Lecture 5: More machine operations

Last time:
Machine programming, basics

- History of Intel processors and architectures
- C, assembly, machine code
- Assembly (IA32):
  - Registers
  - Operands
  - Move (what’s the 1 in movl?)
    
    ```
    movl $0x4,%eax
    movl %eax,%edx
    movl (%eax),%edx
    ```

Today

- Complete addressing mode, address computation (**leal**)
- Arithmetic operations
- x86-64
- Control: Condition codes
- Conditional branches
- While loops

Complete memory addressing modes

- Most General Form:
  
  ```
  D(Rb,Ri,S) \quad 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
    
    — Unlikely you’d use \%ebp, either
  
  - **S:** Scale: 1, 2, 4, or 8 (**why these numbers?**)

  - Special Cases
    
    ```
    (Rb,Ri) \quad Mem[Reg[Rb]+Reg[Ri]]
    D(Rb,Ri) \quad Mem[Reg[Rb]+Reg[Ri]+D]
    (Rb,Ri,S) \quad 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>0x8(%edx)</td>
<td>0xf000 + 0x8</td>
<td>0xf008</td>
</tr>
<tr>
<td>(%edx,%ecx)</td>
<td>0xf000 + 0x100</td>
<td>0xf100</td>
</tr>
<tr>
<td>(%edx,%ecx,4)</td>
<td>0xf000 + 4*0x100</td>
<td>0xf400</td>
</tr>
<tr>
<td>0x80(%edx,2)</td>
<td>2*0xf000 + 0x80</td>
<td>0x1e080</td>
</tr>
</tbody>
</table>

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*y`
    
    — `k = 1, 2, 4, or 8`
Today

- Complete addressing mode, address computation (leal)
- Arithmetic operations
- x86-64
- Control: Condition codes
- Conditional branches
- While loops

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

Also called shll
Arithmetic
Logical

- No distinction between signed and unsigned int (why?)

Some arithmetic operations

- 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>decl</td>
<td>Dest ← Dest - 1</td>
</tr>
<tr>
<td>negl</td>
<td>Dest ← -Dest</td>
</tr>
<tr>
<td>notl</td>
<td>Dest ← ~Dest</td>
</tr>
</tbody>
</table>

- See book for more instructions

Using leal for arithmetic expressions

```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;
    int rval = t2 * t5;
    return rval;
}
```

```assembly
        pushl %ebp
        movl %esp,%ebp
        movl 8(%ebp),%eax
        movl 12(%ebp),%edx
        leal (%edx,%eax),%ecx
        leal (%edx,%edx,2),%edx
        sall $4,%edx
        addl 16(%ebp),%ecx
        leal 4(%edx,%eax),%eax
        imull %ecx,%eax
        movl %ebp,%esp
        popl %ebp
        ret
```

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
        pushl %ebp
        movl %esp,%ebp
        movl 8(%ebp),%eax
        xorl 12(%ebp),%eax
        leal 12(%ebp),%eax
        sarl $17,%eax
        andl $8185,%eax
        movl %ebp,%esp
        popl %ebp
        ret
```

```
2^{13} = 8192, 2^{13} - 7 = 8185
```
Today

• Complete addressing mode, address computation (lea1)
• Arithmetic operations
• x86-64
• Control: Condition codes
• Conditional branches
• While loops

Data representations:
ia32 and x86-64

• Sizes of C objects (in bytes)

<table>
<thead>
<tr>
<th>C data type</th>
<th>Typical 32-bit</th>
<th>ia32</th>
<th>Intel x86-64</th>
</tr>
</thead>
<tbody>
<tr>
<td>char</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>short</td>
<td>2</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>int</td>
<td>4</td>
<td>4</td>
<td>4</td>
</tr>
<tr>
<td>long</td>
<td>4</td>
<td>4</td>
<td>8</td>
</tr>
<tr>
<td>long long</td>
<td>8</td>
<td>8</td>
<td>8</td>
</tr>
<tr>
<td>float</td>
<td>4</td>
<td>4</td>
<td>4</td>
</tr>
<tr>
<td>double</td>
<td>8</td>
<td>8</td>
<td>8</td>
</tr>
<tr>
<td>long double</td>
<td>8</td>
<td>10/12</td>
<td>10/16</td>
</tr>
<tr>
<td>char *(or any other pointer)</td>
<td>4</td>
<td>4</td>
<td>8</td>
</tr>
</tbody>
</table>

x86-64 integer registers

- Extend existing registers. Add 8 new ones.
- Make %ebp/%rbp general purpose

Instructions

- Long word 1 (4 Bytes) ↔ Quad word q (8 Bytes)

- New instructions:
  - movl → movq
  - addl → addq
  - sall → salq
  - etc.

