Assembly Language Programming

ICT 2203 Computer Architecture

Based on Computer Organization and Architecture, 6th Edition, by William Stallings

High and Low Level Languages

- C, C++, Java, Basic and the likes are all high level languages.
- They are machine independent.
- Assembler language is a low level language.
- Each different computer architecture has its own assembler language.
- The machine dependent compilers translate the high level language to machine language.

Microcode

- Assembler or machine language is the lowest level access a programmer has to the hardware.
- Internally, the machine language is implemented with microcode.
- Microcode is a series of instruction words that turn switches on and off to implement each machine language instruction.

Compilers

- A compiler translates a high level language, such as C or C++, into machine language.
- Each line of a high level language usually produces many machine instructions.
- Compilers can rearrange the instructions to produce optimal code.

Compiled Results

```
//  C++ or Java method
int isflush( void ) {
int  i, ok;
ok = FLUSHBASE + hand[4].face - 5;
for (i = 0; i < 4; i++)
     if (hand[i].suit != hand[i+1].suit)
          ok = 0;
return ok;
}
```

```
// Portion of same method in assembler
         mov    eax,  PTR hand+32
         add    eax, 117
         mov     PTR ok[ebp], eax
         mov     PTR i[ebp], 0
         jmp    SHORT label4
label3:  mov    eax,  PTR i[ebp]
         add    eax, 1
         mov     PTR i[ebp], eax
label4:  cmp     PTR i[ebp], 4
         jge    SHORT label2
         mov    eax,  PTR i[ebp]
         mov    ecx,  PTR i[ebp]
         mov    edx,  PTR hand[eax*8+4]
         cmp    edx,  PTR hand[ecx*8+12]
         je     SHORT label1
         mov     PTR ok[ebp], 0
label1:  jmp    SHORT label3
label2:  mov    eax, PTR ok[ebp]
```

Architecture Specific

- The same C++, Fortran, Pascal or Java source code can be compiled on any architecture.
- When executed, it will give the same results.
- Each architecture has its own assembler language.
- Assembler for one type of machine will not run on another machine.
- Assembler language is a simplified way of writing machine language.

Writing Assembler

- You need an assembler program that will translate your assembler source into machine language.
- You can use the Inline Assembler in Microsoft Visual Studio to embed assembly-language instructions directly in your C and C++ source programs.

```
__asm {
    assembler code here
}
```

- More details on using inline assembler are at:
  - https://docs.microsoft.com/en-us/cpp/assembler/inline/inline-assembler

Assembler Programmer’s Model of Processor

- Registers
  - Everything moves through the registers
  - Arithmetic appears to occur in the registers.
- Status Register
  - Updated automatically by most instructions
  - Status bits are the basis for jumps
- Instructions and data are in memory
  - The assembler program deals with addresses

Registers

- Registers are high speed, temporary storage in the processor.
- Some registers you can manipulate directly with assembler.
- The number of registers varies with the architecture.
- The Pentium has 8. IBM mainframes have 16, Itanium has 32.
- In some architectures, all registers are the same. In others, registers are specialized.

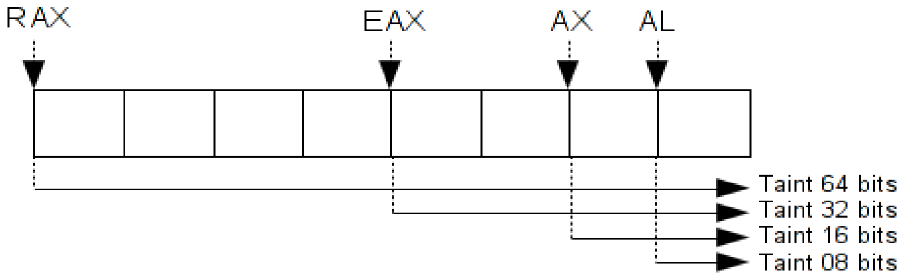
Registers Do Everything

- All data moves through the registers
  - Register to register instructions
  - Memory to register instructions
  - Memory to memory instructions (rare).
- Registers can hold addresses.
- Instructions accessing data in memory, can use an index register to specify the address

