Assembler Review

COMP375

“Do you program in Assembly ?” she asked.

“NOP”, he said.

Intel Pentium has 8 User Registers

EAX	General use, division only in EAX
EBX	General use
ECX	General use
EDX	General use
EBP	Base pointer
ESI	Source pointer
EDI	Destination pointer
ESP	Stack pointer

Load and Store

- A **Load** instruction copies a value from memory into a register (Reads memory)
- A **Store** instruction copies a value from a register into memory (Writes memory)
- The Intel assembler is confusing because it uses the same mnemonic **`mov`** for both load and store

`mov` Instruction

- The **`mov`** instruction moves data between memory and a register or between two registers
- The format is

```
mov  destination, source
```

- where destination and source can be
  - register, memory to load data into a register
  - memory, register to store data into memory
  - register, register to move data between regs

Assignment Statements

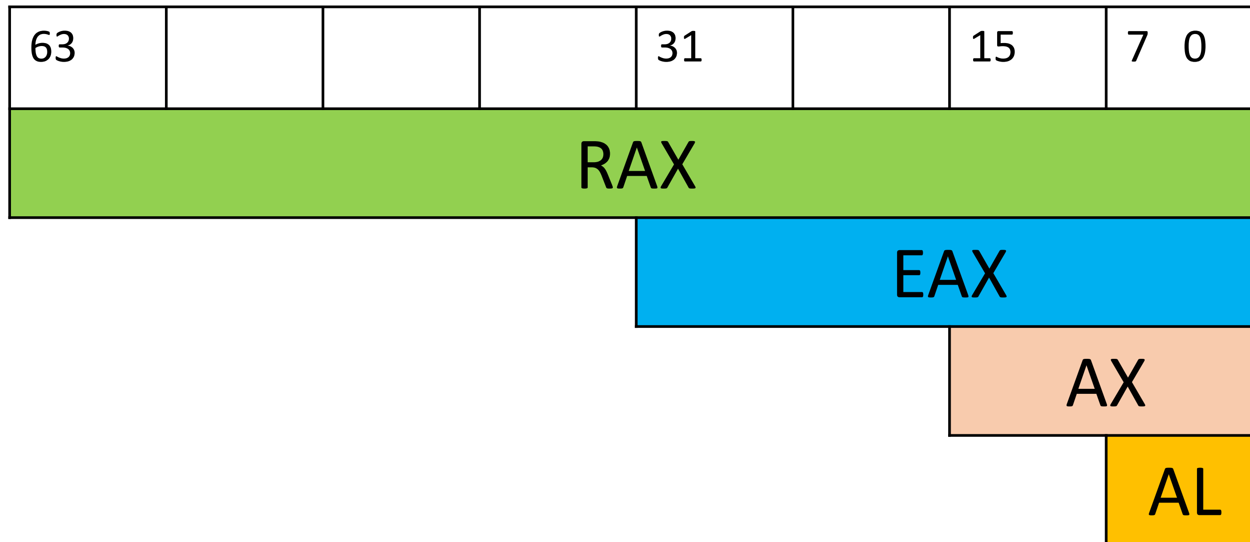
```
int    cat=3,  dog=5;
short bird=2, worm=7;
char  cow=41, goat=75;    //note: char is one byte integer
_asm {
    mov  eax, cat          ; dog = cat
    mov  dog, eax

    mov  cx, bird          // worm = bird
    mov  worm, cx

    mov  bl, goat          /* cow = goat */
    mov  cow, bl
}
```

Intel Registers

- The registers can hold data of different sizes



Arithmetic

- All arithmetic and logical functions (AND, OR, XOR, etc.) appear to be done in the registers
- Each instruction has one operand in a register and the other in memory or another register

```
add   eax, dog
```

- The result is saved in the first register

Arithmetic and Logical Instructions

mnemonic	operation
ADD	Add
SUB	Subtract
MUL	Unsigned Multiply
IMUL	Signed Multiply
DIV	Unsigned Divide
IDIV	Signed Divide
AND	Logical AND
OR	Logical OR

Arithmetic Example

```
int dog=3, cat=4, bird=5,
 cow;
_asm {    // cow = dog + cat - bird;
    mov   eax,dog
    add   eax,cat
    sub   eax,bird
    mov   cow,eax
}
```

