Procedures

Covered

- Stack operations
  - PUSH,POP,PUSHFD,POPFD,PUSHAD,POPAD
  - Decrementing Stack → grows downward
- Defining and calling procedures
- What happens on call and return
- Parameter Passing
  - Through registers
  - Through stack → Today Lecture

Parameter passing via Registers

- Lets we make a procedure named SUM to add two variables.
- In Assembly lets two variables named v1 and v2 are defined in .DATA as DWORD
- Prototype of SUM is like this

  int SUM(int, int);

- In Assembly sum will be called as

  CALL SUM

- SUM adds two values in v1 and v2.how to pass values (parameters)? one way is through registers

Parameter passing via Registers

- Lets v1's values is passed in EAX and v2 in EBX, THEN before calling SUM, We will store values in Registers

```
MOV EAX, v1
MOV EBX, v2      → PASSING PARAMTERS in Registers
CALL SUM  ——————→ CALLING
```

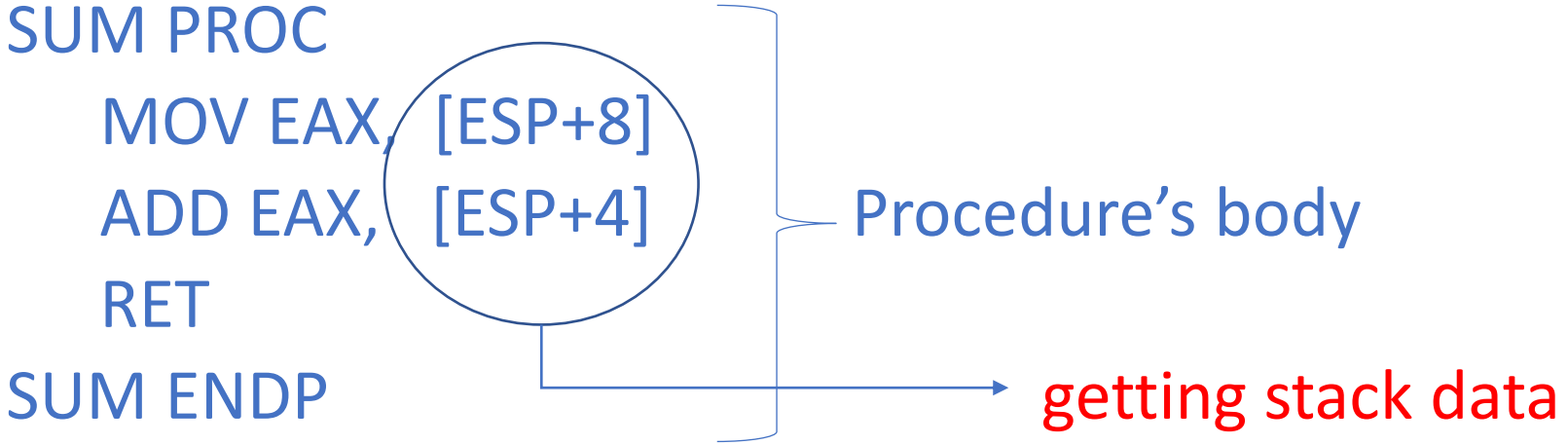
- Now SUM will get values from registers and add them, Body of SUM will be written like this

```
SUM PROC
    ADD EAX, EBX     Procedure's body
    RET
SUM ENDP
```

Parameter passing via Stack

- Lets v1's value is pushed on Stack and v2's value as well, THEN before calling SUM, We will store values in Stack

- Now SUM will get values from stack , can store them in registers and add them, Body of SUM will be written like this



Passing Parameters in Registers

```
;-----------------------------------------------
; ArraySum: Computes the sum of an array of integers
; Receives: ESI = pointer to an array of doublewords
;           ECX = number of array elements
; Returns:  EAX = sum
;-----------------------------------------------
ArraySum PROC
    mov eax,0                ; set the sum to zero
L1: add eax, [esi]           ; add each integer to sum
    add esi, 4               ; point to next integer
    loop L1                  ; repeat for array size
    ret
ArraySum ENDP
```

ESI: Reference parameter = array address

ECX: Value parameter = count of array elements

How to call?

