Assembly Language:
IA-32 Instructions

Goals of this Lecture

• Help you learn how to:
  • Manipulate data of various sizes
  • Leverage more sophisticated addressing modes
  • Use condition codes and jumps to change control flow
  … and thereby …
  • Write more efficient assembly-language programs
  • Understand the relationship to data types and common programming constructs in high-level languages

• Focus is on the assembly-language code
  • Rather than layout of memory for storing data (precept)
Handling Different Data Sizes

Variable Sizes in High-Level Language

- C data types vary in size
  - Character: 1 byte
  - Short, int, and long: ??
  - Float and double: ??
  - Pointers: ??

- Programmer-created types
  - Struct: ??

- 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., for EAX register: %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

• IA-32 is a little endian architecture
  • Least significant byte of multi-byte entity is stored at lowest memory address
  • “Little end goes first”

<table>
<thead>
<tr>
<th>1003</th>
<th>1002</th>
<th>1001</th>
<th>1000</th>
</tr>
</thead>
<tbody>
<tr>
<td>0000</td>
<td>0000</td>
<td>0000</td>
<td>0001</td>
</tr>
</tbody>
</table>

The 4-byte int 5 (hex 00 00 00 05) at address 1000:

<table>
<thead>
<tr>
<th>1000</th>
<th>1001</th>
<th>1002</th>
<th>1003</th>
</tr>
</thead>
<tbody>
<tr>
<td>0000</td>
<td>0000</td>
<td>0000</td>
<td>0001</td>
</tr>
<tr>
<td>0001</td>
<td>0000</td>
<td>0000</td>
<td>0000</td>
</tr>
<tr>
<td>0000</td>
<td>0001</td>
<td>0000</td>
<td>0000</td>
</tr>
<tr>
<td>0000</td>
<td>0000</td>
<td>0001</td>
<td>0000</td>
</tr>
</tbody>
</table>

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

<table>
<thead>
<tr>
<th>1003</th>
<th>1002</th>
<th>1001</th>
<th>1000</th>
</tr>
</thead>
<tbody>
<tr>
<td>0000</td>
<td>0000</td>
<td>0000</td>
<td>0001</td>
</tr>
</tbody>
</table>

The 4-byte int 5 (hex 00 00 00 05) at address 1000:
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

<table>
<thead>
<tr>
<th>Byte 0</th>
<th>Byte 1</th>
<th>Byte 2</th>
<th>Byte 3</th>
<th>Portable?</th>
</tr>
</thead>
<tbody>
<tr>
<td>ff</td>
<td>77</td>
<td>33</td>
<td>0</td>
<td>•</td>
</tr>
</tbody>
</table>

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>
<th>Common Use</th>
</tr>
</thead>
<tbody>
<tr>
<td>31</td>
<td>AH</td>
<td>AL</td>
<td></td>
<td></td>
<td>AX</td>
<td>EAX</td>
<td>Accumulator</td>
</tr>
<tr>
<td>29</td>
<td>BH</td>
<td>BL</td>
<td></td>
<td></td>
<td>BX</td>
<td>EBX</td>
<td>Pointer to data</td>
</tr>
<tr>
<td>27</td>
<td>CH</td>
<td>CL</td>
<td></td>
<td></td>
<td>CX</td>
<td>ECX</td>
<td>Counter for loops</td>
</tr>
<tr>
<td>25</td>
<td>DH</td>
<td>DL</td>
<td></td>
<td></td>
<td>DX</td>
<td>EDX</td>
<td>I/O pointer</td>
</tr>
<tr>
<td>23</td>
<td>SI</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Pointers (string)</td>
</tr>
<tr>
<td>21</td>
<td>DI</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>source and dest)</td>
</tr>
</tbody>
</table>

General-purpose registers

- EBP: pointer to data on stack
- ESP: stack pointer
C Example: One-Byte Data

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

cmpb $5, %al
jle else
incb %al
jmp endif
else:
decb %al
endif:

char i;
...
if (i > 5) {
   i++;  
else  
   i--;  
}

C Example: Four-Byte Data

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

cmpl $5, %eax
jle else
incl %eax
jmp endif
else:
decl %eax
endif:

int i;
...
if (i > 5) {
   i++;  
else  
   i--;  
}
Memory Addressing Modes

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
    Initialize register ECX with zero
  • Data (e.g., number “0”) embedded in the instruction

• Register addressing
  • Example: movl %edx, %ecx
    Copy value in register EDX into register ECX
  • Choice of register(s) embedded in the instruction
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 copy the data between main memory and registers
  - Or manipulate the data directly in memory

- 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

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

- 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

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

- Useful when address is not known in advance
  - Dereference a pointer, for dynamically allocated data

