CS221
Miscellaneous x86 Instructions

Multiplication and Division – Chapter 7

The MUL instruction performs an unsigned multiplication between an 8, 16, or 32 bit operand with the AL, AX, or EAX register. The instruction formats allowed are:

```
MUL register
MUL memory
```

Notice there is no MUL instruction for an immediate value, so if you want to multiply by a constant then you must first MOV the constant to some other register or memory location.

MUL is a bit tricky because exactly what gets multiplied depends upon the operand. Here are three examples. Note that the product occupies more bits than the operands, and in some cases we modify the DX register:

```
MUL bl          ;  AX = AL * BL
MUL bx          ;  DX:AX = AX * BX
MUL ebx         ;  EDX:EAX = EAX * EBX
```

Be careful to note that the contents of the EDX register may change as a side effect to store the high bits of the multiplication. The Carry and Overflow flags are set if the upper half of the product is not zero.

The following multiplies AX * 160:

```
PUSH BX         ; Save values in these registers
PUSH DX
MOV BX, 160
MUL BX
POP DX          ; Restore values in these registers
POP BX          ; Ignore any possible high bits stored in DX
```

The MUL instruction performs unsigned multiplication. For signed multiplication, use the IMUL instruction. The results are identical.

Example:

```
mov AX, 5
mov BX, -1
IMUL BX         ; AX = -5 or FFFB
```

```
mov AX, 5
mov BX, -1
MUL BX              ; Treats BX as unsigned int, or FFFF = 65535
                    ; DX:AX = 4FFFB = 327,675 = 65535*5
```

Division operates in a manner similar to multiplication. DIV performs unsigned integer division on either a 8, 16, or 32 bit value. The quotient is stored in the AL/AX/EAX register while the remainder is stored in the AH, DX, or EDX register:

```
DIV bl              ; AL = AX / BL              AH = AX % BL
DIV bx              ; AX = (DX:AX) / BX         DX = (DX:AX) % BX
DIV ebx             ; EAX = (EDX:EAX) / EBX  EDX = (EDX:EAX) % EBX
```

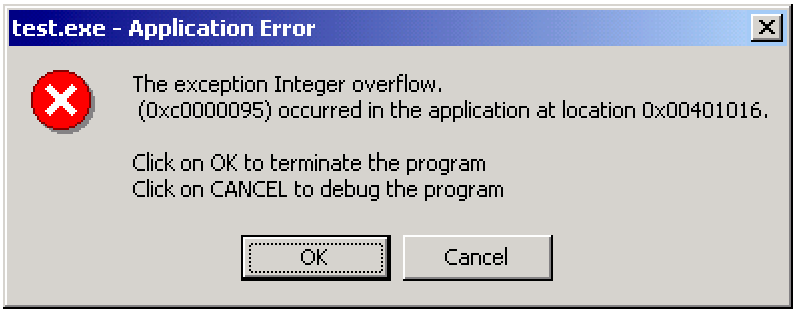
Here are some examples:

```
mov AX, 10
mov BL, 2
div BL              ; AH = 0, AL = 5

mov DX, 0
mov AX, 65535
mov BX, 4
div BX              ; DX = 3,  AX = 16383
```

Do you see a problem here?

```
mov AX, 0FFFFh
mov BL, 1
div BL              ; AH = 0?,  AL = ?
```

In this last case, we try to set AL = AX/BL. But AX/1 does not fit into AL. This causes a division overflow error that can crash the program.



test.exe - Application Error

The exception Integer overflow.
(0xc0000095) occurred in the application at location 0x00401016.

Click on OK to terminate the program
Click on CANCEL to debug the program

The IDIV instruction operates in a manner similar to DIV, except it performs a signed integer division instead of unsigned.

Local Variables – Chapter 8

In the preceding lectures we have declared all variables in the data segment. These are considered static global variables. They are global because they are accessible from anywhere in the program. They are static because the variables "live" throughout the lifetime of the entire program.

In contrast, local variables are:

- Accessible only by the procedure they are defined in
- Variable "dies" when the procedure exits
- More efficient use of memory than global since the storage space can be released and made available for new variables
- Same variable name can appear in multiple procedures without a name conflict

As we have seen before when discussing the theory with the Null & Lobur textbook, local variables are created on the runtime stack.