Parameters passing in Registers + Proceudre calling

MOV ESI, OFFSET array
MOV ECX, LENGTHOF array
CALL Arraysum

Note:
ESI and ECX will be changed by the procedure. We want original values back when returning from a procedure.

Remember:
Stack use? Temporary storage
ESI and ECX original values can be pushed into stack and then procedure change it →no problem

Example on Preserving Registers

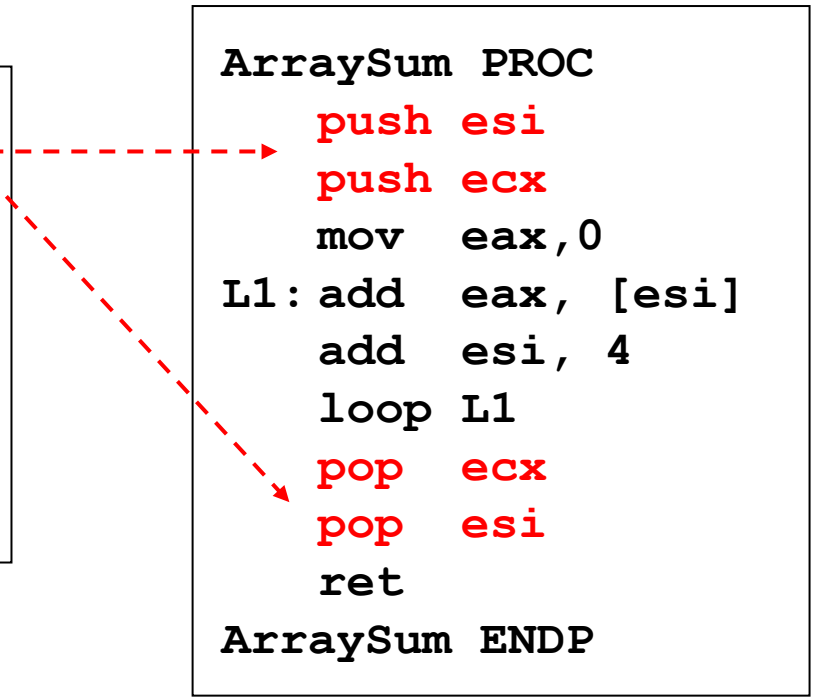
```
;-----------------------------------------------------
; ArraySum: Computes the sum of an array of integers
; Receives: ESI = pointer to an array of doublewords
;           ECX = number of array elements
; Returns:  EAX = sum
;-----------------------------------------------------
ArraySum PROC
    push esi                    ; save esi, it is modified
    push ecx                    ; save ecx, it is modified
    mov  eax,0                  ; set the sum to zero
L1: add  eax, [esi]             ; add each integer to sum
    add  esi, 4                 ; point to next integer
    loop L1                     ; repeat for array size
    pop  ecx                    ; restore registers
    pop  esi                    ; in reverse order
    ret
ArraySum ENDP
```

No need to save EAX. Why?