What value is in EAX at the end?

```
int dog=4, cat=3,
bird=5;
_asm {
    mov  eax,dog
    sub  eax,cat
    mov  bird,eax
}
```

A. 1
B. 2
C. 3
D. 4
E. 5

Try it

- Complete this program to compute mouse = rat + shrew + mole;

```
int  mouse, rat, shrew, mole;
cin >>rat>>shrew>>mole;  // read data
_asm{
        // set mouse to the sum of rat, shrew and mole
}
cout << mouse << endl;  // print result
```

Big Operands

- Multiplication and Division use two registers to store a 64 bit value.
- A number is stored in EDX:EAX with the most significant bits in the EDX register and the least significant bits in EAX.

EDX	EAX
bits 63, 62, … 33, 32	bits 31, 30 … 2, 1, 0

Multiplication

- The imul signed multiply instruction has three forms.
- Multiply memory * EAX putting the 64 bit result in EDX:EAX

```
imul    memory
```

- Multiply memory * register putting the 32 bit result back in the register

```
imul    reg, memory
```

Division

- The 64 bit number in the EDX:EAX pair of registers is divided by the 32 bit value in a memory location or another register
- The resulting quotient is stored in EAX
- The resulting remainder is stored in EDX
- Since the EDX:EAX registers are always used, you do not have to specify them

```
idiv   memoryAddr
```

Multiply then Divide

- When you multiply a value in the EAX register, it creates a result in EDX:EAX
- idiv divides a value in the EDX:EAX pair

```
mov   eax, cow
imul  dog    EDX:EAX contains cow*dog
idiv  cat    divides EDX:EAX
```

What is in dog at the end?

```
char  dog = 1;
mov   BL, 5
mov   EBX, 3
mov   BL, dog
```

A. 1
B. 2
C. 3
D. 5
E. none of the above

If statements

- The high level language IF statement is easily implemented by a conditional jump

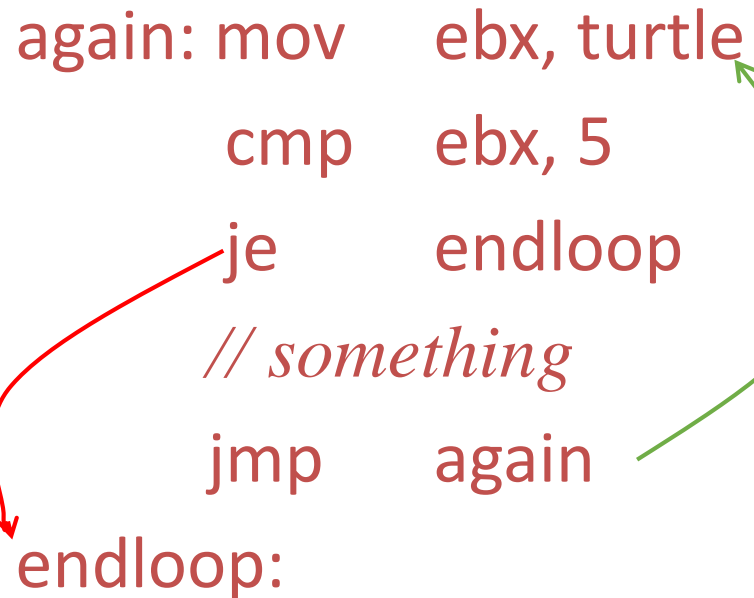
```
if (cat >= 13)
   cow = goat;
else
   bull = 47;
```

```
        MOV     eax, cat
        CMP     eax,13
        JGE     tiger
        MOV     bull, 47
        JMP     after      don't forget
tiger:  MOV  ebx, goat
        MOV  cow, ebx
after:
```

While Loops

- Loops are implemented with conditional jumps

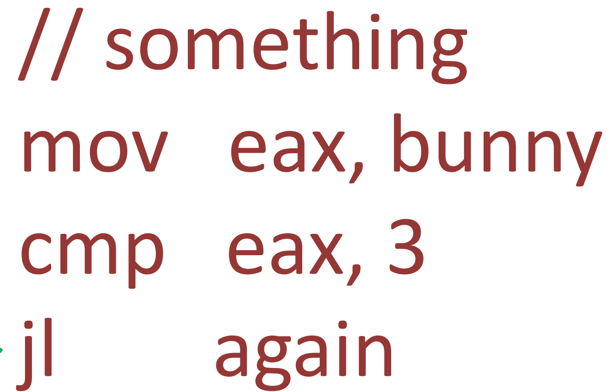
```
while (turtle != 5) {
      // something
}
```



Do While Loops

```
do {
    // something
} while (bunny < 3);
```

again:



- Loops involve a jump back

Jump to Next Line

```
       mov    ebx, dog
       cmp    ebx, cow
       jg     here
here:  mov    ebx, goat
```

- What happens if dog ≤ cow?
- What happens if dog > cow?

