Machine-Level Programming II: Arithmetic & Control

Kai Shen

Outline

- Arithmetic operations
- Control: Condition codes
- Conditional branches
- Loops
- Switch

Some Arithmetic Operations

Two Operand Instructions:

<table>
<thead>
<tr>
<th>Format</th>
<th>Computation</th>
</tr>
</thead>
<tbody>
<tr>
<td>addl</td>
<td>Dest = Dest + Src</td>
</tr>
<tr>
<td>subl</td>
<td>Dest = Dest – Src</td>
</tr>
<tr>
<td>imull</td>
<td>Dest = Dest * Src</td>
</tr>
<tr>
<td>sall</td>
<td>Dest = Dest &lt;&lt; Src</td>
</tr>
<tr>
<td>sarl</td>
<td>Dest = Dest &gt;&gt; Src</td>
</tr>
<tr>
<td>shr1</td>
<td>Dest = Dest &amp; Src</td>
</tr>
<tr>
<td>and1</td>
<td>Dest = Dest &amp; Src</td>
</tr>
<tr>
<td>orl</td>
<td>Dest = Dest</td>
</tr>
</tbody>
</table>

One Operand Instructions

<table>
<thead>
<tr>
<th>Format</th>
<th>Computation</th>
</tr>
</thead>
<tbody>
<tr>
<td>incl</td>
<td>Dest = Dest + 1</td>
</tr>
<tr>
<td>dec1</td>
<td>Dest = Dest – 1</td>
</tr>
<tr>
<td>neg1</td>
<td>Dest = ~Dest</td>
</tr>
<tr>
<td>not1</td>
<td>Dest = ~Dest</td>
</tr>
</tbody>
</table>

Arithmetic Example

```c
int arith(int x, int y, int z)
{
    int t1 = x+y;
    int t2 = z+t1;
    int t3 = x+4;
    int t4 = y * 48;
    int t5 = t3 + t4;
    int rval = t2 * t5;
    return rval;
}
```

```assembly
movl 8(%ebp), %ecx # ecx = x
movl 12(%ebp), %edx # edx = y
lea1 (%edx,%edx,2), %eax # eax = y*3
sall $4, %eax # eax *= 16 (t4)
lea1 4(%ecx,%eax), %eax # eax = t4 +x+4 (t5)
addl %ecx, %edx # edx += y (t1)
addl 16(%ebp), %edx # edx += z (t2)
imull %edx, %eax # eax = t2 * t5 (rval)
```
Outline

- Arithmetic operations
- Control: Condition codes
- Conditional branches
- Loops
- Switch

Processor State (IA32, Partial)

- Information about currently executing program
  - Temporary data (%eax, ...)
  - Location of runtime stack (%ebp, %esp)
  - Location of current code control point (%eip, ...)
- Status of recent tests (CF, ZF, SF, OF)

Condition Codes (Arithmetic Setting)

- Single bit registers
  - CF: Carry Flag (for unsigned)
  - ZF: Zero Flag
  - SF: Sign Flag (for signed)
  - OF: Overflow Flag (for signed)
- Set by arithmetic operations
  - Example: addl/addr Src, Dest \( \leftrightarrow t = a + b \)
  - CF set if carry out from most significant bit (unsigned overflow)
  - ZF set if \( t = 0 \)
  - SF set if \( t < 0 \) (as signed)
  - OF set if two’s-complement (signed) overflow
    \( a > 0 \land b > 0 \land (a-b) < 0 \) \( \lor \) \( a < 0 \land b < 0 \land (a-b) > 0 \)
- Not set by lea instruction

Condition Codes (Setting by Compare)

- Setting by compare instruction
  - cmpl Src2, Src1
  - cmpl b, a like computing \( a - b \) without setting destination
- CF set if carry out from most significant bit (used for unsigned comparisons)
- SF set if \( a = b \)
- SF set if \( (a-b) < 0 \) (as signed)
- OF set if two’s-complement (signed) overflow
  \( a > 0 \land b < 0 \land (a-b) < 0 \) \( \lor \) \( a < 0 \land b > 0 \land (a-b) > 0 \)
**Condition Codes (Setting by Test)**