USES Operator

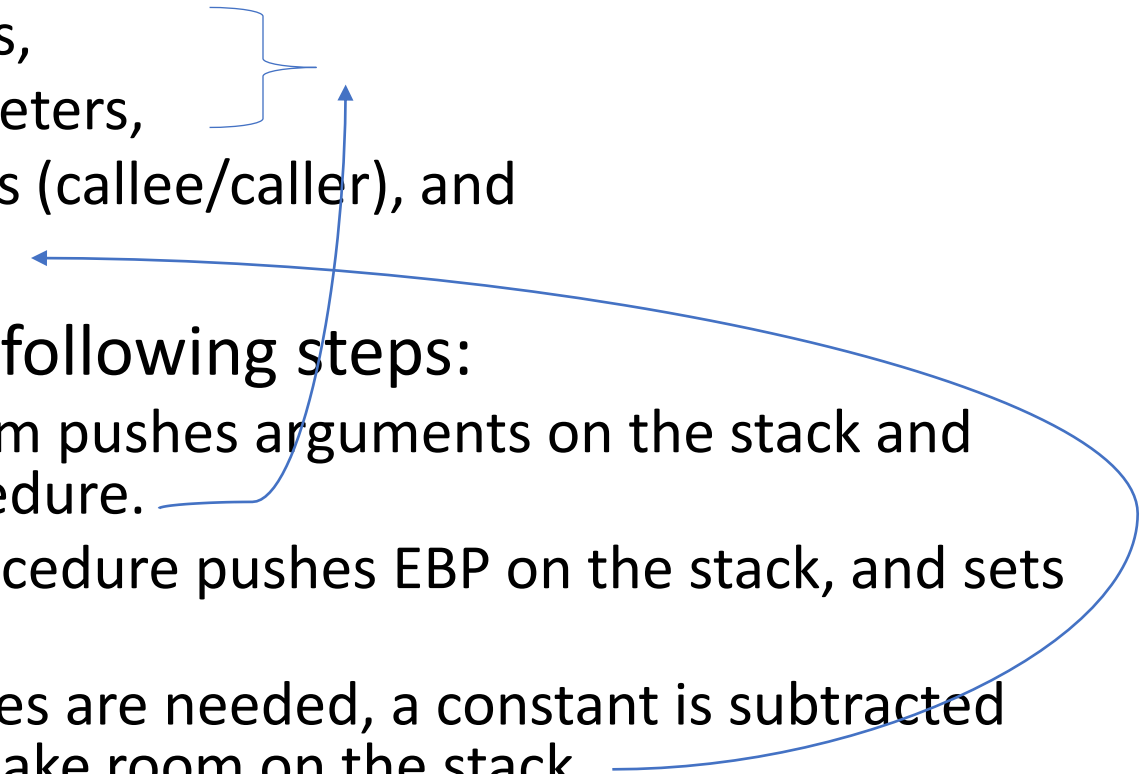
- The **USES** operator simplifies the writing of a procedure
  - Registers are frequently modified by procedures
  - Just list the registers that should be preserved after **USES**
  - Assembler will **generate** the **push** and **pop** instructions

```
ArraySum PROC USES esi ecx
    mov  eax,0
L1: add  eax, [esi]
    add  esi, 4
    loop L1
    ret
ArraySum ENDP
```



Stack Frame /Activation Record

- Area of the stack built when procedure is called.It stores
  - ✓return address,
  - ✓passed parameters,
  - ✓saved registers (callee/caller), and
  - ✓local variables
- Created by the following steps:
  - Calling program pushes arguments on the stack and calls the procedure.
  - The called procedure pushes EBP on the stack, and sets EBP to ESP.
  - If local variables are needed, a constant is subtracted from ESP to make room on the stack.



Passing parameters via registers vs. via stack

- Lets call a built in procedure called DumpMem which requires three arguments

```
pushad
mov esi,OFFSET array
mov ecx,LENGTHOF array
mov ebx,TYPE array
call DumpMem
Popad
```

Passed via registers

```
push TYPE array
push LENGTHOF array
push OFFSET array
call DumpMem
```

Passed via stack

Passing Arguments by Value vs. via address

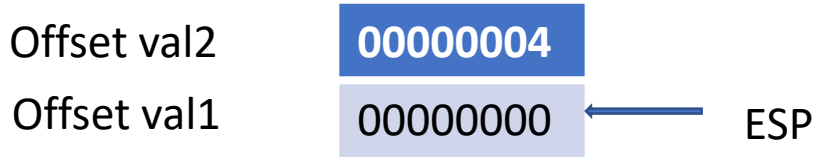
```
.data
val1   DWORD 5
val2   DWORD 6

.code
push val2
push val1
```