Intel Call instructions

- The **`CALL`** instruction pushes the return address on the stack and branches to the specified location

- The **`RET`** or return instruction pops the value from the top of the stack and jumps to that address

Example Function Call

- Consider the function

```
void thefunc(Widget b, int a ){
    int r = a;
}
```

- that is called by the main program

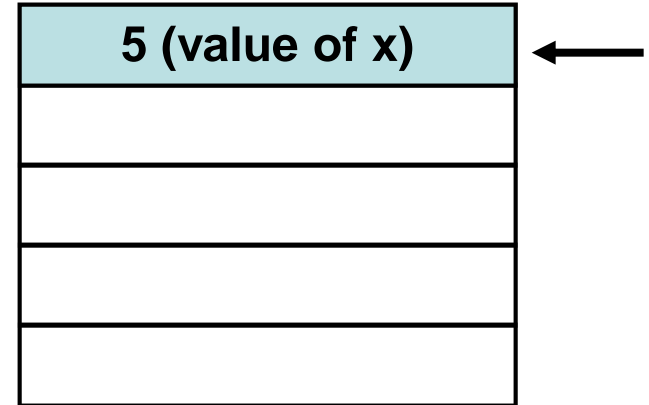
```
int x = 5;
Widget y = new Widget();
thefunc( y, x );
```

- **The Widget y is passed by reference. The int x is passed by value.**

```
thefunc( y, x );
```

Stack for Call

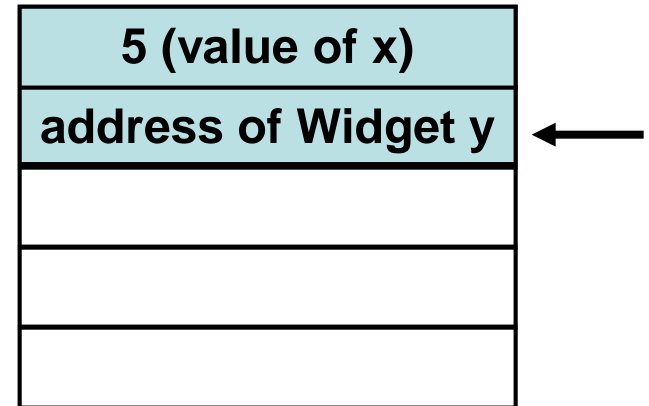
- push x



```
thefunc( y, x );
```

Stack for Call

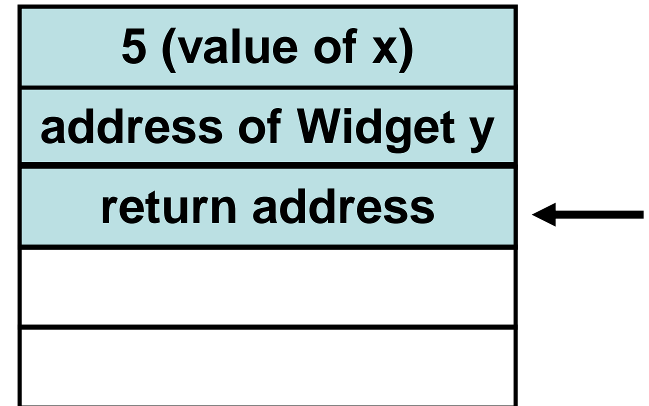
- push x
- push address of y



```
thefunc( y, x );
```

