Programowanie niskopoziomowe

Ćwiczenia laboratoryjne w środowisku Visual Studio 2019

Labolatoria 2

„Odczytywanie danych, operacje na liczbach oraz pamięci”

1 Zadania

1.1 Wyświetlenie komunikatu

Wyświetl komunikaty na konsoli:

„Proszę wpisać zmienną A ”
„Proszę wpisać zmienną B ”
„Proszę wpisać zmienną C ”
„Proszę wpisać zmienną D ”

Wskazówki: GetStdHandle - pobranie uchwytu do konsoli
WriteConsoleA - wypisywanie znaków na konsolę
Jeśli zawiera polskie znaki CharToOemA - konwersja znaków

1.2 Wczytanie z konsoli

Wczytaj na konsolę zmienne A, B, C, D

Wskazówki: ReadConsoleA - odczytywanie znaków z konsoli
GetStdHandle - pobranie uchwytu do konsoli

1.3 Obliczenie wariantu

Oblicz swój wariant

Wskazówki : Warianty do zadania 1.3:
Operacje na liczbach:

1.4 Wyswietlenie na konsoli

Wyświetl wynik na konsoli

1.5 Podmiana danych w ciągu znaków

Plik lab02zad5 zmodyfikuj tak, aby co drugi znak wprowadzony przez użytkownika został podmieniony na spację a następnie wyświetlony. Np. „1234567890” na „1 3 5 7 9 „

2 Warianty do zadania 1.3: (wróc Zadania)

1	A+B-C*D
2	A+B-C/D
3	D-C/A/B
4	A*B+C-D
5	A+B+C/D
6	A*B-C/D
7	D-C/A*B
8	A+B+C+D/A
9	A+B+C+D*B
10	2*A+D-C/B
11	2*A*B-C/D
12	10/A/B/C+D
13	A*A-B/D+C
14	D+C*A/B
15	2*A*B-C/2*D

3 Wskazówki

3.1 Operacje na liczbach: (wróc Zadania)

ARITHMETIC				Flags								
Name	Comment	Code	Operation	O	D	I	T	S	Z	A	P	C
ADD	Add	ADD Dest,Source	Dest:=Dest+Source	±				±	±	±	±	±
ADC	Add with Carry	ADC Dest,Source	Dest:=Dest+Source+CF	±				±	±	±	±	±
SUB	Subtract	SUB Dest,Source	Dest:=Dest-Source	±				±	±	±	±	±
SBB	Subtract with borrow	SBB Dest,Source	Dest:=Dest-(Source+CF)	±				±	±	±	±	±
DIV	Divide (unsigned)	DIV Op	Op=byte: AL:=AX / Op AH:=Rest	?				?	?	?	?	?
DIV	Divide (unsigned)	DIV Op	Op=word: AX:=DX:AX / Op DX:=Rest	?				?	?	?	?	?
DIV 386	Divide (unsigned)	DIV Op	Op=doublew.: EAX:=EDX:EAX / Op EDX:=Rest	?				?	?	?	?	?
IDIV	Signed Integer Divide	IDIV Op	Op=byte: AL:=AX / Op AH:=Rest	?				?	?	?	?	?
IDIV	Signed Integer Divide	IDIV Op	Op=word: AX:=DX:AX / Op DX:=Rest	?				?	?	?	?	?
IDIV 386	Signed Integer Divide	IDIV Op	Op=doublew.: EAX:=EDX:EAX / Op EDX:=Rest	?				?	?	?	?	?
MUL	Multiply (unsigned)	MUL Op	Op=byte: AX:=AL*Op if AH=0 ♦	±				?	?	?	?	±
MUL	Multiply (unsigned)	MUL Op	Op=word: DX:AX:=AX*Op if DX=0 ♦	±				?	?	?	?	±
MUL 386	Multiply (unsigned)	MUL Op	Op=double: EDX:EAX:=EAX*Op if EDX=0 ♦	±				?	?	?	?	±
IMUL *i*	Signed Integer Multiply	IMUL Op	Op=byte: AX:=AL*Op if AL sufficient ♦	±				?	?	?	?	±
IMUL	Signed Integer Multiply	IMUL Op	Op=word: DX:AX:=AX*Op if AX sufficient ♦	±				?	?	?	?	±
IMUL 386	Signed Integer Multiply	IMUL Op	Op=double: EDX:EAX:=EAX*Op if EAX sufficient ♦	±				?	?	?	?	±
INC	Increment	INC Op	Op:=Op+1 (Carry not affected !)	±				±	±	±	±	
DEC	Decrement	DEC Op	Op:=Op-1 (Carry not affected !)	±				±	±	±	±	

Operacje na pamięci:

Przeniesienie znaku spacji do miejsca w pamięci o adresie podanym w rejestrze *EBX*.

Instrukcja *BYTE PTR* wskazuje ze zostanie podmieniony tylko 1 bajt.

```
mov BYTE PTR [EBX], 20h
```

Pętle:
W celu stworzenia pętli w assemblerze wykorzystujemy instrukcję ***loop***.

```
mov ECX, 10
petla:
    add EAX, 1
loop petla
```

Rejestr ***ECX*** odpowiedzialny jest za ilość powtórzeń pętli, za każdym razem, gdy wykonana zostanie instrukcja ***loop*** jego wartość zmniejszana jest o 1. Następnie sprawdzana jest wartość rejestru ***ECX***, jeśli nie wynosi 0 to skacze do etykiety.