Stack prior to CALL

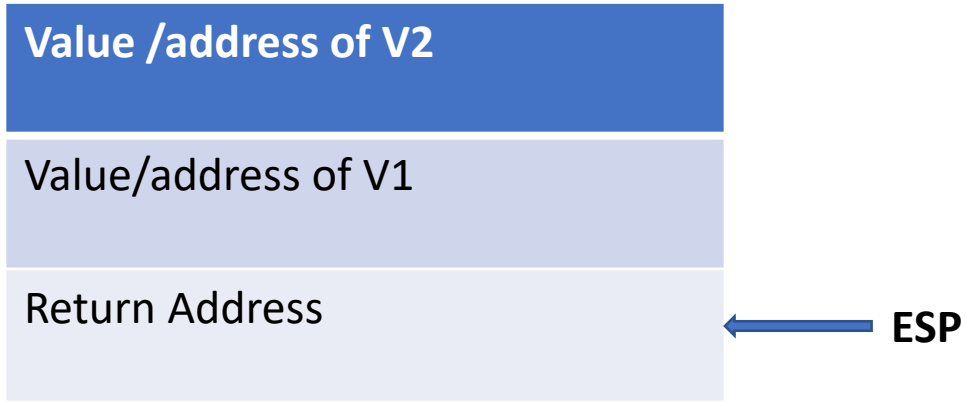
```
.data
val1   DWORD 5
val2   DWORD 6

.code
push OFFSET val2
push OFFSET val1
```



Stack prior to CALL

Stack after the CALL



How procedure will access these parameters from Stack?

Accessing Stack Parameters (C++/Java)

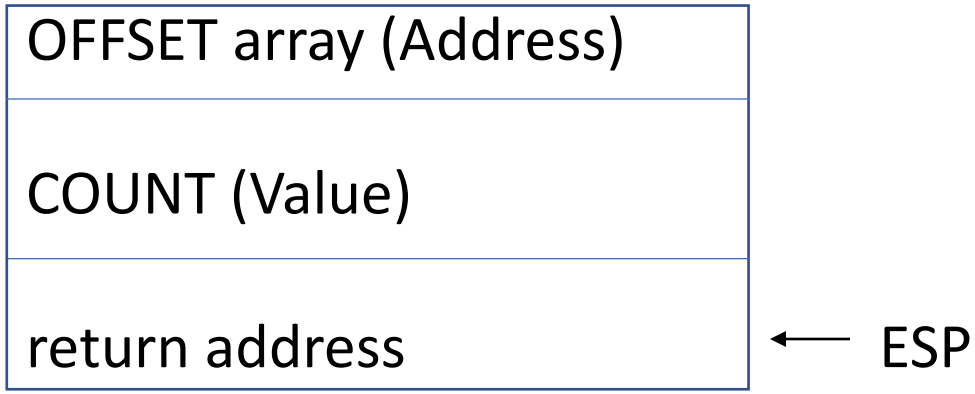
- Functions/methods access stack parameters using constant offsets from EBP.

  Example: [EBP + 8]

- EBP is called the base pointer or frame pointer because it holds the base address of the stack frame.
- EBP does not change value during the function.
- EBP must be restored to its original value when a function returns.

Example
Passing an Array via stack by Reference (1 of 2)

```
.data
count = 100
array WORD count DUP(?)
.code
   push OFFSET array
   push COUNT
   call ArrayFill
```



Passing an Array by Reference (2 of 2)

You have seen how to call ArrayFill prodecure.