Changing Names

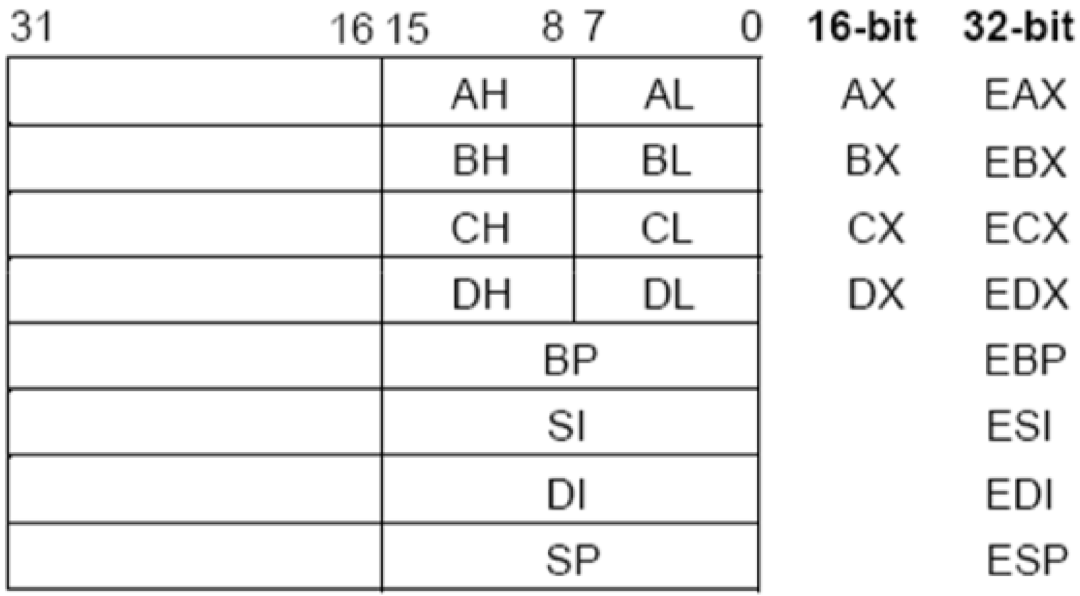
- The first IBM PC had an Intel 8088 with 16 bit registers.
- The registers were named **AX**, **BX**, etc.
- When Intel extended the processor to 32 bit registers, they called the longer registers **EAX**, **EBX**, etc.
- **AX** is the lower 16 bits of **EAX**.
- **AH** and **AL** are the high and low byte of the 16 bit register, now bytes 3 & 4.

Intel Registers



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

General-Purpose 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

General or Specialized

- In some architectures all of the registers have the same functionality. In other machines the registers each have a specific purpose.
- The Intel registers have special purposes, although most can do several operations.
  - **`EAX`** Accumulator for arithmetic
  - **`EBX`** Pointer to data
  - **`ECX`** Counter for loop operations
  - **`EDX`** I/O 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)
- `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
}
```

Data Types

- Hardware 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 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 1

```
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
}
```

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

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 or memory location

```
inc     eax
dec     memoryAddr
```

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`**.

Multiplication

- The **`imul`** signed multiply instruction has three forms.
- Multiply memory *** EAX** .

```
imul      memory
```

- Multiply memory ***** register.

```
imul      reg, memory
```

- Multiply the value in the memory location times the constant and store the result in the register .

```
imul      reg, memory, const
```

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 Examples 3 & 4

```
int dog=3, cat=4, bird=5, cow;
_asm {    // cow = dog * cat / bird;
    mov   eax,dog
    imul  cat
    idiv  bird
    mov   cow,eax
}
```

```
int dog=3, cat=4, bird=5, cow;
_asm {    // cow = dog % cat - bird;
    mov   eax,dog
    mov   edx,0      ; clear EDX
    idiv  cat
    sub   edx,bird
    mov   cow,edx
}
```

Shifts

- The shift instructions can shift the values in a register or memory location.
- 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 shift count can be a constant or the **`cl`** reg.
  - **`sar    eax, 5`**
  - **`shl    eax, cl`**

Shift Example

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

Learn Assembly Language Programming

- Assembly Programming Tutorial
  - https://www.tutorialspoint.com/assembly_programming/index.htm
- Compile and Execute Assembly Online
  - https://www.tutorialspoint.com/compile_assembly_online.php
- Online Assembler - NASM Compiler IDE
  - https://www.jdoodle.com/compile-assembler-nasm-online/

Next:

Multicores, Multiprocessors, and Clusters