More
Assembler Language

COMP375
Computer Architecture and Organization
“You can do everything in assembler, but no one wants to program in assembler anymore.”

Yukihiro Matsumoto
Assembler Assignment

• The first assembler programming assignment has been posted on Blackboard
• You are required to write four short program segments in assembler
• It is easiest to embed the assembler in C++
• Upload your .cpp files to Blackboard by noon on Wednesday, September 4, 2019
Goals for Today

• Introduce assembler language constructs to:
  – multiply and divide
  – shift

• Write simple assembler programs
Each Layer Adds More Detail

• High level application
  \[ \text{result} = \text{first} + \text{second}; \]

• C++ program

• Assembler
  \[
  \text{mov ebx, first} \\
  \text{add ebx, second} \\
  \text{mov result, ebx}
  \]
### Intel User Registers

<table>
<thead>
<tr>
<th>Register size in bits</th>
<th>64</th>
<th>32</th>
<th>16</th>
<th>8</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>RAX</strong></td>
<td>EAX</td>
<td>AX</td>
<td>AL</td>
<td></td>
</tr>
<tr>
<td><strong>RBX</strong></td>
<td>EBX</td>
<td>BX</td>
<td>BL</td>
<td></td>
</tr>
<tr>
<td><strong>RCX</strong></td>
<td>ECX</td>
<td>CX</td>
<td>CL</td>
<td></td>
</tr>
<tr>
<td><strong>RDX</strong></td>
<td>EDX</td>
<td>DX</td>
<td>DL</td>
<td></td>
</tr>
<tr>
<td><strong>RBP</strong></td>
<td>EBP</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>RSI</strong></td>
<td>ESI</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>RDI</strong></td>
<td>EDI</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>RSP</strong></td>
<td>ESP</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
All lengths are the same register

- All lengths of a register share the same lower bits
  - R?X is 64 bits
  - E?X is 32 bits
  - ?X is 16 bits
  - ?L is 8 bits
**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 registers
Constants

• Assembler programs can also use constants
• If you want to move the number 47 to eax
  \[\text{mov eax,47}\]
• Constants can also be characters
  \[\text{mov al,'Z'}\]
• You can also move constants to memory
  \[\text{mov aardvark,15}\]
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

<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 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 bird at the end?

```c
int dog=5, cat=2, bird=3;
_asm {
    mov    eax,dog
    sub    eax,bird
    add    eax,cat
    mov    bird,eax
}
```

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

<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 multiple forms

• Multiply memory * EAX, save in EDX:EAX
  
  \[
  \textbf{imul} \quad \text{mem}or\text{reg}
  \]

• Multiply memory * register, save the 32 bit result in the register
  
  \[
  \textbf{imul} \quad \text{reg, mem}or\text{reg}
  \]
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}
Division Inconsistency

• The idiv instruction cannot use a constant as an operand
  \[\text{idiv } 3 \quad \text{// will not divide by 3}\]

• A register or memory operand must be used
  \[
  \text{mov ebx, 3} \\
  \text{idiv ebx}
  \]
Arithmetic Example 2

```c
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

