOR d
2	a AND (b AND c) XOR d
3	a AND (b OR c) XOR d
4	a XOR (b OR c) AND d
5	(a OR b) AND (c XOR d)
6	(a AND b AND c) OR d
7	a AND (b AND c) OR d
8	a XOR b AND c OR d
9	(a OR b) AND (c OR d)
10	(a AND b) AND c OR d
11	a AND b AND (c OR d)
12	a XOR (b AND c) OR d
13	a OR b XOR c XOR d
14	(a AND b) XOR c OR d
15	a AND b XOR c OR d
16	a XOR b XOR (c OR d)
17	(a OR b) AND c AND d
18	(a AND b) AND c XOR d
19	a AND (b OR c) AND d
20	a XOR (b OR c AND d)
21	a OR (b AND c) AND d
22	a AND b AND c XOR d
23	a OR b AND c AND d
24	(a AND b) AND (c AND d)
25	a AND b AND c AND d
26	a OR b XOR c A d
27	a AND b XOR c AND d
28	a AND b XOR c AND d

4. Pomoc

4.1. GetStdHandle - pobranie uchwytu do konsoli

W celu uzyskania uchwytu do aktualnie uruchomionego okna konsoli możemy wykorzystać procedure GetStdHandle:

```
HANDLE WINAPI GetStdHandle(
        _In_ DWORD nStdHandle
);
```

nStdHandle może przyjmować następujące wartości:
1. -10 - uchwyt wejściowy, do odczytu z konsoli,
2. -11 - uchwyt wyjsciowy, do wypisywania znaków na konsolę,
3. -12 - uchwyt umożliwiający odczyt błędów.

Uchwyt zwracany jest do rejestru EAX po wykonaniu procedury.

Przykład wykorzystania:

```
push STD_OUTPUT_HANDLE ; stala -11
call GetStdHandle
mov outputHandle, EAX
```

4.2. WriteConsoleA - wypisywanie znaków na konsolę

Biblioteki Win32 udostępniają procedury umożliwiające odczyt znaków z interesującej nas konsoli. Można wykorzystać do tego procedurę WriteConsoleA o następującej sygnaturze:

```
BOOL WINAPI WriteConsoleA(
        _In_             HANDLE  hConsoleOutput,
        _In_       const VOID    *lpBuffer,
        _In_             DWORD   nNumberOfCharsToWrite,
        _Out_            LPDWORD lpNumberOfCharsWritten,
        _Reserved_       LPVOID  lpReserved
);
```

Parametry:

1. hConsoleOutput - **uchwyt wyjśćiowy** do konsoli, pobierany za pomocą procedury GetStdHandle (4.1),
2. *lpBuffer - adress tablicy przechowującej znaki do wypisania,
3. nNumerOfCharsToWrite - liczba znaków które zostaną wypisane z poprzednio podanego adresu,
4. lpNumberofCharsWriten - adres zmiennej typu DWORD do której procedura zapisze ilość faktycznie wypisanych znaków,
5. lpReserved - wstawiamy null czyli 0

Przykład wykorzystania:

```
push 0
push OFFSET nOfCharsWritten
push nOfCharsToWrite
push OFFSET charsToWrite
push outputHandle
call WriteConsoleA
```

Segment danych:

```
nOfCharsWritten DWORD 0
charsToWrite    BYTE "Wprowadz argument A",0
nOfCharsToWrite DWORD $ - charsToWrite
```

4.3. CharToOemA - konwersja znaków

Konsola systemu windows wykorzystuje tak naprawdę kodowanie OEM, więcej można znaleźć tutaj. W tym celu by poprawnie wyświetlić polskie znaki, musimy przekonwertować znaki ASCII na OEM za pomocą procedury CharToOemA:

```
BOOL WINAPI CharToOemA(
        _In_  LPCTSTR lpszSrc,
        _Out_ LPSTR   lpszDst
);
```

Parametry:
1. lpszSrc - adres tablicy z znakami którą chcemy przekonwertować,
2. lpszDst - adress tablicy do której chcemy zapisać przekonwertowane znaki,

Przykład wykorzystania:

```
push OFFSET charsToWrite
push OFFSET charsToWrite
call CharToOemA
```

Jako adres docelowy możemy wykorzystać naszą tablicę z znakami, ponieważ pierwotne kodowanie nie będzie nam potrzebne.