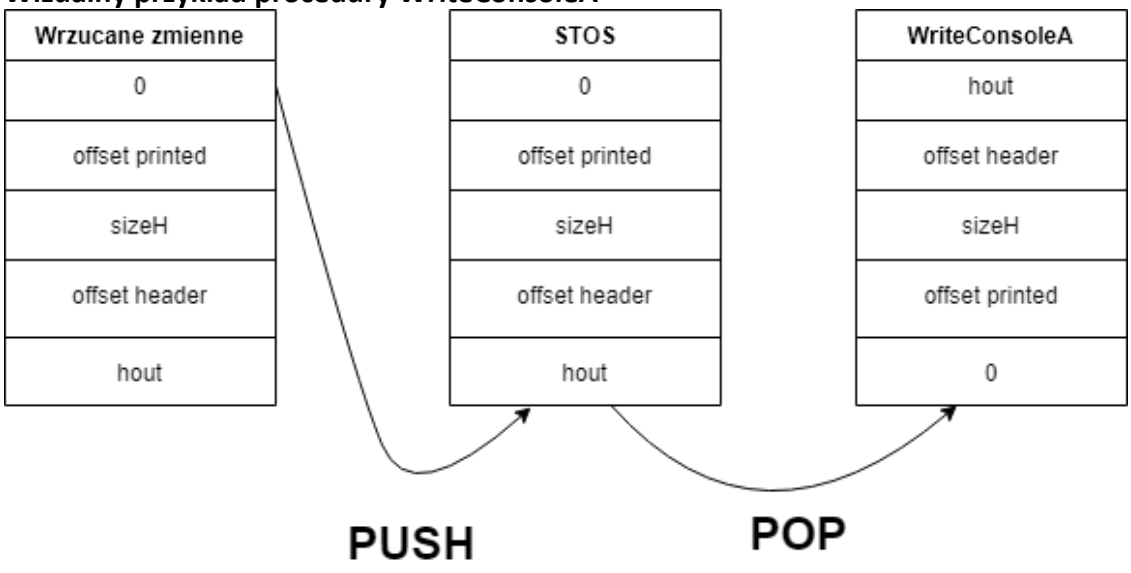
4 Procedury

Aby wywołać procedurę z paremetrem musimy użyć stosu (według konwencji STDCALL) czyli obszaru pamięci zorganizowanego w postaci listy **LIFO (Last In, First Out)**. Przy obsłudze stosu korzystamy z instrukcji **PUSH**, odkładającej dane na stos i z instrukcji **POP**, ściągającej dane ze stosu.
Rejestr ESP zawiera adres szczytu stosu. Jest on automatycznie modyfikowany przez instrukcje push i pop. Stos rośnie w "dół" tzn. odłożenie danych na stos zmniejsza ESP

Po wykonaniu procedury, według konwencji STDCALL stos powinien być wyczyszczony przez wywołaną procedurę.

```
push  0
push OFFSET printed
push sizeH
push OFFSET header
push hout
call WriteConsoleA
```

Wizualny przykład procedury WriteConsoleA



4.1 GetStdHandle - pobranie uchwytu do konsoli: (wróc Zadania)

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

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

Parametry:
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.

Segment danych:

```
STD_OUTPUT_HANDLE       equ -11
```

Fragment kodu:

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

4.2 WriteConsole

Wypisuje łańcuch znaków na ekran konsoli.

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

Parametry:

1. **hConsoleOutput** - uchwyt wyjściowy do konsoli, pobierany za pomocą procedury GetStdHandle

2. ***lpBuffer** - adress tablicy przechowującej znaki do wypisania,

3. **nNumerOfCharsToWrite** - liczba znaków które zostaną wypisane z po-przednio podanego adresu,

4. **lpNumberofCharsWriten** - adres zmiennej typu DWORD do której pro-cedura zapisze ilość faktycznie wypisanych znaków,

5. **lpReserved** - wstawiamy null czyli 0

Segment danych:

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

Fragment kodu:

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

4.3 ReadConsoleA - odczytywanie znaków z konsoli (wróc Zadania)

Odczytuje znaki wejściowe z bufora wejścia konsoli

```
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(3.4),
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

Segment danych:

```
nOfCharsRead  dword  0
inputBuffer   byte   128 dup(?)
```

Fragment kodu:

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

Dodatkowo!

Fragment kodu:

```
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 zna-ki na liczbę, nasz ciąg znaków musi kończyć się nullem (00 w pamięci).

4.4 CharToOemA - konwersja znaków (wróc Zadania)

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 zna-ki,

Fragment kodu:

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

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

4.5 wsprintfA – konkatenacja stringów (wróc Zadania)

Zapisuje sformatowane dane w określonym buforze. Wszystkie argumenty są konwertowane i kopiowane do bufora wyjściowego zgodnie z odpowiednią specyfikacją formatu

```
int WINAPIV wsprintfA(
        _Out_   LPSTR   lpOut,
        _In_    LPCTSTR ipFmt
);
```

Parametry:

1. **lpOut** - adres tablicy do którego zostaną zapisane znaki ASCII
2. **ipFmt** - adress tablicy w której przechowywany jest format wiadomości