Machine-Level Programming II: Arithmetic & Control
15-213/18-243: Introduction to Computer Systems
6th Lecture, 31 January 2013

Instructors:
Anthony Rowe, Seth Goldstein and Gregory Kesden

The course that gives CMU its “Zip”!

Today
• Memory addressing modes
• Address computation (Lea)
• Arithmetic operations
• Control: Condition codes
• Conditional branches & moves
• Loops

Last Time: Machine Programming, Basics

<table>
<thead>
<tr>
<th>$eax</th>
<th>$ecx</th>
<th>$edx</th>
<th>$ebx</th>
<th>$esi</th>
<th>$edi</th>
<th>$esp</th>
<th>$ebp</th>
</tr>
</thead>
</table>

movl $0x4,%eax
movl %eax,%edx
movl (%eax),%edx

Complete Memory Addressing Modes

• Most General Form
  D(Rb,Ri,S)  Mem[Reg[Rb]+S*Reg[Ri]+ D]
  • D: Constant “displacement” 1, 2, or 4 bytes
  • Rb: Base register: Any of 8 integer registers
  • Ri: Index register: Any, except for %esp
  • S: Scale: 1, 2, 4, or 8 (why these numbers?)

• Special Cases
  (Rb,Ri)  Mem[Reg[Rb]+Reg[Ri]]
  D(Rb,Ri)  Mem[Reg[Rb]+Reg[Ri]+D]
  (Rb,Ri,S)  Mem[Reg[Rb]+S*Reg[Ri]]
Address Computation Examples

<table>
<thead>
<tr>
<th>Expression</th>
<th>Address Computation</th>
<th>Address</th>
</tr>
</thead>
<tbody>
<tr>
<td>%edx 0xf000</td>
<td></td>
<td></td>
</tr>
<tr>
<td>%ecx 0x0100</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- \(0x8(\%edx)\)  \(0xf000 + 0x8\)  \(0xf008\)
- \(\%edx,\%ecx\) \(0xf000 + 0x100\)  \(0xf100\)
- \(\%edx,\%ecx, 4\) \(0xf000 + 4 \times 0x100\)  \(0xf400\)
- \(0x80(\%edx,\%ecx, 2)\) \(2 \times 0xf000 + 0x80\)  \(0x1e080\)

Address Computation Instruction

- **leal Src, Dest**
  - Src is address mode expression
  - Set Dest to address denoted by expression

- **Uses**
  - Computing addresses without a memory reference
    - E.g., translation of \(p = \&x[i]\);
  - Computing arithmetic expressions of the form \(x + k \times y\)
    - \(k = 1, 2, 4,\) or 8

- **Example**

```c
int mul12(int x) {
  return x*12;
}
```

Converted to ASM by compiler:

```
leal (%eax,%eax,2), %eax  ;t <- x+x*2
sall $2, %eax             ;return t<<2
```

Today

- Complete addressing mode, address computation (leal)
- Arithmetic operations
- Control: Condition codes
- Conditional branches
- While loops

Some Arithmetic Operations

- **Two Operand Instructions:**
  - **Format**  **Computation**

<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>shrl</td>
<td>Dest = Dest &gt;&gt; Src</td>
</tr>
<tr>
<td>xorl</td>
<td>Dest = Dest ^ Src</td>
</tr>
<tr>
<td>andl</td>
<td>Dest = Dest &amp; Src</td>
</tr>
<tr>
<td>orl</td>
<td>Dest = Dest</td>
</tr>
</tbody>
</table>

- Watch out for argument order!
- No distinction between signed and unsigned int (why?)
Some Arithmetic Operations

- One Operand Instructions

  incl  Dest  Dest = Dest + 1
  decl  Dest  Dest = Dest - 1
  negl  Dest  Dest = - Dest
  notl  Dest  Dest = ~Dest

- See book for more instructions

Understanding `arith`

```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;
}
```

Arithmetic Expression 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;
}
```

### Stack

```
0  Old %ebp
4  Rtn Addr
8  %eax
12  y
16  z
```
Observations about arith

- Instructions in different order from C code
- Some expressions require multiple instructions
- Some instructions cover multiple expressions
- Get exact same code when compile:

```c
(int x+y+z)*(x+4+48*y)
```

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

```assembly
logical:
    pushl %ebp
    movl %ebp, %ebp  # Set Up
    movl 12(%ebp), %eax
    xorl $0(%ebp), %eax  # eax = x*y (t1)
    sarl $17,%eax  # eax = t1>>17 (t2)
    andl $8185,%eax  # eax = t2 & mask (rval)
    popl %ebp
    ret  # Finish
```

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;
}
```

```assembly
logical:
    pushl %ebp
    movl %ebp, %ebp  # Set Up
    movl 12(%ebp), %eax
    xorl $0(%ebp), %eax  # eax = x*y (t1)
    sarl $17,%eax  # eax = t1>>17 (t2)
    andl $8185,%eax  # eax = t2 & mask (rval)
    popl %ebp
    ret  # Finish
```
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;
}
```

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;
}
```

Today

- Complete addressing mode, address computation (leal)
- Arithmetic operations
- Control: Condition codes
- Conditional branches
- Loops

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
  - %eax
  - %ecx
  - %edx
  - %ebx
  - %esi
  - %edi
- Current stack top (%esp)
- Current stack frame (%ebp)
- Instruction pointer (%eip)
- Condition codes (CF, ZF, SF, OF)
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)
- Implicity 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
- Full documentation (IA32), link on course website

Condition Codes (Explicit Setting: Compare)

- Explicit setting by Compare instruction
  - `cmp/cmpq 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 (Explicit Setting: Test)

- Explicit setting by Test instruction
  - `testl/testq 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 combinations of condition codes

<table>
<thead>
<tr>
<th>CR</th>
<th>Condition</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>sete</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>seto</td>
<td>OF</td>
<td>Negative</td>
</tr>
<tr>
<td>setns</td>
<td>~SF</td>
<td>Non negative</td>
</tr>
<tr>
<td>setg</td>
<td>~(SF&amp;OF)</td>
<td>Greater (Signed)</td>
</tr>
<tr>
<td>setge</td>
<td>~(SF&amp;OF)</td>
<td>Greater or Equal (Signed)</td>
</tr>
<tr>
<td>setl</td>
<td>(SF&amp;OF)</td>
<td>Less (Signed)</td>
</tr>
<tr>
<td>setle</td>
<td>(SF&amp;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>
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 (long x, long y) {
    return x > y;
}
```

SetX Instructions: x86-64
- Set single byte based on combination of condition codes
- Does not alter remaining 3 bytes

```c
long lgt (long x, long y) {
    return x > y;
}
```

Today
- Complete addressing mode, address computation (leal)
- Arithmetic operations
- x86-64
- Control: Condition codes
- Conditional branches & Moves
- Loops

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>jw</td>
<td>=2F</td>
<td>Equal / Zero</td>
</tr>
<tr>
<td>jnw</td>
<td>=~2F</td>
<td>Not Equal / Not Zero</td>
</tr>
<tr>
<td>js</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)+~2F</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^~ZF</td>
<td>Above (unsigned)</td>
</tr>
<tr>
<td>jb</td>
<td>CF</td>
<td>Below (unsigned)</td>
</tr>
</tbody>
</table>
## Conditional Branch Example

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

### absdiff: Push Frame, Save EBP, Move EBP to ESP, Move 8(%ebp), %edx, Move 12(%ebp), %eax, Compare %eax, %edx, Jump if Less than or Equal to .L6, Subtract %eax, %edx, Move %edx, %eax, Jump .L7, Subtract %edx, %eax, Pop %ebp, Ret

### .L6: Subtract %edx, %eax, .L7: Subtract %edx, %eax, .L8: Push %ebp, Move %ebp, %ebp, Move 8(%ebp), %edx, Move 12(%ebp), %eax, Compare %eax, %edx, Jump if Less than or Equal to .L6, Subtract %eax, %edx, Move %edx, %eax, Jump .L7, Subtract %edx, %eax, Pop %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;
}
```

### absdiff: Push Frame, Save EBP, Move EBP to ESP, Move 8(%ebp), %edx, Move 12(%ebp), %eax, Compare %eax, %edx, Jump if Less than or Equal to .L6, Subtract %eax, %edx, Move %edx, %eax, Jump .L7, Subtract %edx, %eax, Pop %ebp, Ret