4.4. ReadConsoleA - odczytywanie znaków z konsoli

W celu wprowadzania znaków przez konsolę możemy wykorzystać procedure ReadConsoleA o następującej sygnaturze:

```
BOOL WINAPI ReadConsole(
        _In_     HANDLE  hConsoleInput,
        _Out_    LPVOID  lpBuffer,
        _In_     DWORD   nNumberOfCharsToRead,
        _Out_    LPDWORD lpNumberOfCharsRead,
        _In_opt_ LPVOID  pInputControl
);
```

Parametry:
1. hConsoleInput - **uchwyt wejściowy** do konsoli, pobierany za pomocą procedury GetStdHandle(4.1),
2. lpBuffer - adress tablicy do której zapisane zostaną odczytane znaki,
3. nNumberOfCharsToRead - liczba znaków która ma zostać odczytana z konsoli,
4. lpNumberOfCharsRead - adres do miejsca pamięci do którego zapisana zostanie liczba zczytanych znaków,
5. pInputControl - wstawiamy null czyli 0

Przykład wykorzystania:

```
push 0
push OFFSET nOfCharsRead
push 10
push OFFSET inputBuffer
push inputHandle
call ReadConsoleA

mov EBX, OFFSET inputBuffer
add EBX, nOfCharsRead
mov [EBX-2], BYTE PTR 0
```

W celu zrozumienia 3 ostatnich instrukcji wykorzystaj debugger i podejrzyj co zapisywane jest w inputBuffer. Na potrzeby procedury konwertującej znaki na liczbę, nasz ciąg znaków musi kończyć się nullem (00 w pamięci).

4.5. Konwersja znaków na liczbę

By skonwertować znaki na liczbę możemy wykorzystać poniższą procedurę, możemy ją dodać pod procedurą main(ta procedura nie obsługuje liczb ujemnych, wykorzystaj tą z pliku scanInt.asm):

```
atoi proc uses esi edx inputBuffAddr:DWORD
        mov esi, inputBuffAddr
        xor edx, edx
        .Repeat
        lodsb
        .Break .if !eax
        imul edx, edx, 10
        sub eax, "0"
        add edx, eax
        .Until 0
        mov EAX, EDX
        ret
atoi endp
```

Przekazujemy do niej jeden parametr w postaci adresu tablicy w której przechowywany jest ciąg znaków. Wynik zwracany jest przez rejestr EAX.

Przykład wykorzystania procedury w programie:

```
push offset inputBuffer
call atoi
mov varA, EAX
```

4.6. Konwersja liczby na znaki

By skonwertować liczbę na znak, możemy wykorzystać procedurę WsprintfA. Procedura ta podmienia znaki specjalne (np "%i") i wstawia na ich miejsce odpowiednio przekonwertowane argumenty.

```
int __cdecl wsprintf(
        _Out_ LPTSTR  lpOut,
        _In_  LPCTSTR lpFmt,
        _In_          ...
);
```

```
wsprintfA PROTO C :VARARG ;prototyp procedury w masm32
```

Procedura zwraca długość nowo powstałego ciągu znaków do rejestru EAX.
Parametry:
1. lpOut - adres bufora do którego zostaną zapisane znaki ASCII,
2. lpFmt - adress tablicy w której przechowywany jest format wiadomości,
3. ... - pozostałe argumenty zależne od formatu wiadomości.

Przykład wykorzystania:

```
.data
        varA                DWORD 0
        solutionText        BYTE "Wynik: %i",0
        solutionBuffer      BYTE 255 dup(0)
```

```
.code
        push varA
        push OFFSET solutionText
        push OFFSET     solutionBuffer
        call wsprintfA
        add     ESP, 12         ;czyszczenie stosu
        mov     rinp, EAX       ;zapamietanie dlugosci powstalego ciagu
```

Segment danych:

```
varA                    DWORD 0
solutionText    BYTE "Wynik:%i",0
solutionBuffer  BYTE 255 dup(0)
rinp                    DWORD 0
```

W związku z tym że w formatce występuje tylko 1 znacznik, podajemy tylko 1 dodatkowy agrument. Do łączenia ciągów znaków musielibyśmy wykorzystać znacznik "s" zamiast "i".

4.7. Operacje mnożenia

Do mnożenia możemy wykorzystać 2 instrukcje:

4.7.1. MUL

Instrukcja MUL(unsigned multiply) mnoży 8,16 lub 32 bitowy operand (operandem może być r-rejestr lub m-miejsce w pamięci) i zapisuje wynik odpowiednio do rejestru AX, EAX lub EDX:EAX:

```
MUL r/m8 ;operandem jest rejestr 8-bitowy rejestr lub 8-bitowe m
 ;wynik zapisywany do rejestru AX
MUL r/m16 ;16-bitowy rejestr lub 16-bitowa zmienna
 ;wynik zapisywany do rejestru EAX
MUL r/r32 ;32-bitowy rejestr lub zmienna
 ;wynik zapisywany do rejestrow EDX:EAX
```

Przykład wykorzystania:

```
MOV EAX, 10
MUL varA ;varA - 32-bitowa zmienna, o wartosci 2
 ;wynik zapisany do EDX = 0 EAX = 14 (h)
```