- 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
  - The “(%eax)” essentially dereferences the pointer stored in register %eax

Base Pointer Addressing

- 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

- Load or store with an offset from a base address
  - movl offset(r1), r2
  - Register r1 stores 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)
  - Load the value into the ECX register
Indexed Addressing

- Load/store with offset made of register, multiplier
  - Fixed base address embedded in the instruction
  - Offset = register * constant multiplier

- 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”)

- 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 (here, 4)
  - Added to a fixed base of 2000 (to get 2040)

Indexed Addressing Example

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

EAX: temporary
EBX: sum
ECX: i

movl $0, %ecx
movl $0, %ebx
sumloop:
    movl a(,%ecx,4), %eax
    addl %eax, %ebx
    incl %ecx
    cmpl $19, %ecx
    jle sumloop

EAX: temporary
EBX: sum
ECX: i

global variable
Effective Address: More Generally

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

- Displacement: \texttt{movl foo, %ebx}
- Base: \texttt{movl (%eax), %ebx}
- Base + displacement: \texttt{movl foo(%eax), %ebx}
- \texttt{movl 1(%eax), %ebx}
- \texttt{movl (%edx,%eax,4), %ebx}
- \texttt{movl foo(%edx,%eax,4), %ebx}
- (Index * scale) + displacement
- Base + (index * scale) + displacement

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: indirect plus offset
  - \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}
  - Can also have an additional displacement register
Condition Codes and Control Flow

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

```asm
    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 (unsigned)
    • 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 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
Jumps after Comparison (cmpl)

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

- **Below/above (e.g., unsigned arithmetic)**
  - Below: jb (CF is set)
  - 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>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>=</td>
<td>e</td>
<td>e</td>
<td>“equal”</td>
</tr>
<tr>
<td>≠</td>
<td>ne</td>
<td>ne</td>
<td>“not equal”</td>
</tr>
<tr>
<td>&gt;</td>
<td>g</td>
<td>a</td>
<td>“greater,above”</td>
</tr>
<tr>
<td>≥</td>
<td>ge</td>
<td>ae</td>
<td>“...-or-equal”</td>
</tr>
<tr>
<td>&lt;</td>
<td>l</td>
<td>b</td>
<td>“less,below”</td>
</tr>
<tr>
<td>≤</td>
<td>le</td>
<td>be</td>
<td>“...-or-equal”</td>
</tr>
<tr>
<td>overflow/carry</td>
<td>o</td>
<td>c</td>
<td></td>
</tr>
<tr>
<td>no ovf/carry</td>
<td>no</td>
<td>nc</td>
<td></td>
</tr>
</tbody>
</table>

- **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

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

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

Jumping (continued)

- Pseudocode example: Do-While loop

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

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

- Pseudocode example: While loop

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

  ```
  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:

Example Instruction Types
Arithmetic Instructions

- **Simple instructions**
  - \( \text{add}\{b,w,l\} \text{ source, dest} \) \( \rightarrow \) dest = source + dest
  - \( \text{sub}\{b,w,l\} \text{ source, dest} \) \( \rightarrow \) dest = dest - source
  - \( \text{Inc}\{b,w,l\} \text{ dest} \) \( \rightarrow \) dest = dest + 1
  - \( \text{dec}\{b,w,l\} \text{ dest} \) \( \rightarrow \) dest = dest - 1
  - \( \text{neg}\{b,w,l\} \text{ dest} \) \( \rightarrow \) dest = ~dest + 1
  - \( \text{cmp}\{b,w,l\} \text{ source1, source2} \) \( \rightarrow \) source2 – source1

- **Multiply**
  - \( \text{mul} \) (unsigned) or \( \text{imul} \) (signed)

- **Divide**
  - \( \text{div} \) (unsigned) or \( \text{idiv} \) (signed)

- **Many more in Intel manual (volume 2)**
  - \( \text{adc, sbb, decimal arithmetic instructions} \)

Bitwise Logic Instructions

- **Simple instructions**
  - \( \text{and}\{b,w,l\} \text{ source, dest} \) \( \rightarrow \) dest = source & dest
  - \( \text{or}\{b,w,l\} \text{ source, dest} \) \( \rightarrow \) dest = source | dest
  - \( \text{xor}\{b,w,l\} \text{ source, dest} \) \( \rightarrow \) dest = source ^ dest
  - \( \text{not}\{b,w,l\} \text{ dest} \) \( \rightarrow \) dest = ~dest
  - \( \text{sal}\{b,w,l\} \text{ source, dest (arithmetic)} \) \( \rightarrow \) dest = dest << source
  - \( \text{sar}\{b,w,l\} \text{ source, dest (arithmetic)} \) \( \rightarrow \) dest = dest >> source

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

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

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

- **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