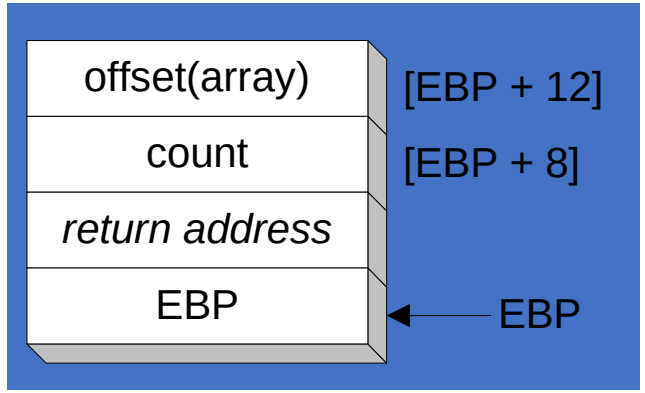
Now lets see how ArrayFill procedure access arguments pushed onto stack.

Procedure uses EBP to access arguments from stack?

First the procedure sets up EBP as shown in procedure code

```
ArrayFill PROC
    push EBP
    mov  EBP,ESP
    pushad  ;Preserving registers

    mov  esi,[ebp+12]
    mov  ecx,[ebp+8]
    .
    .
```



Every procedure you will see translated from HLL to assembly will be like the above

RET Instruction

- Syntax:
  - **RET**
  - **RET *n***

- Optional operand *n* causes *n* bytes to be added to the stack pointer (ESP) after EIP is loaded with Return Address.

Who removes parameters from the stack?

Caller (C) ..... or ..... Called-procedure (STDCALL):

```
                        AddTwo PROC
push val2                 push  ebp
push val1                mov   ebp,esp
call AddTwo               mov   eax,[ebp+12]
add   esp,8               add    eax,[ebp+8]

                          pop    ebp
                          ret     8
                        AddTwo ENDP
```

(Covered later: The MODEL directive specifies calling conventions)

Your turn . . .

- Create a procedure named Difference that subtracts the first argument from the second one. Following is a sample call:
  - push 14 ; first argument
  - push 30 ; second argument
  - call Difference ; EAX = 16

```
Difference PROC
    push ebp
    mov  ebp,esp
    mov  eax,[ebp + 8]      ; second argument
    sub  eax,[ebp + 12]     ; first argument
    pop  ebp
    ret  8
Difference ENDP
```

