Assembly Language: IA-32 Instructions

Professor Jennifer Rexford
http://www.cs.princeton.edu/~jrex

Goals of Today’s Lecture

• Help you learn…
  ◦ To manipulate data of various sizes
  ◦ To leverage more sophisticated addressing modes
  ◦ To use condition codes and jumps to change control flow

• Focusing on the assembly-language code
  ◦ Rather than the layout of memory for storing data

• Why?
  ◦ Know the features of the IA-32 architecture
  ◦ Write more efficient assembly-language programs
  ◦ Understand the relationship to data types and common programming constructs in higher-level languages
### Variable Sizes in High-Level Language

- **C data types vary in size**
  - Character: 1 byte
  - Short, int, and long: varies, depending on the computer
  - Float and double: varies, depending on the computer
  - Pointers: typically 4 bytes

- **Programmer-created types**
  - Struct: arbitrary size, depending on the fields

- **Arrays**
  - Multiple consecutive elements of some fixed size
  - Where each element could be a struct

### Supporting Different Sizes in IA-32

- **Three main data sizes**
  - Byte (b): 1 byte
  - Word (w): 2 bytes
  - Long (l): 4 bytes

- **Separate assembly-language instructions**
  - E.g., addb, addw, and addl

- **Separate ways to access (parts of) a register**
  - E.g., %ah or %al, %ax, and %eax

- **Larger sizes (e.g., struct)**
  - Manipulated in smaller byte, word, or long units
**Byte Order in Multi-Byte Entities**

- **Intel is a little endian architecture**
  - Least significant byte of multi-byte entity is stored at lowest memory address
  - “Little end goes first”

```
00000000 00000000 00000000 00000101
1000      1001      1002      1003
```

The int 5 at address 1000:

- Some other systems use big endian
  - Most significant byte of multi-byte entity is stored at lowest memory address
  - “Big end goes first”

```
00000000 00000000 00000000 00000101
1000      1001      1002      1003
```

**Little Endian Example**

```c
int main(void) {
    int i=0x003377ff, j;
    unsigned char *p = (unsigned char *) &i;
    for (j=0; j<4; j++)
        printf("Byte %d: %x\n", j, p[j]);
}
```

Output on a little-endian machine:

- Byte 0: ff
- Byte 1: 77
- Byte 2: 33
- Byte 3: 0
IA-32 General Purpose Registers

<table>
<thead>
<tr>
<th>31</th>
<th>15</th>
<th>8</th>
<th>7</th>
<th>0</th>
<th>16-bit</th>
<th>32-bit</th>
</tr>
</thead>
<tbody>
<tr>
<td>AH</td>
<td>AL</td>
<td>AX</td>
<td>EAX</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>BH</td>
<td>BL</td>
<td>BX</td>
<td>EBX</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>CH</td>
<td>CL</td>
<td>CX</td>
<td>ECX</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>DH</td>
<td>DL</td>
<td>DX</td>
<td>EDX</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>SI</td>
<td></td>
<td></td>
<td></td>
<td>ESI</td>
<td></td>
<td></td>
</tr>
<tr>
<td>DI</td>
<td></td>
<td></td>
<td></td>
<td>EDI</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

General-purpose registers

C Example: One-Byte Data

Global char variable i is in %al, the lower byte of the “A” register.

```c
char i;
...
if (i > 5) {
  i++;
else
  i--; 
}
cmpb $5, %al
  jle else
  incb %al
else:
  decb %al
endif:
```
C Example: Four-Byte Data

Global int variable i is in %eax, the full 32 bits of the “A” register.

```c
int i;
...
if (i > 5) {
  i++;
} else
  i--;
```

```assembly
cmpl $5, %eax
jle else
incli %eax
jmp endif
else:
decli %eax
endif:
```

Loading and Storing Data

- Processors have many ways to access data
  - Known as “addressing modes”
  - Two simple ways seen in previous examples

- Immediate addressing
  - Example: movl $0, %ecx
  - Data (e.g., number “0”) embedded in the instruction
  - Initialize register ECX with zero

- Register addressing
  - Example: movl %edx, %ecx
  - Choice of register(s) embedded in the instruction
  - Copy value in register EDX into register ECX
Accessing Memory