Stack for Call

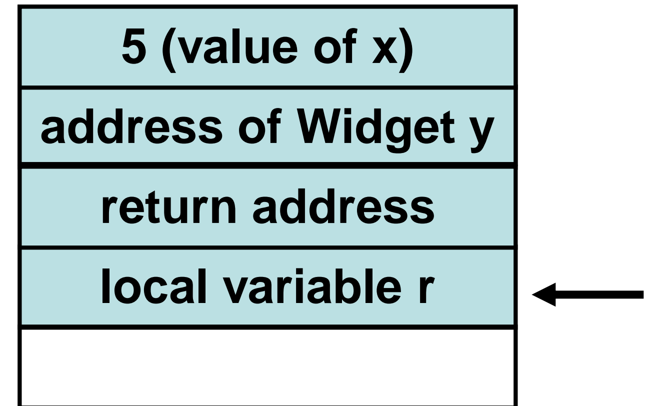
- push x
- push address of y
- call thefunc



```
thefunc( y, x );
```

Stack Use by Function

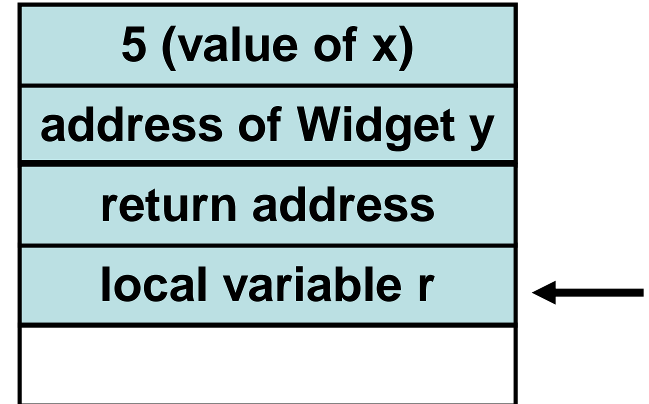
- push   x
- push address of y
- call **`thefunc`**
- increment stack



```
thefunc( y, x );
```

Stack for Call

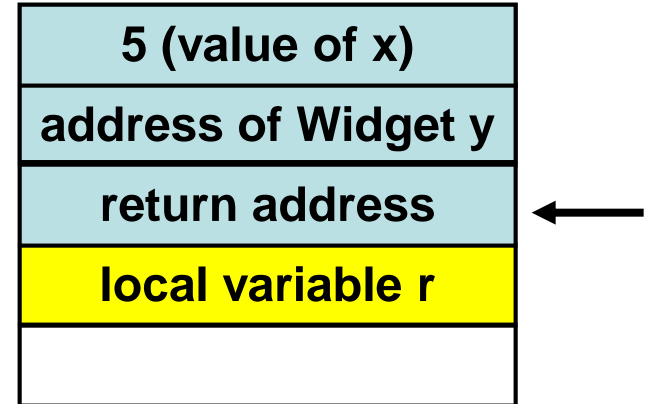
- push x
- push address of y
- call **`thefunc`**
- increment stack
- mov eax,12[esp] // param a



```
thefunc( y, x );
```

Stack for Return

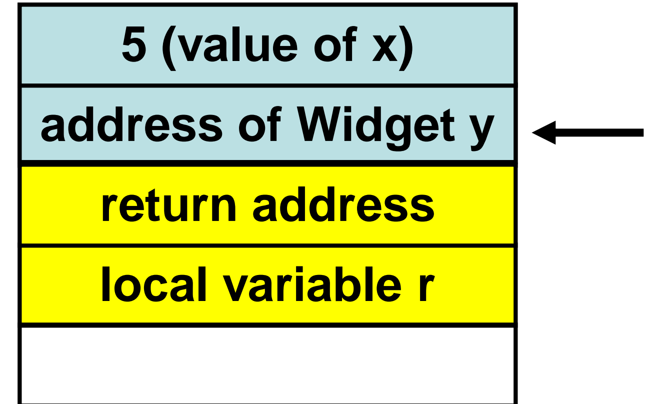
- push x
- push address of y
- call **`thefunc`**
- increment stack
- mov eax,12[esp] // param a
- decrement stack



```
thefunc( y, x );
```

Stack for Return

- push x
- push address of y
- call `thefunc`
- increment stack
- mov eax,12[esp] // param a
- decrement stack
- return



```
thefunc( y, x );
```

Cleanup Stack

- push   x
- push address of y
- call **`thefunc`**
- increment stack
- mov eax,12[esp] // param a
- decrement stack
- return
- decrement stack by 2