To declare local variables in MASM, use the LOCAL directive:

```
LOCAL varname:vartype       [, varname:vartype]
```

For arrays, use: LOCAL varname[SIZE]:varitype

Here are some examples:

```
MySub proc
        LOCAL var1:BYTE                         ; one byte
        LOCAL var2:WORD, var3: DWORD            ; a word, a dword
        LOCAL tempArray[10]:DWORD               ; Array of 10 dword's
        …
        ret
MySub endp
```

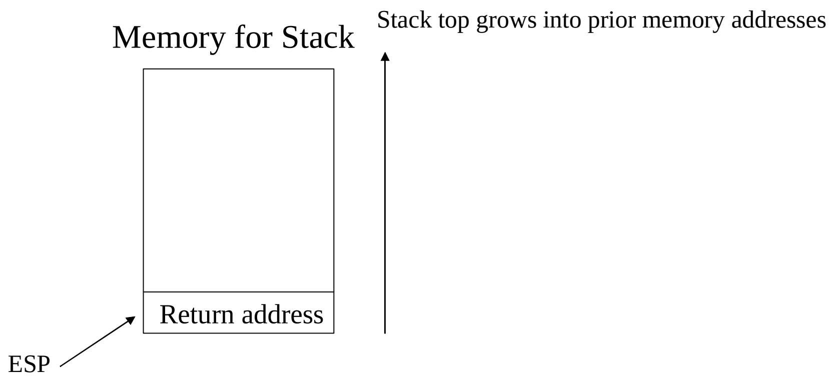
What does the assembler actually generate for the directive? Here is a code example and the corresponding disassembly taken from Visual Studio's debugger:

```
myproc proc
        LOCAL var1:WORD
        LOCAL var2:WORD

        ; Body of code here

        ret
myproc endp
```

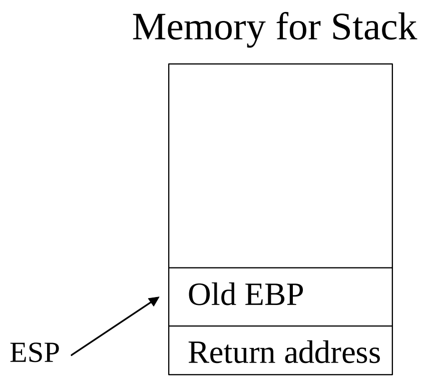
Here is the disassembly:

```
myproc proc
        push ebp
        mov ebp, esp
        add esp, 0FCh                ; Add -4 to ESP

        ; Body of code

        mov esp, ebp
        pop ebp                      ; Restore original ESP, EBP
        ret
```

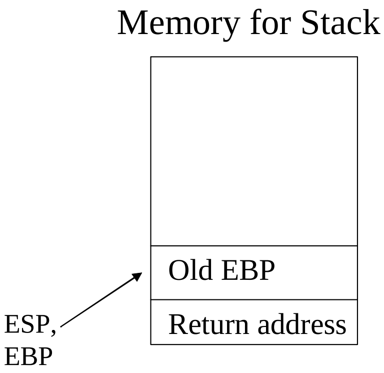
The ADD instruction adds -4 to ESP, moving it downward and creating an opening in the stack between ESP and EBP to store two local variables of size word as illustrated in the following diagrams. When we first enter the procedure, the stack looks something like the following, with the return address of the caller placed on the stack:



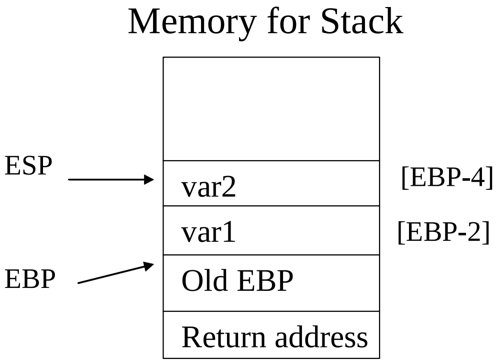
The first thing we do is push the existing EBP value on the stack, which decrements the ESP by the size of the EBP register:



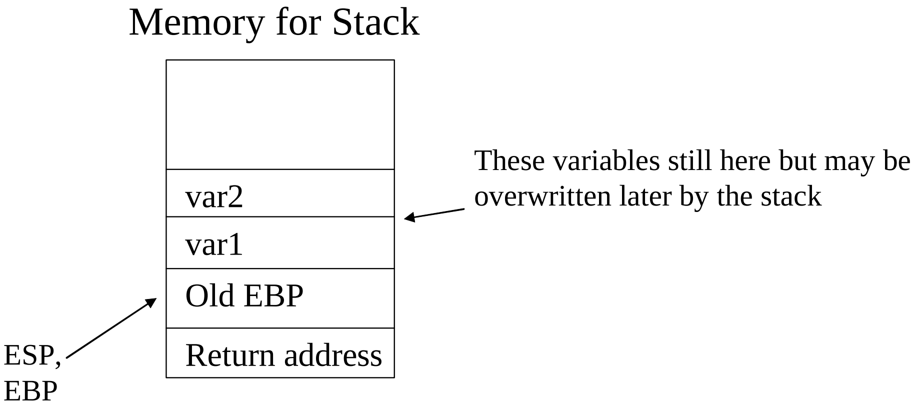
Next we copy the value of ESP into EBP, which makes EBP point to the same thing as ESP:



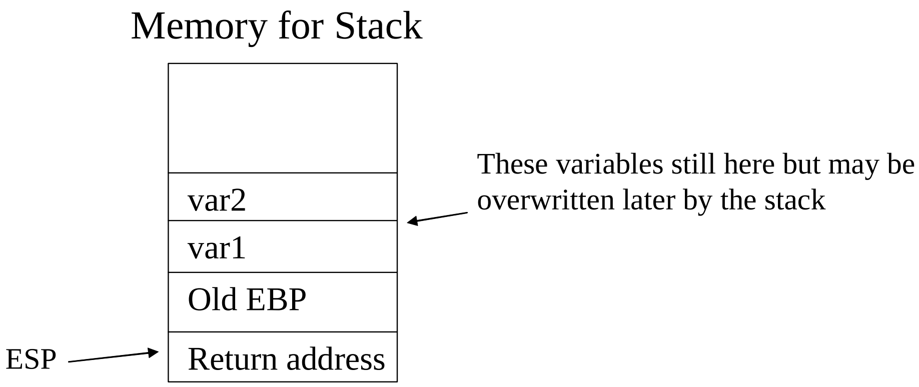
Next we add -4 to ESP, which is a large enough of an offset to store two words. The local variables of size word. The variables var1 and var2 are stored here:



We can now refer to the local variables via [EBP-2], [EBP-4], or even via absolute address if we wish. When the procedure exits, the first thing we do is move EBP to ESP, which has the effect of popping off all the local variables we allocated:


Memory for Stack
var2
These variables still here but may be overwritten later by the stack
var1
Old EBP
ESP,
EBP
Return address

Then we POP EBP, which restores into EBP the original value that it contained:


Memory for Stack
These variables still here but may be overwritten later by the stack
var2
var1
Old EBP
ESP
Return address

Finally, we execute the RET instruction, which pops off the return address into EIP so that we continue execution where we left off.

Within a procedure we can use stack-created variables much as we would use global variables. One exception is to get the offset of a variable. We can no longer use the OFFSET directive, as this only applies to global variables. OFFSET returns the distance from the start of the data segment, which is constant and known at assembly time. However, the address of an operand created on the stack could be anywhere in memory. We can still get its address though, using the LEA instruction, Load Effective Address:

Format:

LEA reg, memoperand

Loads the address of memoperand into the register reg.

Example:

```
include irvine32.inc

.code
main proc
      call myproc
      exit
main endp

myproc proc
      LOCAL mystring[50]:BYTE

      lea edx, mystring ; offset of mystring copied to EDX
      mov ecx, 50
      call readstring         ; Invoke Irvine readstring method
      call crlf
      call writestring  ; Output string from mystring

      mov ecx, 49
      lea edx, mystring
      mov al, 'Z'
L:    mov [edx],al            ; Fill mystring with Z's
      inc edx
      loop L
      mov al, 0
      mov [edx], al           ; Null at end
      call crlf
      lea edx, mystring ; Restore EDX to beginning of mystring data
area
      call writestring  ; Output string

      ret
myproc endp

end main
```

This example waits for the user to input data, outputs it, then fills the buffer with Z’s and outputs it again. Note that in the above example, I copied ‘Z’ into AL and then moved AL to [EDX]. This tells the assembler to only move one byte worth of data. I could also have used the BYTE PTR directive instead, as illustrated in the following equivalent code:

```
L:    mov [edx],byte ptr 'Z'  ; Fill mystring with Z's
      inc edx
      loop L
      mov [edx], byte ptr 0   ; Null at end
```