Another Example

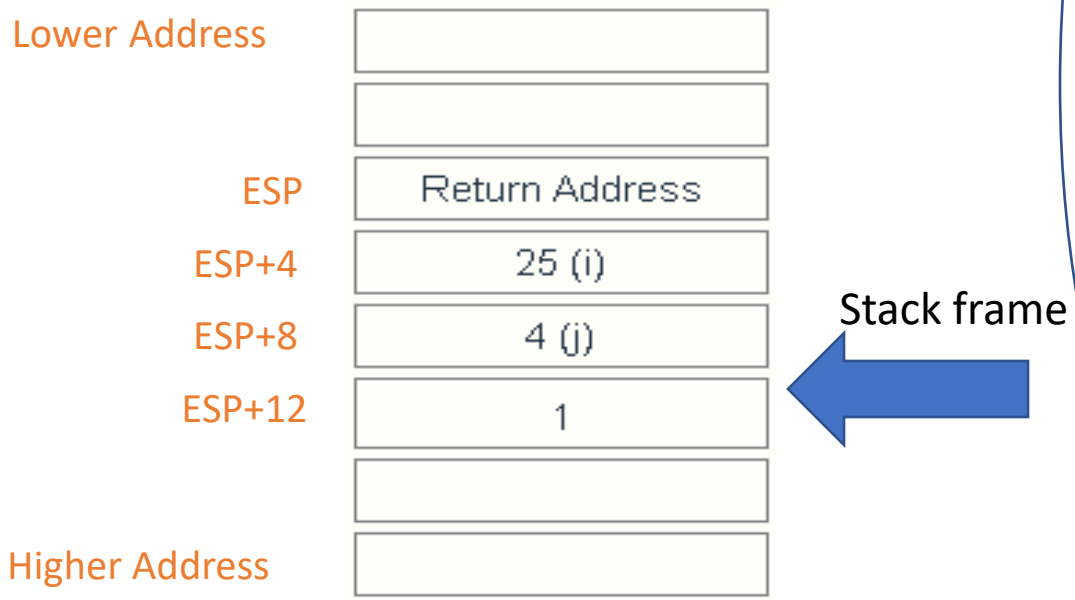
```
i = 25;
 j = 4;
Test(i, j, 1);
```

Compiler

```
mov i, 25
mov j, 4

push 1
push j
push i

call Test
```

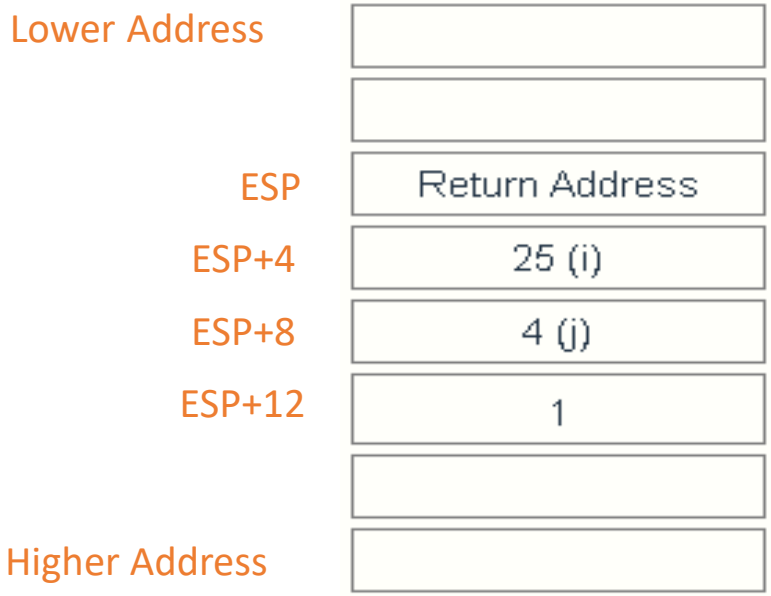


How Test procedure will access arguments from stack???

Parameter Passing Through Stack

Example: Accessing parameters on the stack

```
Test  PROC
   mov EAX, [ESP + 4] ;          get i
   add EAX, [ESP + 8] ;          add j
   sub EAX, [ESP + 12]
   ret
Test ENDP
```



Freeing Passed Parameters From Stack

- Use RET N instruction to free parameters from stack

```
Test  PROC
   mov AX, [ESP + 4] ;get i
   add AX, [ESP + 8] ;add j
   sub AX, [ESP + 12]
   ret 12
Test ENDP
```

Local Variables

- Local variables are dynamic data whose values must be preserved over the lifetime of the procedure, but not beyond its termination.
- At the termination of the procedure, the current environment disappears and the previous environment must be restored.
- Space for local variables can be reserved by subtracting the required number of bytes from ESP.
- Offsets from ESP are used to address local variables.

Local Variables

Pseudo-code (Java-like)	Assembly Language

void Test(int i){
  int k;
 	
  k = i+9;
  ……
}
	Test PROC
	  push EBP
	  mov EBP, ESP
	  sub ESP, 4
	  push EAX
	  mov DWORD PTR [EBP-4], 9
	  mov EAX, [EBP + 8]
	  add [EBP-4], EAX
	  ……
	  pop EAX
	  mov ESP, EBP
	  pop EBP
	  ret 4
	Test ENDP

Summary

```
void MyFunction3(int x, int y, int z)
{
  int a, int b, int c;
  ...
  return;
}
```

```
_MyFunction3:
  push ebp
  mov ebp, esp
  sub esp, 12 ; sizeof(a) + sizeof(b) + sizeof(c)
  ;x = [ebp + 8], y = [ebp + 12], z = [ebp + 16]
  ;a = [ebp - 4] = [esp + 8], b = [ebp - 8] = [esp + 4], c = [ebp - 12] = [esp]
  mov esp, ebp
  pop ebp
  ret 12 ; sizeof(x) + sizeof(y) + sizeof(z)
```