- Variables are stored in memory
  - Global and static local variables in Data or BSS section
  - Dynamically allocated variables in the heap
  - Function parameters and local variables on the stack

- Need to be able to load from and store to memory
  - To manipulate the data directly in memory
  - Or copy the data between main memory and registers

- IA-32 has many different addressing modes
  - Corresponding to common programming constructs
  - E.g., accessing a global variable, dereferencing a pointer, accessing a field in a struct, or indexing an array

Direct Addressing

- Load or store from a particular memory location
  - Memory address is embedded in the instruction
  - Instruction reads from or writes to that address

- IA-32 example: `movl 2000, %ecx`
  - Four-byte variable located at address 2000
  - Read four bytes starting at address 2000
  - Load the value into the ECX register

- Useful when the address is known in advance
  - Global variables in the Data or BSS sections

- Can use a label for (human) readability
  - E.g., “i” to allow “movl i, %eax”
Indirect Addressing

• Load or store from a previously-computed address  
  ○ Register with the address is embedded in the instruction  
  ○ Instruction reads from or writes to that address

• IA-32 example: movl (%eax), %ecx  
  ○ EAX register stores a 32-bit address (e.g., 2000)  
  ○ Read long-word variable stored at that address  
  ○ Load the value into the ECX register

• Useful when address is not known in advance  
  ○ Dynamically allocated data referenced by a pointer  
  ○ The "(%eax)" essentially dereferences a pointer

Base Pointer Addressing

• Load or store with an offset from a base address  
  ○ Register storing the base address  
  ○ Fixed offset also embedded in the instruction  
  ○ Instruction computes the address and does access

• IA-32 example: movl 8(%eax), %ecx  
  ○ EAX register stores a 32-bit base address (e.g., 2000)  
  ○ Offset of 8 is added to compute address (e.g., 2008)  
  ○ Read long-word variable stored at that address  
  ○ Load the value into the ECX register

• Useful when accessing part of a larger variable  
  ○ Specific field within a “struct”  
  ○ E.g., if “age” starts at the 8th byte of “student” record
Indexed Addressing

• Load or store with an offset and multiplier
  ◦ Fixed based address embedded in the instruction
  ◦ Offset computed by multiplying register with constant
  ◦ Instruction computes the address and does access

• IA-32 example: movl 2000,(%eax,4), %ecx
  ◦ Index register EAX (say, with value of 10)
  ◦ Multiplied by a multiplier of 1, 2, 4, or 8 (say, 4)
  ◦ Added to a fixed base of 2000 (say, to get 2040)

• Useful to iterate through an array (e.g., a[i])
  ◦ Base is the start of the array (i.e., “a”)
  ◦ Register is the index (i.e., “i”)
  ◦ Multiplier is the size of the element (e.g., 4 for “int”)

Indexed Addressing Example

```c
int a[20]; — global variable
int i, sum=0;
for (i=0; i<20; i++)
    sum += a[i];
```

```
 movl $0, %eax
 movl $0, %ebx
 sumloop:
    movl a(%eax,4), %ecx
    addl %ecx, %ebx
    incl %eax
    cmpl $19, %eax
    jle sumloop
```
**Effective Address: More Generally**

Offset = \[
\begin{array}{c}
\text{Base} \\
\text{Index} \\
\text{scale} \\
\text{displacement}
\end{array}
\] + \[
\begin{array}{c}
eax \\
ebx \\
ecx \\
edx \\
esp \\
esp \\
esi \\
edi
\end{array}
\] + \[
\begin{array}{c}
eax \\
ebx \\
ecx \\
edx \\
esp \\
esp \\
esi \\
edi
\end{array}
\] \times \[
\begin{array}{c}
1 \\
2 \\
4 \\
8 \\
\end{array}
\] + \[
\begin{array}{c}
\text{None} \\
\text{8-bit} \\
\text{16-bit} \\
\text{32-bit}
\end{array}
\]

- **Displacement**
  - \texttt{movl foo, %ebx}

- **Base**
  - \texttt{movl (%eax), %ebx}

- **Base + displacement**
  - \texttt{movl foo(%eax), %ebx}
  - \texttt{movl 1(%eax), %ebx}

