Machine-Level Programming II: Arithmetic & Control

Kai Shen

Outline

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

Some Arithmetic Operations

- Two Operand Instructions:
  - Format
  - Computation
  - addl  Src, Dest  Dest = Dest + Src
  - subl  Src, Dest  Dest = Dest – Src
  - imull  Src, Dest  Dest = Dest * Src
  - sall  Src, Dest  Dest = Dest << Src  Also called shll
  - sarl  Src, Dest  Dest = Dest >> Src  Arithmetic
  - shr1  Src, Dest  Dest = Dest >> Src  Logical
  - xorl  Src, Dest  Dest = Dest ^ Src
  - andl  Src, Dest  Dest = Dest & Src
  - orl  Src, Dest  Dest = Dest | Src
- Little distinction between signed and unsigned int (why?)

Some Arithmetic Operations

- One Operand Instructions
  - incl  Dest = Dest + 1
  - decl  Dest = Dest – 1
  - negl  Dest = – Dest
  - notl  Dest = ~Dest

- See book for more instructions
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;
}
```

Another Example

```c
int logical(int x, int y) {
    int t1 = x^y;
    int t2 = t1 >> 17;
    int mask = (1<<13) - 7;
    int rval = t2 & mask;
    return 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) 

General purpose registers

Current stack top

Current stack frame

Instruction pointer

Condition codes
### Condition Codes (Implicit Setting)

- Single bit registers
  - CF  Carry Flag (for unsigned)
  - SF  Sign Flag (for signed)
  - ZF  Zero Flag
  - OF  Overflow Flag (for signed)
- Implicitly set (think of it as side effect) by arithmetic operations
  - Example: `addl addq Src, Dest <-> 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 && b>0 && t<0) || (a<0 && b<0 && t<=0)`
- Not set by `lea` instruction

### Condition Codes (Setting by Compare)

- Explicit 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)
  - ZF set if `a == b`
  - SF set if `(a-b) < 0` (as signed)
  - OF set if two's-complement (signed) overflow
    - `(a>0 && b<0 && (a-b)<0) || (a<0 && b>0 && (a-b)>0)`

### Condition Codes (Setting by Test)

- Explicit setting by test instruction
  - `testl Src2, Src1`
  - `testl b, a` 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

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

```assembly
movl 12(%ebp), %eax # eax = y
cmpl %eax, 8(%ebp) # Compare x : y
setb %al # al = x > y
movb %al, %eax # Zero rest of %eax
```
**Reading Condition Codes**

- **SetX instructions**
  - Set single byte based on combinations of condition codes

<table>
<thead>
<tr>
<th>SetX</th>
<th>Condition</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>sete</td>
<td>ZF</td>
<td>Equal / Zero</td>
</tr>
<tr>
<td>setn</td>
<td>~ZF</td>
<td>Not Equal / Not Zero</td>
</tr>
<tr>
<td>seta</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><del>CF</del>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></td>
<td>Unconditional</td>
</tr>
<tr>
<td>jle</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>jae</td>
<td>SF</td>
<td>Negative</td>
</tr>
<tr>
<td>jns</td>
<td>~SF</td>
<td>Nonnegative</td>
</tr>
<tr>
<td>jg</td>
<td>~(SF^OF) &amp; ~ZF</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><del>CF</del>ZF</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(...) {}
### Conditional Branch Example

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

#### Setup
- `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`

#### Body1
- `subl %edx, %eax`

#### Body2a
- `subl %eax, %edx`

#### Body2b
- `subl %edx, %eax`

#### Finish
- `popl %ebp`
- `ret`

### Conditional Branch Example (Cont.)

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

#### Setup
- `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`

#### Body1
- `subl %edx, %eax`

#### Body2a
- `subl %eax, %edx`

#### Body2b
- `subl %edx, %eax`

#### Finish
- `popl %ebp`
- `ret`

### General Conditional Expression

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

#### Generalized to:
- `val = Test ? Then_Expr : Else_Expr;`

### 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
- Support in practice
  - Supported in post-1995 x86 processors
  - GCC doesn’t use them by default
    - Wants to preserve compatibility with ancient processors
  - Use switch -march=686 for IA32
  - Enabled for x86-64

### C Code

#### C Code

- `val = Test ? Then_Expr : Else_Expr;`

#### Cond-mov Version

- `tval = Then_Expr; result = Else_Expr; t = Test; if (t) result = tval; return result;`
Bad Cases for Conditional Move

Expensive computations

\[
\text{val} = \text{Test}(x) \ ? \ \text{Hard1}(x) : \text{Hard2}(x);
\]
- Both values get computed
- Only makes sense when computations are very simple

Risky computations

\[
\text{val} = p ? *p : 0;
\]
- Compute before conditional check

Computations with side effects

\[
\text{val} = x > 0 ? x*=7 : x+=3;
\]
- Both values get computed
- Must be side-effect free

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 x</td>
</tr>
<tr>
<td>%ecx result</td>
</tr>
</tbody>
</table>

```assembly
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

**Goto Version**

```c
C Code
    do
        Body
        while (Test);
    done;
```

**Goto Version**

```c
Goto Version
    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
}
```

**For 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);
```
“For” Loop Conversion Example

C Code

```c
#define WSIZE 8*sizeof(int)

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

- Initial test can be optimized away

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

```c
jump_table_structure:

<table>
<thead>
<tr>
<th>Switch Form</th>
<th>Jump Table</th>
</tr>
</thead>
<tbody>
<tr>
<td>switch(x) {</td>
<td></td>
</tr>
<tr>
<td>case val_0:</td>
<td></td>
</tr>
<tr>
<td>Block 0</td>
<td></td>
</tr>
<tr>
<td>• • •</td>
<td></td>
</tr>
<tr>
<td>case val_1:</td>
<td></td>
</tr>
<tr>
<td>Block 1</td>
<td></td>
</tr>
<tr>
<td>• • •</td>
<td></td>
</tr>
<tr>
<td>case val_n-1:</td>
<td></td>
</tr>
<tr>
<td>Block n-1</td>
<td></td>
</tr>
<tr>
<td>}</td>
<td></td>
</tr>
</tbody>
</table>
```

Jump Targets

- Targ0: Code Block 0
- Targ1: Code Block 1
- Targ2: Code Block 2
- Targn-1: Code Block n-1

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) {
    . . .
}    return w;
}
```

Jump Table

```c
section .rodata
.align 4
.L7:
.long .L2 # x = 0
.long .L3 # x = 1
.long .L4 # x = 2
.long .L5 # x = 3
.long .L2 # x = 4
.long .L6 # x = 5
.long .L6 # x = 6
```

Discover Jump Table in Machine Code

- First, find the address of the jump table
- Disassembled Object Code
```
08048410 <switch_eg>:
  . . .
80484419: 77 07 ja 8048422 <switch_eg+0x12>
804841b: ff 24 85 60 86 04 08 jmp 8048660(.l7,%.eax,4)
```
- Then read into the jump table
  - But the table doesn’t show up in disassembled code
  - Can inspect using GDB: `x/7xw 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

```c
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:    // .L6
        w -= z;
        break;
    default:    // .L2
        w = 2;
    }
```
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.