```c
int dog=9, cat=4, bird=5, cow;
诓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
}
```
int dog=9, cat=4, bird=5, cow;

_asm { // cow = dog % cat + bird;
    mov eax, dog; ; move dog to eax reg
    cdq ; expand to edx:eax
    idiv cat ; edx = dog % cat
    add edx, bird; ; add bird to dog % cat
    mov cow, edx ; save result in cow
}
# Logical Operations

<table>
<thead>
<tr>
<th>X</th>
<th>Y</th>
<th>AND</th>
<th>OR</th>
<th>XOR</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>
Logical Summary

- $Z \text{ AND } 0 = 0$
- $Z \text{ AND } 1 = Z$
- $Z \text{ OR } 0 = Z$
- $Z \text{ OR } 1 = 1$
- $Z \text{ XOR } 1 = \bar{Z}$
- $Z \text{ XOR } 0 = Z$
Logical Instructions

• The AND, OR and XOR assembler instructions perform the operation on the bits of two operands

\[
\begin{align*}
\text{mov} & \quad \text{AL, cat} \quad \text{cat} = 01100101 \\
\text{and} & \quad \text{AL, dog} \quad \text{dog} = 11001100 \\
\text{mov} & \quad \text{rat, AL} \quad \text{rat} = 01000100
\end{align*}
\]
Shifting Bits

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

<table>
<thead>
<tr>
<th>before</th>
<th></th>
<th>after</th>
</tr>
</thead>
<tbody>
<tr>
<td>00000101</td>
<td>Left shift 2</td>
<td>00010100</td>
</tr>
<tr>
<td>11111011</td>
<td>Right shift 1</td>
<td>01111101</td>
</tr>
<tr>
<td>11111011</td>
<td>Arith R shift 1</td>
<td>11111101</td>
</tr>
</tbody>
</table>
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.

```assembly
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, tiger          ; get word
shr eax, 2              ; shift desired bits to end
and eax, 0xF            ; move other bits
mov lion, 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
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

A. 1111111000000011
B. 0000011000000011
C. 0000110000100000
D. 1100000001100001
What is the result after `sar ax,5` when the initial value of `ax` is `1100000001100001`?

A. 1111111000000011
B. 0000011000000011
C. 0000110000100000
D. 1100000001100001
Try It

- Complete this program to compute the average of two numbers

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

```cpp
int bull, dog, goat, two=2;
cin >> bull >> dog;
_asm{
    mov eax, bull ; eax = bull
    add eax, dog ; eax = bull + dog
    cdq
    idiv two ; divide by 2
    mov goat, eax ; save result
}
cout << goat << endl;
```
Another Possible Solution

```cpp
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;
```
What is in dog at the end?

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

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

\[
\text{inc eax} \\
\text{dec memoryAddr}
\]
Intel Status Register

- The status register records the results of executing the instruction
- Performing arithmetic sets the status register
- Some instructions, such as mov, push or jmp, do not change the status flags
- All jump instructions are based on the status register
Intel Status Register

31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0

- X: ID Flag (ID)
- X: Virtual Interrupt Pending (VIP)
- X: Virtual Interrupt Flag (VIF)
- X: Alignment Check (AC)
- X: Virtual-8086 Mode (VM)
- X: Resume Flag (RF)
- X: Nested Task (NT)
- X: I/O Privilege Level (IOPL)
- S: Overflow Flag (OF)
- C: Direction Flag (DF)
- X: Interrupt Enable Flag (IF)
- X: Trap Flag (TF)
- S: Sign Flag (SF)
- S: Zero Flag (ZF)
- S: Auxiliary Carry Flag (AF)
- S: Parity Flag (PF)
- S: Carry Flag (CF)
Flag Setting During Execution

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

<table>
<thead>
<tr>
<th>Zero</th>
<th>Sign</th>
<th>Carry</th>
<th>Overflow</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

Zero: 0
Sign: 0
Carry: 0
Overflow: 0
Compare Instruction

• The `cmp` instruction compares two values and sets the status flags appropriately
  \[
  \text{cmp} \quad \text{register, operand}
  \]
• 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
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 \(SF \neq OF\)
JL  Jump if less  \(SF \neq OF\)
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 \(SF=OF\)
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
```
Compare and Jump

• A compare instruction sets the flags as if the second operand was subtracted from the first.
  \[
  \text{cmp } \text{eax, goat} \\
  \text{jge } \text{someplace}
  \]

• This will jump if eax ≥ goat

• Imagine the jump comparison is between the register and operand
Assembler Assignment

• The first assembler programming assignment has been posted on Blackboard
• You are required to write four short program segments in assembler
• It is easiest to embed the assembler in C++
• Upload your .cpp files to Blackboard by noon on Wednesday, September 4, 2019