- **(Index * scale) + displacement**
  - \texttt{movl (,%eax,4), %ebx}

- **Base + (index * scale) + displacement**
  - \texttt{movl foo(%edx,%eax,4),%ebx}

---

**Data Access Methods: Summary**

- **Immediate addressing**: data stored in the instruction itself
  - \texttt{movl $10, %ecx}

- **Register addressing**: data stored in a register
  - \texttt{movl %eax, %ecx}

- **Direct addressing**: address stored in instruction
  - \texttt{movl foo, %ecx}

- **Indirect addressing**: address stored in a register
  - \texttt{movl (%eax), %ecx}

- **Base pointer addressing**: includes an offset as well
  - \texttt{movl 4(%eax), %ecx}

- **Indexed addressing**: instruction contains base address, and specifies an index register and a multiplier (1, 2, 4, or 8)
  - \texttt{movl 2000(,%eax,1), %ecx}
### Control Flow

- **Common case**
  - Execute code sequentially
  - One instruction after another

- **Sometimes need to change control flow**
  - If-then-else
  - Loops
  - Switch

- **Two key ingredients**
  - Testing a condition
  - Selecting what to run next based on result

```plaintext
    cmpl $5, %eax
    jle else
    incl %eax
    jmp endif

else:
    decl %eax
endif:
```

### Condition Codes

- **1-bit registers set by arithmetic & logic instructions**
  - ZF: Zero Flag
  - SF: Sign Flag
  - CF: Carry Flag
  - OF: Overflow Flag

- **Example: “addl Src, Dest” (“t = a + b”)**
  - ZF: set if \( t = 0 \)
  - SF: set if \( t < 0 \)
  - CF: set if carry out from most significant bit
    - \textit{Unsigned} overflow
  - OF: set if two’s complement overflow
    - \((a>0 && b>0 && t<0)\)
    - \((a<0 && b<0 && t>0)\)
Condition Codes (continued)

- Example: “cmpl Src2,Src1” (compare b,a)
  - Like computing a-b without setting destination
  - ZF: set if a == b
  - SF: set if (a-b) < 0
  - CF: set if carry out from most significant bit
    - Used for unsigned comparisons
  - OF: set if two’s complement overflow
    - (a>0 && b<0 && (a-b)<0) || (a<0 && b>0 && (a-b)>0)

- Flags are not set by lea, inc, or dec instructions
  - Hint: this is useful for the extra-credit part of the assembly-language programming assignment! 😊

Example Five-Bit Comparisons

- Comparison: cmp $6, $12
  - Not zero: ZF=0 (diff is not 00000)
  - Positive: SF=0 (first bit is 0)
  - No carry: CF=0 (unsigned diff is correct)
  - No overflow: OF=0 (signed diff is correct)

- Comparison: cmp $12, $6
  - Not zero: ZF=0 (diff is not 00000)
  - Negative: SF=1 (first bit is 1)
  - Carry: CF=1 (unsigned diff is wrong)
  - No overflow: OF=0 (signed diff is correct)

- Comparison: cmp $-6, $-12
  - Not zero: ZF=0 (diff is not 00000)
  - Negative: SF=1 (first bit is 1)
  - Carry: CF=1 (unsigned diff of 20 and 28 is wrong)
  - No overflow: OF=0 (signed diff is correct)
Jumps after Comparison (cmpl)

- **Equality**
  - Equal: je (ZF)
  - Not equal: jne (~ZF)

- **Below/above (e.g., unsigned arithmetic)**
  - Below: jb (CF)
  - Above or equal: jae (~CF)
  - Below or equal: jbe (CF | ZF)
  - Above: ja (~CF | ZF)

- **Less/greater (e.g., signed arithmetic)**
  - Less: jl (SF ^ OF)
  - Greater or equal: jge (~(SF ^ OF))
  - Less or equal: jle ((SF ^ OF) | ZF)
  - Greater: jg (!((SF ^ OF) | ZF))