- Setting by Test instruction
  - `testl Src2, Src1` like computing `a & b` without setting destination
  - Sets condition codes based on value of `Src1 & Src2`
  - Useful to have one of the operands be a mask
  - ZF set when `a & b == 0`
  - SF set when `a & b < 0`

**Reading Condition Codes**

- SetX Instructions:
  - Set single byte based on combination of condition codes
  - One of 8 addressable byte registers
  - Does not alter remaining 3 bytes
  - Typically use `movzbl` to finish job

```
int gt (int x, int y) {
  return x > y;
}
```

```
movl 12(%ebp),%eax # eax = y
cmp %eax,%eax,%eax # Compare x : y
setg %al # al = x > y
movzbl %al,%eax # Zero rest of %eax
```

---

<table>
<thead>
<tr>
<th>SetX</th>
<th>Condition</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>seta</td>
<td>ZF</td>
<td>Equal / Zero</td>
</tr>
<tr>
<td>setne</td>
<td>~ZF</td>
<td>Not Equal / Not Zero</td>
</tr>
<tr>
<td>setz</td>
<td>SF</td>
<td>Negative</td>
</tr>
<tr>
<td>setns</td>
<td>~SF</td>
<td>Nonnegative</td>
</tr>
<tr>
<td>setg</td>
<td>~(SF^OF) &amp; ~ZF</td>
<td>Greater (Signed)</td>
</tr>
<tr>
<td>setge</td>
<td>~(SF^OF)</td>
<td>Greater or Equal (Signed)</td>
</tr>
<tr>
<td>setl</td>
<td>(SF^OF)</td>
<td>Less (Signed)</td>
</tr>
<tr>
<td>setle</td>
<td>(SF^OF)</td>
<td>Less or Equal (Signed)</td>
</tr>
<tr>
<td>seta</td>
<td>~CF&amp;~ZF</td>
<td>Above (unsigned)</td>
</tr>
<tr>
<td>setb</td>
<td>CF</td>
<td>Below (unsigned)</td>
</tr>
</tbody>
</table>

**Outline**

- Arithmetic operations
- Control: Condition codes
- Conditional branches & Moves
- Loops
- Switch
Jumping

**jX Instructions**
- Jump to different part of code depending on condition codes

<table>
<thead>
<tr>
<th>jX</th>
<th>Condition</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>jmp</td>
<td>1</td>
<td>Unconditional</td>
</tr>
<tr>
<td>je</td>
<td>ZF</td>
<td>Equal / Zero</td>
</tr>
<tr>
<td>jne</td>
<td>~ZF</td>
<td>Not Equal / Not Zero</td>
</tr>
<tr>
<td>jg</td>
<td>~SF</td>
<td>Greater (Signed)</td>
</tr>
<tr>
<td>jge</td>
<td>~SF^OF</td>
<td>Greater or Equal (Signed)</td>
</tr>
<tr>
<td>jl</td>
<td>(SF^OF)</td>
<td>Less (Signed)</td>
</tr>
<tr>
<td>jle</td>
<td>(SF^OF)</td>
<td>Less or Equal (Signed)</td>
</tr>
<tr>
<td>ja</td>
<td>~CF</td>
<td>Above (unsigned)</td>
</tr>
<tr>
<td>jb</td>
<td>CF</td>
<td>Below (unsigned)</td>
</tr>
</tbody>
</table>

Conditional Control

- In assembly/machine code, jump is pretty much all we got
- But there are a variety of conditional control in high-level languages
  - if ... else ...
  - switch(...){}
  - while loop
  - for loop
  - ... ...

