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
  - 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 the layout of memory for storing data (see precept)

Handling Different Data Sizes

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., 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”
      The 4-byte int 5 (hex 00 00 00 05) at address 1000:
      1000 00000000
      1001 00000000
      1002 00000000
      1003 00000000

• Some other systems use big endian
  • Most significant byte of multi-byte entity is stored at lowest memory address
    • “Big end goes first”
      The 4-byte int 5 (hex 00 00 00 05) at address 1000:
      1000 00000500
      1001 00000000
      1002 00000000
      1003 00000000

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>DI</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Common Use

• AH: Accumulator
• BX: Pointer to data
• CX: Counter for loops
• DX: I/O pointer
• SI: Pointers (string source and dest)
• DI: Pointers (string source and dest)

General-purpose registers

• EBP: pointer to data on stack
• ESP: stack pointer
Memory Addressing Modes

- **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

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--; }
```

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--; }
```

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., "l" 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 \times 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];
```

Data Access Methods: Summary

- Immediate addressing: data stored in the instruction itself
  - movl $10, %ecx
- Register addressing: data stored in a register
  - movl %eax, %ecx
- Direct addressing: address stored in instruction
  - movl foo, %ecx
- Indirect addressing: address stored in a register
  - movl (%eax), %ecx
- Base pointer addressing: indirect plus offset
  - movl 4(%eax), %ecx
- Indexed addressing: instruction contains base address, and specifies an index register and a multiplier (1, 2, 4, or 8)
  - movl 2000(%eax,4), %ecx
  - Can also have an additional displacement register

Effective Address: More Generally

<table>
<thead>
<tr>
<th>Offset</th>
<th>Base</th>
<th>Index</th>
<th>scale</th>
<th>displacement</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>eax</td>
<td>ebx</td>
<td>edx</td>
<td>ecx</td>
</tr>
</tbody>
</table>

- Displacement: movl foo, %ebx
- Base: movl (%eax), %ebx
- Base + displacement: movl foo(%eax), %ebx
- (Index \times scale) + displacement: movl (%edx,%eax,4), %ebx
- Base + (index \times scale) + displacement: movl foo(%edx,%eax,4), %ebx
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)
  - 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 (unsigned)
  - 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)
  01100 01100

• 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)
  00110 00110

• 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)
  01100 01100

Branch Instructions

• Conditional jump
  • \{l,e,ne\} target
  if (condition) (eip = target)

  Comparison  Signed  Unsigned
  • =  e  e  "equal"
  • ≠  ne  ne  "not equal"
  • >  g  a  "greater, above"
  • ≥  ge  ae  "greater or equal"
  • <  l  b  "less, below"
  • ≤  le  be  "less or equal"
  • overflow/carry  o  c  "greater or equal"
  • no overflow/carry  no  nc  "less or equal"

• Unconditional jump
  • jmp  target
  • jmp *register

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))

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

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

- Pseudocode example: While loop

```plaintext
while (Test) {
    Body;
    if (Test) then jump to loop;
}
```

Jumping (continued)

- Pseudocode example: For loop

```plaintext
for (Init; Test; Update) {
    Body;
    if (!Test) jump to done;
    loop:
        Body;
        Update;
        if (Test) jump to loop;
    done:
}
```

Example Instruction Types

- 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
    - `cmp(b,w,l) source1, source2`  
      source2 – source1
  - Multiply
    - `mul (unsigned)` or `imul (signed)`
  - Divide
    - `div (unsigned)` or `idiv (signed)`
  - Many more in Intel manual (volume 2)
    - `adc; sbb; decimal arithmetic instructions`
Bitwise Logic Instructions

- Simple instructions
  - and(b,w,l) source, dest  \( \text{dest} = \text{source} \& \text{dest} \)
  - or(b,w,l) source, dest  \( \text{dest} = \text{source} \lor \text{dest} \)
  - xor(b,w,l) source, dest  \( \text{dest} = \text{source} \oplus \text{dest} \)
  - not(b,w,l) dest  \( \text{dest} = \neg \text{dest} \)
  - sal(b,w,l) source, dest (arithmetic)  \( \text{dest} = \text{dest} << \text{source} \)
  - sar(b,w,l) source, dest (arithmetic)  \( \text{dest} = \text{dest} >> \text{source} \)

- 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
  - Equivalent instructions
    - pushl %ebx
    - subl $4, %esp
    - movl %ebx, (%esp)

- pop(w,l) dest
  - Equivalent instructions
    - popl %ebx
    - 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