Branch Instructions

- **Conditional jump**
  - j{l,g,e,ne,...} target if (condition) {eip = target}

<table>
<thead>
<tr>
<th>Comparison</th>
<th>Signed</th>
<th>Unsigned</th>
</tr>
</thead>
</table>
| =          | e      | e        | "equal"
| ≠          | ne     | ne       | "not equal"
| >          | g      | a        | "greater, above"
| ≥          | ge     | ae       | "...-or-equal"
| <          | l      | b        | "less, below"
| ≤          | le     | be       | "...-or-equal"
| overflow/carry | o | c |
| no ovl/carry   | no     | nc       |

- **Unconditional jump**
  - jmp target
  - jmp *register
Jumping

• Simple model of a “goto” statement
  ◦ Go to a particular place in the code
  ◦ Based on whether a condition is true or false
  ◦ Can represent if-the-else, switch, loops, etc.

• Pseudocode example: If-Then-Else

```plaintext
if (Test) {
  then-body;
} else {
  else-body;
}
```

```plaintext
if (!Test) jump to Else;
then-body;
jump to Done;
Else:
  else-body;
Done:
```

Jumping (continued)

• Pseudocode example: Do-While loop

```plaintext
do {
  Body;
} while (Test);
```

```plaintext
loop:
  Body;
if (Test) then jump to loop;
```

• Pseudocode example: While loop

```plaintext
while (Test)
  Body;
```

```plaintext
jump to middle;
loop:
  Body;
middle:
  if (Test) then jump to loop;
```
Jumping (continued)

- Pseudocode example: For loop

```
for (Init; Test; Update)
    Body
```

- Init:
  - if (!Test) jump to done;
- loop:
  - Body;
  - Update;
  - if (Test) jump to loop;
- done:

Arithmetic Instructions

- Simple instructions
  - `add{b,w,l}` source, dest  
    `dest = source + dest`
  - `sub{b,w,l}` source, dest  
    `dest = dest - source`
  - `Inc{b,w,l}` dest  
    `dest = dest + 1`
  - `dec{b,w,l}` dest  
    `dest = dest - 1`
  - `neg{b,w,l}` dest  
    `dest = ~dest + 1`
  - `cmp{b,w,l}` source1, source2  
    `source2 - source1`

- Multiply
  - `mul (unsigned) or imul (signed)`
    `mull %ebx # edx, eax = eax * ebx`

- Divide
  - `div (unsigned) or idiv (signed)`
    `idiv %ebx # edx = edx,eax / ebx`

- Many more in Intel manual (volume 2)
  - `adc, sbb`, decimal arithmetic instructions
Bitwise Logic Instructions

- Simple instructions
  - and{b,w,l} source, dest \( \rightarrow \text{dest} = \text{source} \& \text{dest} \)
  - or{b,w,l} source, dest \( \rightarrow \text{dest} = \text{source} \mid \text{dest} \)
  - xor{b,w,l} source, dest \( \rightarrow \text{dest} = \text{source} \oplus \text{dest} \)
  - not{b,w,l} dest \( \rightarrow \text{dest} = \neg \text{dest} \)
  - sal{b,w,l} source, dest (arithmetic) \( \rightarrow \text{dest} = \text{dest} \ll \text{source} \)
  - sar{b,w,l} source, dest (arithmetic) \( \rightarrow \text{dest} = \text{dest} \gg \text{source} \)

- Many more in Intel Manual (volume 2)
  - Logic shift
  - Rotation shift
  - Bit scan
  - Bit test
  - Byte set on conditions

Data Transfer Instructions

- \textbf{mov{b,w,l} source, dest}
  - General move instruction

- \textbf{push{w,l} source}
  - pushl %ebx \# equivalent instructions
    - subl $4, %esp
    - movl %ebx, (%esp)

- \textbf{pop{w,l} dest}
  - popl %ebx \# equivalent instructions
    - movl (%esp), %ebx
    - addl $4, %esp

- Many more in Intel manual (volume 2)
  - Type conversion, conditional move, exchange, compare and exchange, I/O port, string move, etc.
Conclusions

• Accessing data
  ◦ Byte, word, and long-word data types
  ◦ Wide variety of addressing modes

• Control flow
  ◦ Common C control-flow constructs
  ◦ Condition codes and jump instructions

• Manipulating data
  ◦ Arithmetic and logic operations

• Next time
  ◦ Calling functions, using the stack