Assembler Language

COMP375
Computer Architecture and Organization
“A programming language is low level when its programs require attention to the irrelevant.”

Alan J. Perlis
Clickers

We are clicking TODAY

We are using the loaner clickers
Goals for Today

• Introduce Intel assembler language
• Learn how to write simple assembler programs
High and Low Levels

- C++, Java, Pascal, Cobol, Basic and the like are all high level languages. They are machine independent. The machine dependent compilers translate the high level language to machine language.

- Assembler language is a low level language. Each different computer architecture has its own assembler 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
// 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, hand+32
add   eax, 117
mov   ok[ebp], eax
mov   i[ebp], 0
jmp   SHORT label4
label3: mov   eax, i[ebp]
         add   eax, 1
         mov   i[ebp], eax
label4: cmp   i[ebp], 4
         jge   SHORT label2
         mov   eax, i[ebp]
         mov   ecx, i[ebp]
         mov   edx, hand[eax*8+4]
         cmp   edx, hand[ecx*8+12]
         je    SHORT label1
         mov   ok[ebp], 0
label1: jmp   SHORT label3
label2: mov   eax, ok[ebp]
         mov   ok[ebp], eax
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

• Microsoft Visual Studio Community will allow you to embed assembler into a C++ program

visualstudio.microsoft.com

Details on using inline assembler are at

Assembling with a Mac

• Students have found the Xcode program allows you to code assembler on a Mac
• Available for free from the Mac store
• Use global variables and not local variables
• Do not put an underscore with _asm {
But I only know Java!

- Our simple C programs look just like Java
- The source code for all of the C programs shown in the slides are available on the class website
- Cut and paste your stuff into the examples

*These are simple programs
*We don’t need no stinking classes*
Inline Assembler

• You can insert assembler code in a C++ program using Microsoft Visual Studio.
  
```c++
    _asm {
      assembler code here
    }
```

• You can reference C++ program variables by name
Assembler Programmer’s Model of the 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
- User registers are the ones 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)*

• Although arithmetic is done in the ALU, it *appears* to be done in the register

• Registers can hold addresses

• Instructions accessing data in memory can use an index register to specify the address
Usual Assembler

dog = cat;

• move the value from memory location cat into a register
• move the value from the register to memory location dog
dog = cat + cow;

• move the value from memory location cat into a register
• Add the value from memory location cow to the register
• Move the value from register to memory dog
Intel Pentium has 8 User Registers

<table>
<thead>
<tr>
<th>Register</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EAX</td>
<td>General use, division only in EAX</td>
</tr>
<tr>
<td>EBX</td>
<td>General use</td>
</tr>
<tr>
<td>ECX</td>
<td>General use</td>
</tr>
<tr>
<td>EDX</td>
<td>General use</td>
</tr>
<tr>
<td>EBP</td>
<td>Base pointer</td>
</tr>
<tr>
<td>ESI</td>
<td>Source pointer</td>
</tr>
<tr>
<td>EDI</td>
<td>Destination pointer</td>
</tr>
<tr>
<td>ESP</td>
<td>Stack pointer</td>
</tr>
</tbody>
</table>
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

<table>
<thead>
<tr>
<th>General-Purpose Registers</th>
<th>16-bit</th>
<th>32-bit</th>
</tr>
</thead>
<tbody>
<tr>
<td>AH</td>
<td>AX</td>
<td>EAX</td>
</tr>
<tr>
<td>AL</td>
<td></td>
<td></td>
</tr>
<tr>
<td>BH</td>
<td>BX</td>
<td>EBX</td>
</tr>
<tr>
<td>BL</td>
<td></td>
<td></td>
</tr>
<tr>
<td>CH</td>
<td>CX</td>
<td>ECX</td>
</tr>
<tr>
<td>CL</td>
<td></td>
<td></td>
</tr>
<tr>
<td>DH</td>
<td>DX</td>
<td>EDX</td>
</tr>
<tr>
<td>DL</td>
<td></td>
<td></td>
</tr>
<tr>
<td>BP</td>
<td></td>
<td>EBP</td>
</tr>
<tr>
<td>SI</td>
<td></td>
<td>ESI</td>
</tr>
<tr>
<td>DI</td>
<td></td>
<td>EDI</td>
</tr>
<tr>
<td>SP</td>
<td></td>
<td>ESP</td>
</tr>
</tbody>
</table>
Intel® Extended Memory 64 Technology

- Most processors support 64 bit computing
- The CPU must be in 64 bit mode, set by the OS
- Integers can be 64 bits in length holding numbers up to $10^{19}$ instead of $10^9$
- Addresses are 64 bits allowing you to address more than 4 GB of RAM

- Apple computers always run in 64 bit mode
Log₂ ( 4 GB ) is

A. 4
B. 16
C. 32
D. 64
E. none of the above
64 bit Registers

• The Intel Pentium has 16 user registers
• Registers RAX, RBX, RCX and RDX extend the IA-32 registers EAX, EBX, ECX and EDX
• There are 8 new registers R8, R9, … , R15
• You cannot use the AH, BH, CH and DH parts of the IA-32 registers at the same time as the new 64 bit registers
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)**
- 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

```c
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
}
```

```asm1```
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
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
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
  \[\text{add eax, dog}\]
• The result is saved in the first register
## Arithmetic and Logical Instructions

<table>
<thead>
<tr>
<th>mnemonic</th>
<th>operation</th>
</tr>
</thead>
<tbody>
<tr>
<td>ADD</td>
<td>Add</td>
</tr>
<tr>
<td>SUB</td>
<td>Subtract</td>
</tr>
<tr>
<td>MUL</td>
<td>Unsigned Multiply</td>
</tr>
<tr>
<td>IMUL</td>
<td>Signed Multiply</td>
</tr>
<tr>
<td>DIV</td>
<td>Unsigned Divide</td>
</tr>
<tr>
<td>IDIV</td>
<td>Signed Divide</td>
</tr>
<tr>
<td>AND</td>
<td>Logical AND</td>
</tr>
<tr>
<td>OR</td>
<td>Logical OR</td>
</tr>
</tbody>
</table>
Arithmetic Example

```c
int dog=3, cat=4, bird=5;
__asm {
    // bird = dog + cat;
    mov eax, dog
    add eax, cat
    mov bird, eax
}
```
Arithmetic Example 2

```c
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?

```c
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
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
  ```
Try it

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

```cpp
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

<table>
<thead>
<tr>
<th>EDX</th>
<th>EAX</th>
</tr>
</thead>
<tbody>
<tr>
<td>bits 63, 62, … 33, 32</td>
<td>bits 31, 30 … 2, 1, 0</td>
</tr>
</tbody>
</table>
Multiplication

- The imul signed multiply instruction has three forms
- Multiply memory * EAX
  \[
  \text{imul} \quad \text{memory}
  \]
- Multiply memory * register
  \[
  \text{imul} \quad \text{reg, memory}
  \]
- Multiply the value in the memory location times the constant and store the result in the register
  \[
  \text{imul} \quad \text{reg, memory, const}
  \]
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
  \texttt{idiv memoryAddr}
Arithmetic Example 3

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

```c
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 SAR copies the sign bit into the emptied bits

• The shift count can be a constant or the cl reg

    sar    eax, 5    shl eax, cl
int dog=3;

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

Shift Example
Clickers

Clickers are REQUIRED starting Wednesday

We are using the loaner clickers