Conditional Branch Example

```c
int absdiff(int x, int y)
{
    int result;
    if (x > y) {
        result = x - y;
    } else {
        result = y - x;
    }
    return result;
}
```

```assembly
absdiff:
    pushl %ebp
    movl %esp, %ebp
    movl 8(%ebp), %edx
    movl 12(%ebp), %eax
    cmpl %eax, %edx
    jle .L6
    subl %eax, %edx
    movl %edx, %eax
    jmp .L7
.L6:
    subl %edx, %eax
.L7:
    popl %ebp
    ret
```

Conditional Branch Example (Cont.)

```c
int goto_ad(int x, int y)
{
    int result;
    if (x <= y) goto Else;
    result = x - y;
    goto Exit;
Else:
    result = y - x;
Exit:
    return result;
}
```

```assembly
absdiff:
    pushl %ebp
    movl %esp, %ebp
    movl 8(%ebp), %edx
    movl 12(%ebp), %eax
    cmpl %eax, %edx
    jle .L6
    subl %edx, %eax
    jmp .L7
.L6:
    subl %edx, %eax
.L7:
    popl %ebp
    ret
```

Better resemblance to "goto"-style C code
- Bad coding style, but that’s not our concern here
Using Conditional Moves

- Conditional move instructions
  - if (Test) Dest ← Src
- Why?
  - Branches are very disruptive to instruction flow through pipelines
  - Cond-mov require no control transfer

C Code

```c
val = Test;
? Then_Expr : Else_Expr;
```

Cond-mov Version

```c
tval = Then_Expr;
result = Else_Expr;
t = Test;
if (t) result = tval;
return result;
```

Outline

- Arithmetic operations
- Control: Condition codes
- Conditional branches and moves
- Loops
- Switch

Loop Control

- In assembly/machine code, jump is pretty much all we got for conditional control (similar to "goto" in C)
- Convert typical loop control into goto-based C code.
  - Familiarize with assembly/machine programming
  - Understand compiler transformation
  - Able to read/comprehend assembly/machine code

“Do-While” Loop Example

C Code

```c
int pcount_do(unsigned x)
{
  int result = 0;
do {
    result += x & 0x1;
x >>= 1;
} while (x);
return result;
}
```

Goto Version

```c
int pcount_do(unsigned x)
{
  int result = 0;
loop:
  result += x & 0x1;
x >>= 1;
  if (x)
    goto loop;
return result;
}
```

- Count number of 1’s in argument x
- Use conditional branch to either continue looping or to exit loop
**“Do-While” Loop Compilation**

**Goto Version**

```c
int pcount_do(unsigned x) {
    int result = 0;
    loop:
        result += x & 0x1;
        x >>= 1;
        if (!x)
            goto loop;
    return result;
}
```

<table>
<thead>
<tr>
<th>Registers:</th>
</tr>
</thead>
<tbody>
<tr>
<td>%edx</td>
</tr>
<tr>
<td>x</td>
</tr>
<tr>
<td>%ecx</td>
</tr>
<tr>
<td>result</td>
</tr>
</tbody>
</table>

Goto Version

```c
movl $0, %ecx # result = 0
.L2: # loop:
    movl %edx, %eax
    andl $1, %eax # t = x & 1
    addl %eax, %ecx # result += t
    shrl %edx # x >>= 1
    jna .L2 # If 0, goto loop
```

**General “Do-While” Translation**

**C Code**

```c
do
    Body
    while (Test);
```

**Goto Version**

```c
loop:
    Body
    if (Test)
        goto loop
```

**General “While” Translation**

**While version**

```c
while (Test)
    Body
```

**Do-While Version**

```c
if (!Test)
    goto done;
do
    Body
while (Test);
done:
```

**Goto Version**

```c
if (!Test)
    goto done;
loop:
    Body
    if (Test)
        goto loop;
done:
```

**General “For” Translation**

**For Version**

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

**Goto Version**

```c
Init;
if (!Test)
    goto done;
loop:
    Body
    Update
    if (Test)
        goto loop;
done:
```

**Do-While Version**

```c
Init;
if (!Test)
    goto done;
do
    Body
    Update;
while (Test);
done:
```
“For” Loop Conversion Example

C Code

```c
#define WSIZE 8*sizeof(int)
int pcount_for(unsigned x) {
    int i;
    int result = 0;
    for (i = 0; i < WSIZE; i++) {
        unsigned mask = 1 << i;
        result += (x & mask) != 0;
    }
    return result;
}
```

Goto Version

```c
int pcount_for_gt(unsigned x) {
    int i;
    int result = 0;
    if (i < WSIZE) goto done;
    loop:
        unsigned mask = 1 << i;
        result += (x & mask) != 0;
        i++;
        if (i < WSIZE) goto loop;
    done:
    return result;
}
```

