More Assembler Language

COMP375

Computer Architecture and Organization

Assembler Programming

"A programming language is low level when its programs require attention to the irrelevant." - Alan J. Perlis

"You can do everything in assembler, but no one wants to program in assembler anymore." - Yukihiro Matsumoto

Goals for Today

- Introduce assembler language constructs to:
  - Compute integer equations
  - If statements
  - loops
- Write simple assembler programs.

Intel Registers

- The Intel Pentium has eight 32-bit general-purpose registers

General-Purpose Registers

31–16	15–8	7–0	**16-bit**	**32-bit**
	AH	AL	AX	EAX
	BH	BL	BX	EBX
	CH	CL	CX	ECX
	DH	DL	DX	EDX
	BP			EBP
	SI			ESI
	DI			EDI
	SP			ESP

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

Constants

- Assembler programs can also use small constants.
- If you want to move the number 47 to eax

```
mov  eax,47
```

- Constants can also be characters

```
mov  al,'Z'
```

- You can also move constants to memory

```
mov  aardvark,15
```

Memory Model

- Memory is a huge one dimensional array of bytes.
- Both data and instructions are stored in memory.
- Registers can hold copies of the data or the address of the data in memory.

Hardware Data Types

- The hardware provides only a few primitive data types
  - long, int and short (8, 4 & 2 bytes)
  - float and double (4 & 8 bytes)
  - char or byte (1 byte)
- Integer data types can be signed or unsigned

Software Data Types

- All other data types are created by software
  - strings
  - objects
  - boolean
  - multi-dimensional arrays

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;
_asm {   // bird = dog + cat;
    mov   eax,dog
    add   eax,cat
    mov   bird,eax
}
```

Arithmetic Example 2

```
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
}
```

asm1-2

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
}
```

1. 1
2. 2
3. 3
4. 4
5. 5

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, save in EDX:EAX

```
imul    memorreg
```

- Multiply memory * register, save in register

```
imul    reg, memorreg
```

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
```

Arithmetic Example 2

```
int dog=9, cat=4, bird=5, cow;
_asm {   // cow = dog * cat / bird;
    mov   eax,dog   ; move dog to eax reg
    imul  cat       ; edx:eax = dog*cat
    idiv  bird      ; eax = dog*cat/bird
    mov   cow,eax   ; save result in cow
}
```

Arithmetic Example 3

```
int dog=9, cat=4, bird=5, cow;
_asm {   // cow = dog % cat + bird;
    mov   eax,dog    ; move dog to eax reg
    mov   edx,0      ; clear EDX
    idiv  cat        ; edx = dog % cat
    add   edx,bird   ; add bird to dog % cat
    mov   cow,edx    ; save result in cow
}
```

Shifting Bits

- You can move the bits in a register right or left.

00000101	Left shift 2	00010100
11111011	Right shift 1	01111101
11111011	Arith R shift 1	11111101

Shifts

- The SHR and SHL instructions shift the bits right or left by the specified number of bits.
- The SAR and SAL instructions shift the bit right or left, but not the sign bit. The SAR copies the sign bit into the emptied bits.
- The shift count can be a constant or a number in the **`cl`** register

```
sar   eax, 5
shl   eax, cl
```

Why Shift?

- Sometimes you need to separate some bits from a word
- Example: separate bits 2 – 5 of an int

1 1	1 0 1 1	0 1

```
mov eax, someint   ; get word
shr  eax, 2        ; shift desired bits to end
and  eax, 15       ; move other bits
mov results, eax   ; save results
```

Shift Arithmetic

- Shifting a number to the left multiplies it by 2 for each bit you shift.
- Shifting a number to the right divides it by 2 for each bit you shift.

Shift Example

```
int dog=3;
_asm {
    mov   eax,dog  ; eax = 3
    sal   eax,2    ; eax = 12
    sar   eax,1    ; eax = 6
}
```

What is the result after **shr ax,5** when the initial value of ax is 1100000001100001

1. 1111111000000011
2. 0000011000000011
3. 0000110000100000
4. 1100000001100001

Try It

- Complete this program to compute the average of two numbers

```
int  bull, dog, goat, two=2;
cin >> bull >> dog;
_asm{
    // set goat to the average of dog and bull
}
cout << goat << endl;
```

Possible Solution

```
int  bull, dog, goat, two=2;
cin >> bull >> dog;
_asm{
    mov eax, bull       ; eax = bull
    add eax, dog        ; eax = bull + dog
    mov edx, 0          ; clear for divide
    idiv   two          ;divide by 2
    mov goat, eax       ; save result
}
cout << goat << endl;
```

Another Possible Solution