4.7.2. IMUL

Instrukcja IMUL(signed multiply) mnoży 8,16 lub 32 bitowy operand (operandem może być r-rejestr lub m-miejsce w pamięci) i zapisuje wynik odpowiednio do rejestru AX, EAX lub EDX:EAX:

```
IMUL r/m8 ;operandem jest rejestr 8-bitowy rejestr
 ;lub 8-bitowa zmienna
 ;wynik zapisywany do rejestru AX
IMUL r/m16 ;16-bitowy rejestr lub 16-bitowa zmienna
 ;wynik zapisywany do rejestrow DX:AX
IMUL r/r32 ;32-bitowy rejestr lub zmienna
 ;wynik zapisywany do rejestrow EDX:EAX
 ;mozemy rowniez wykorzystac:
IMUL r32,r32, imm32 ;zapisze wynik mnozenia 2 operandu i 3 operan
 ;do 1 operandu
```

Przykład wykorzystania:

```
MOV EAX, -10
IMUL varA ;varA - 32-bitowa zmienna, o wartosci 2
;wynik zapisany do EDX = FFFFFFFF EAX = FFFFFFEC (h)
```

4.8. Operacje dzielenia

Do dzielenia możemy wykorzystać:

4.8.1. DIV

Instrukcja DIV(unsigned divide) wykonuje dzielenie przez 8,16,32 bitowy operand. Operandem może być rejestr lub pamięć. Wynik zapisywany jest do rejestru EAX, reszta EDX.

```
DIV r/m8 ;dzieli AX, wynik w AL, reszta AH
DIV r/m16 ;dzieli DX:AX, wynik AX, reszta DX
DIV r/m32 ;dzieli EDX:EAX, wynik EAX, reszta EDX
```

4.8.2. IDIV

Składnia jak w przypadku DIV.

4.9. CDQ

W przypadku mnemoniku IDIV, wykorzystywane są rejestry EDX:EAX. Najstarszym bitem w takim zapisie jest bit w rejestrze EDX nie EAX. By wypełnić odpowiednio rejestr EDX (w zależności od nastarszego bitu w rejestrze EAX) możemy wykorzystać instrukcję CDQ.

4.10. Operatory bitowe

Dostępne są instrukcje umożliwiające operacje bitowe:

4.10.1. AND

Wynokuje operację koniunkcji logicznej dla każdego z bitów:

```
AND r8/m8, r8/m8/imm ;wynik zapisany do 1 operandu
AND r16/m16, r16/m16/imm ;wynik zapisany do 1 operandu
AND r32/m32, r32/m32/imm ;wynik zapisany do 1 operandu
```

Przykład użycia:

```
MOV zmA, 11111111b
AND zmA, 00000001b
;mnozenie logiczne dla kazdego z bitow
;wynik 00000001 (b) zapisany w zmA
```

4.10.2. OR

Wykonuje operacja alternatywy logicznej dla każdego z bitów:

```
OR r8/m8, r8/m8/imm ;wynik zapisany do 1 operandu
OR r16/m16, r16/m16/imm ;wynik zapisany do 1 operandu
OR r32/m32, r32/m32/imm ;wynik zapisany do 1 operandu
```

Przykład użycia:

```
MOV zmA, 01010111b
OR zmA, 10000001b
;alternatywa logiczna dla kazdego z bitow
;wynik 11010111 (b) zapisany w zmA
```

4.10.3. XOR

Wykonuje operacje alternatywy wykluczającej dla każdego z bitów:

```
XOR r8/m8, r8/m8/imm ;wynik zapisany do 1 operandu
XOR r16/m16, r16/m16/imm ;wynik zapisany do 1 operandu
XOR r32/m32, r32/m32/imm ;wynik zapisany do 1 operandu
```

Przykład użycia:

```
MOV zmA, 01010111b
XOR zmA, 10000001b
;alternatywa logiczna dla kazdego z bitow
;wynik 11010110 (b) zapisany w zmA
```

4.10.4. NOT

Wykonuje operacje negacji logicznej dla każdego z bitów:

```
XOR r8/m8 ;wynik zapisany do 1 operandu
XOR r16/m16 ;wynik zapisany do 1 operandu
XOR r32/m32 ;wynik zapisany do 1 operandu
```

Przykład użycia:

```
MOV zmA, 01010111b
NOT zmA
;negacja logiczna dla kazdego z bitow
;wynik 10101000 (b) zapisany w zmA
```

4.10.5. Przesunięcia bitowe SHL, SHR

Instrukcje przesunięcia bitowego umożliwiają mnożenie/dzielenie liczby po przez przesunięcie bitów w prawo lub lewo oraz uzupełnienie brakujących zerem:

```
SHL r8/m8, imm ;wynik zapisany do 1 operandu
SHL r16/m16, imm ;wynik zapisany do 1 operandu
SHL r32/m32, imm ;wynik zapisany do 1 operandu
```

4.10.6. SAL, SAR

Instrukcja SAR działa tak samo jak SHR, ale zachowuje stan najstarszego bitu. Instrukcja SAL działa tak samo jak SHL.

4.10.7. ROL, ROR

Instrukcja ROR działa jak SHR, lecz najmłodszy bit zamiast przepaść kopiowany jest na miejsce najstarszego i do flagi przeniesienia. Instrukcja ROL działa jak SHL, lecz najstarszy bit kopiowany jest na miejsce najłodszego i do flagi przeniesienia.