- Initial test can be optimized away

Outline

- Arithmetic operations
- Control: Condition codes
- Conditional branches and moves
- Loops
- Switch

Switch Statement

```c
long switch_eg (long x, long y, long z) {
    long w = 1;
    switch(x) {
    case 1:
        w = y*z;
        break;
    case 2:
        w = y/z;
        /* Fall Through */
    case 3:
        w = z;
        break;
    case 5:
    case 6:
        w = z;
        break;
    default:
        w = 2;
    }
    return w;
}
```

- An example
  - Multiple case labels (5 & 6)
  - Fall through case (2)
  - Missing case (4)

- How to implement switch statement in machine code?
  - Using jumps (or “goto”)
  - Jump table avoids sequencing through cases
    - Constant time, rather than linear

Jump Table Structure

```
```

<table>
<thead>
<tr>
<th>Switch Form</th>
<th>Jump Table</th>
<th>Jump Targets</th>
</tr>
</thead>
<tbody>
<tr>
<td>switch(x) {</td>
<td>jtab: Targ0</td>
<td>Targ0: Code Block 0</td>
</tr>
<tr>
<td>case val_0:</td>
<td>Targ1: Code Block 1</td>
<td></td>
</tr>
<tr>
<td>Block 0</td>
<td>Targ2: Code Block 2</td>
<td></td>
</tr>
<tr>
<td>case val_1:</td>
<td>Targn-1: Code Block n-1</td>
<td></td>
</tr>
<tr>
<td>Block 1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>* * *</td>
<td></td>
<td></td>
</tr>
<tr>
<td>case val_n-1:</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Block n-1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>}</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Approximate Translation

```c
target = jTab[x];
goto *target;
```
Switch Statement Example (IA32)

```c
long switch_eg(long x, long y, long z)
{
    long w = 1;
    switch(x) {
        case 1:      // .L3
            w = y*z;
            break;
        case 2:      // .L4
            w = y/z;
            /* Fall Through */
        case 3:      // .L5
            w += z;
            break;
        case 5: case 6:      // .L6
            w -= z;
            break;
        default:     // .L2
            w = 2;
    }
    return w;
}
```

**Indirect Jump Table**

```
.section .rodata
.align 4
.JT:
.long .L2 # x = 0
.long .L3 # x = 1
.long .L4 # x = 2
.long .L5 # x = 3
.long .L6 # x = 4
.long .L6 # x = 5
.long .L6 # x = 6
```

**Setup:**

```
switch_eg:
pushl %ebp  # Setup
movl %esp, %ebp  # Setup
movl 8(%ebp), %eax  # eax = x
cmpl $6, %eax  # Compare x:6
ja .L2  # If unsigned > goto default
jmp *.JT(,%eax,4)  # Goto *JT[x]
```

**Discover Jump Table in Machine Code**

1. First, find the address of the jump table
2. **Disassembled Object Code**

```
08048410 <switch_eg>:
   ... 8048419:77 07 ja 8048422 <switch_eg+0x12>
   804841b: f2 24 85 60 86 04 08 jmp *0x8048660(,%eax,4)
```

3. Then read into the jump table
4. But the table doesn’t show up in disassembled code
5. Can inspect using GDB: `x/7?qw 0x8048660`
   - Examine 7 hexadecimal format “words” (4-bytes each)
   - Use command “help x” to get format documentation

```
0x8048660: 0x08048422 0x08048432 0x0804843b 0x08048429
0x8048670: 0x08048422 0x0804844b 0x0804844b
```

Matching Disassembled Targets
Summary on Execution Control

- Control in high-level language
  - if-then-else
  - do-while
  - while, for
  - switch
- Control in machine code
  - Conditional jump
  - Conditional move
  - Indirect jump
  - Compiler generates code sequence to implement more complex control
- Standard techniques
  - Loops converted to go-to form
  - Large switch statements use jump tables
  - Sparse switch statements may use decision trees

Disclaimer

These slides were adapted from the CMU course slides provided along with the textbook of “Computer Systems: A programmer’s Perspective” by Bryant and O’Hallaron. The slides are intended for the sole purpose of teaching the computer organization course at the University of Rochester.