### .L6: Subtract %edx, %eax, .L7: Subtract %edx, %eax, .L8: Push %ebp, Move %ebp, %ebp, Move 8(%ebp), %edx, Move 12(%ebp), %eax, Compare %eax, %edx, Jump if Less than or Equal to .L6, Subtract %eax, %edx, Move %edx, %eax, Jump .L7, Subtract %edx, %eax, Pop %ebp, Ret

---

### C allows “goto” as means of transferring control
- Closer to machine-level programming style
- Generally considered bad coding style
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;
}
```

General Conditional Expression Translation

**C Code**

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

**Goto Version**

```
nt = !Test;
if (nt) goto Else;
val = Then_Expr;
else {
    result = y-x;
    return result;
}
```

Using Conditional Moves

- **Conditional Move Instructions**
  - Instruction supports:
    - if (Test) Dest ← Src
  - Supported in post-1995 x86 processors
  - GCC does not always use them
    - Wants to preserve compatibility with ancient processors
    - Enabled for x86-64
    - Use switch `-march=686` for IA32
  - **Why?**
    - Branches are very disruptive to instruction flow through pipelines
    - Conditional move do not require control transfer

Conditional Move Example: x86-64

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

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

```
x in %edi
movl %edi, %edx
subl %esi, %edx # tval = x-y
movl %esi, %eax
subl %edi, %eax # result = y-x
cmpl %esi, %edi # Compare x:y
cmovg %edx, %eax # If >, result = tval
ret
```
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

\[
\text{val} = \text{p} \ ? \ \text{p} : 0;
\]

- Both values get computed
- May have undesirable effects

Computations with side effects

\[
\text{val} = x > 0 \ ? \ x*=7 : x+=3;
\]

- Both values get computed
- Must be side-effect free

“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 (“popcount”)
- Use conditional branch to either continue looping or to exit loop

“Do-While” Loop Compilation

Goto Version

```asm
movl $0, %ecx  # result = 0  
.L2:           # loop:
    result += x & 0x1;  
    x >>= 1;  
    if (x)
        goto loop;  
    return result;  
```

- Registers:
  - %edx, %eax
  - $1, %eax  
  - t = x & 1
  - result += t
  - x >>= 1
  - jne .L2  # If !0, goto loop

Today

- Complete addressing mode, address computation (leal)
- Arithmetic operations
- x86-64
- Control: Condition codes
- Conditional branches and moves
- Loops
General “Do-While” Translation

C Code

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

Goto Version

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

- **Body:**
  ```c
  { Statement_1;
    Statement_2;
    ... Statement_n;
  }
  ```

- Test returns integer
  - = 0 interpreted as false
  - ≠ 0 interpreted as true

“While” Loop Example

C Code

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

Goto Version

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

- Is this code equivalent to the do-while version?

General “While” Translation

While version

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

Do-While Version

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

“For” Loop 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;
}
```

- Is this code equivalent to other versions?
"For" Loop Form

General Form

for (Init; Test; Update) Body

for (i = 0; i < WSIZE; i++) {
    unsigned mask = 1 << i;
    result += (x & mask) != 0;
}

"For" Loop → While Loop

For Version

for (Init; Test; Update)
    Body

While Version

Init:
while (Test) {
    Body
    Update;
}

"For" Loop → ... → Goto

For Version

for (Init; Test; Update)
    Body

While Version

Init;
while (Test) {
    Body
    Update;
}

Carnegie Mellon

"For" Loop Conversion Example

C Code

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

Goto Version

int pcount_for_gt(unsigned x) {
    int i = 0;
    int result = 0;
    if (!(i < WSIZE)) {Test
        goto loop;
    }
    while (Test) {
        Body
        Update;
        goto loop;
    }
    done:
}

- Initial test can be optimized away

Carnegie Mellon
Summary

- Today
  - Complete addressing mode, address computation (leal)
  - Arithmetic operations
  - Control: Condition codes
  - Conditional branches & conditional moves
  - Loops

- Next Time
  - Switch statements
  - Stack
  - Call / return
  - Procedure call discipline