Built-in Procedures

To Get Input

ReadDec - Reads 32-bit unsigned decimal integer from keyboard

ReadInt - Reads 32-bit signed decimal integer from keyboard

ReadHex - Reads 32-bit hexadecimal integer from keyboard

ReadChar - Reads a single character from standard input

ReadString - Reads string from standard input, terminated by [Enter]

To Display on Screen

WriteDec - Writes unsigned 32-bit integer in decimal format

WriteInt - Writes signed 32-bit integer in decimal format

WriteHex - Writes an unsigned 32-bit integer in hexadecimal format

WriteBin - Writes unsigned 32-bit integer in ASCII binary format.

WriteBinB – Writes binary integer in byte, word, or doubleword format

WriteChar - Writes a single character to standard output

WriteString - Writes null-terminated string to console window

Example 1

Clear the screen, delay the program for 500 milliseconds, and dump the registers and flags.

```
.code
    call Clrscr
    mov  eax,500
    call Delay
    call DumpRegs
```

Sample output:

```
EAX=00000613 EBX=00000000 ECX=000000FF EDX=00000000
ESI=00000000 EDI=00000100 EBP=0000091E ESP=000000F6
EIP=00401026 EFL=00000286 CF=0 SF=1 ZF=0 OF=0
```

Example 2

Display a null-terminated string and move the cursor to the beginning of the next screen line.

```
.data
str1 BYTE "Assembly language is easy!",0

.code
   mov  edx,OFFSET str1
   call WriteString
   call Crlf
```

Example 2a

Display a null-terminated string and move the cursor to the beginning of the next screen line (use embedded CR/LF)

```
.data
str1 BYTE "Assembly language is easy!",0Dh,0Ah,0

.code
   mov  edx,OFFSET str1
   call WriteString
```

Example 3

Display an unsigned integer in binary, decimal, and hexadecimal, each on a separate line.

```
IntVal = 35
.code
    mov  eax,IntVal
    call WriteBin           ; display binary
    call Crlf
    call WriteDec           ; display decimal
    call Crlf
    call WriteHex           ; display hexadecimal
    call Crlf
```

Sample output:

```
0000 0000 0000 0000 0000 0000 0010 0011
35
23
```

Example 4

Input a string from the user. EDX points to the string and ECX specifies the maximum number of characters the user is permitted to enter.

```
.data
fileName BYTE 80 DUP(0)

.code
   mov edx,OFFSET fileName
   mov ecx,SIZEOF fileName – 1
   call ReadString
```

A null byte is automatically appended to the string.

Example 5

Generate and display ten pseudorandom signed integers in the range 0 – 99.
Pass each integer to WriteInt in EAX and display it on a separate line.

```
.code
    mov ecx,10                  ; loop counter

L1: mov  eax,100                ; ceiling value
    call RandomRange            ; generate random int
    call WriteInt               ; display signed int
    call Crlf                   ; goto next display line
    loop L1                     ; repeat loop
```

Example 6

Display a null-terminated string with yellow characters on a blue background.

```
.data
str1 BYTE "Color output is easy!",0

.code
   mov  eax,yellow + (blue * 16)
   call SetTextColor
   mov  edx,OFFSET str1
   call WriteString
   call Crlf
```

The background color is multiplied by 16 before being added to the foreground color.

Examples

```
mov dh, 24  ;row number
mov dl, 79   ;column number
call Gotoxy ; Move cursor there

mov al, '*'
call WriteChar ; Write '*' in bottom right

call ReadChar ; Character entered by user is in AL
```

```
; output a row of '&'s to the screen, minus first column
mov al, '&'
mov cx, 79
L1:         mov dh, 5 ; row 5
            mov dl, cl
            call Gotoxy
            call WriteChar
 loop L1
```

Thanks!