```
int  bull, dog, goat;
cin >> bull >> dog;
_asm{
    mov eax, bull       ; eax = bull
    add eax, dog        ; eax = bull + dog
    sar eax, 1          ; divide by 2

    mov goat, eax       ; save result
}
cout << goat << endl;
```

Increment and Decrement

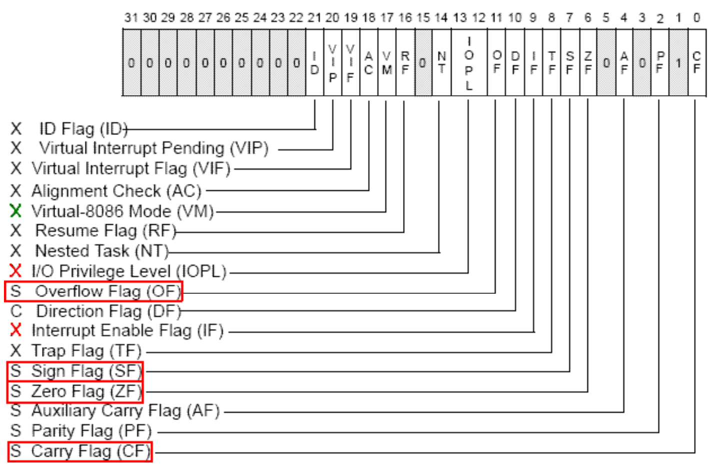
- The **`inc`** and **`dec`** instructions are one of the few that can run on memory locations without using the registers.
- You can increment or decrement the value in a register for memory location

```
inc   eax
dec   memoryAddr
```

Intel Status Register

- The status register records the results of executing the instruction.
- Performing arithmetic sets the status register.
- The compare instruction does a subtraction, but doesn’t store the results. It just sets the status flags.
- All jump instructions are based on the status register

Intel Status Register



Compare Instruction

- The **`cmp`** instruction compares two values and sets the status flags appropriately.

```
cmp   register, operand
```

- where the operand can be a memory location or a register
- The compare instruction subtracts the operand from the register value, but does not save the results.

Jump statements

- A **`JMP`** instruction (sometimes called *branch*) causes the flow of execution to go to a specified location.
- A **`JMP`** instruction loads the Program Counter with the specified address.
- An unconditional jump always jumps.
- A conditional jump will do nothing if the condition is not met.
- Some architectures have a separate compare instruction.

Labels in Assembler

- You can attach a name to a memory location in assembler. This allows you to use the name instead of numerical address
- Labels start in first column and end with a colon :

```
        jmp    rabbit
        // some other stuff here
rabbit: mov    eax, dog
```

Jumps Based on Status Flags

JE	Jump if equal	ZF=1
JZ	Jump if zero	ZF=1
JNE	Jump if not equal	ZF=0
JNZ	Jump if not zero	ZF=0
JLE	Jump if less or equal	ZF=1 or SF≠OF
JL	Jump if less	SF≠OF
JNS	Jump if not sign	SF=0
JS	Jump if sign	SF=1
JGE	Jump if greater or equal	SF=OF
JG	Jump if greater	ZF=0 and SF=OF

The JS will jump if the last arithmetic operation set the sign bit. This means the number was

1. Positive
2. Negative
3. Even
4. Odd
5. Zero

Program Counter

- The Program Counter or Instruction Pointer Register contains the address of the next instruction to be executed.
- At the beginning of the fetch/execute cycle, the CPU fetches the instruction whose address is in the program counter.
- A jump instruction is just a load of the program counter register.

Software Controls

- Assembler only has a simple compare instruction. Jumps are based on the compare.
- Assembler does not have:
  - for
  - while
  - do while
  - switch
  - break
  - else portion of an if

If statements

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

```
if (cat ==dog)
   cow = goat;
else
   cow = bull;
```

```
        MOV   eax,cat
        CMP   eax,dog
        JNE   noteq
        MOV   edx,goat
        JMP   after
noteq:  MOV   edx,bull
after:  MOV   cow,edx
```

Try It

- Complete this program in assembler

```
int  cow=0, dog=0, cat=3;
cin >> dog >> cat;
_asm{
      if (dog > cat)
            cow = 1;
      else
            dog = cat;
}
```

Loops

- There are usually no hardware instructions that directly implement loops (i.e. for, while, do)
- Loops are implemented with conditional jumps.

```
while (what == ever) {
    // something
}
```

```
again: mov  eax,what
       cmp  eax,ever
       jne  endloop
       // something
       jmp  again
endloop:
```

Try It

- Complete this program in assembler

```
int  cow=0, dog=0, cat=3;
_asm{
    do {        // convert this to assembler
        cow++;
        dog = dog + cat;
    } while (dog < 12);
}
```

Possible Solution

```
 cow=0; dog=0; cat=3;
_asm{
    mov eax, dog      ; put dog in eax
again:
    inc cow           ; cow++
    add eax, cat      ; add cat to dog
    cmp eax, 12       ; < 12 ?
    jl  again         ; repeat if not
 }
```

No Operation

- The **NOP** instruction is a one byte instruction that does **nothing**.
- Executing a **NOP** does not change any registers or status bits.
- When patching a machine language program, it is sometimes useful to be able to add a few extra instructions that don't change the program.