- 32-bit instructions that generate 32-bit results
  - Set higher order bits of destination register to 0
  - Example: addl

Swap in 32-bit mode

```c
void swap(int *xp, int *yp)
{
  int t0 = *xp;
  int t1 = *yp;
  *xp = t1;
  *yp = t0;
}
```

Swap in 64-bit Mode

```c
void swap(int *xp, int *yp)
{
  int t0 = *xp;
  int t1 = *yp;
  *xp = t1;
  *yp = t0;
}
```

- Operands passed in registers (why useful?)
  - First [xp] in %rdi, second [yp] in %rsi
  - 64-bit pointers
- No stack operations required
- 32-bit data
  - Data held in registers %eax and %edx
  - movl operation
Swap Long Ints in 64-bit Mode

```c
void swap_l(long int *xp, long int *yp)
{
    long int t0 = *xp;
    long int t1 = *yp;
    *xp = t1;
    *yp = t0;
}
```

- 64-bit data
  - Data held in registers `%rax` and `%rdx`
  - `movq` operation
  - “q” stands for quad-word

Today

- Complete addressing mode, address computation (`leal`)
- Arithmetic operations
- x86-64
- Control: Condition codes
- Conditional branches
- While 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
  - %esp
  - %ebp

- Current stack top
- Current stack frame
- Instruction pointer
- Condition codes
  - CF  Carry Flag (for unsigned)
  - ZF  Zero Flag
  - SF  Sign Flag (for signed)
  - 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
- Full documentation (ia32), link also on course website

Condition Codes (Explicit Setting: Compare)

- Explicit Setting by Compare Instruction
  - `cmp`/`cmpq` `Srch, Src`l
  - `cmp` l 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
  - `test`/`testq` `Srch, Src`l
  - `test` l b, a like computing a&b w/o setting destination

- Sets condition codes based on value of `Src1` & `Src2`
- Useful to have one of the operands be a mask

- CF 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>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>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)~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>ZF</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>

Reading Condition Codes (Cont.)

- **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;
}
```

Body

```c
movl 12(%ebp),%eax  # eax = y
cmpl %eax,8(%ebp)   # Compare x : y
movzbl %al,%eax     # Zero rest of %eax
```

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>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>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)~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>ZF</td>
</tr>
<tr>
<td>ja</td>
<td>~CF&amp;~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;
}
```

Body

```c
absdiff:
pushl %ebp
  movl %esp, %ebp
  movl 8(%ebp), %edx
  movl 12(%ebp), %eax
  cmpl %eax, %edx
  jle .L7
  subl %eax, %edx
  movl %edx, %eax
.L8:
  leave
  ret
.L7:
  subl %edx, %eax
  jmp .L8
```

Today

- Complete addressing mode, address computation (*lea*)
- Arithmetic operations
- x86-64
- Control: Condition codes
- Conditional branches
- While loops
Conditional Branch Example

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

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

Conditionals: x86-64

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

- Conditional move instruction
  - cmovC src, dest
  - Move value from src to dest if condition C holds
  - More efficient than conditional branching (simple control flow)
  - But overhead: both branches are evaluated

Today

- Complete addressing mode, address computation (leal)
- Arithmetic operations
- x86-64
- Control: Condition codes
- Conditional branches
- While loops

“Do-While” Loop Example

```c
int fact_do(int x) {
    int result = 1;
    do {
        result *= x;
        x = x-1;
    } while (x > 1);
    return result;
}
```

- Use backward branch to continue looping
- Only take branch when “while” condition holds

General Conditional Expression Translation

```c
C Code
val = Test ? Then-Expr : Else-Expr;
```

- Test is expression returning integer
  - = 0 interpreted as false
  - ≠ 0 interpreted as true

Goto Version

```c
nt = !Test;
if (nt) goto Else;
val = Then-Expr;
Done:
    ...
        Else:
            val = Else-Expr;
            goto Done;
```

- Create separate code regions for then & else expressions
- Execute appropriate one

```
General Form with Conditional Move

```c
C Code
val = Test ? Then-Expr : Else-Expr;
```

- Both values get computed
- Overwrite then-value with else-value if condition doesn’t hold
- Don’t use when:
  - Then or else expressions have side effects
  - Then and else expressions are too expensive

```
Goto Version

```c
nt = !Test;
if (nt) goto Else;
val = Then-Expr;
Done:
    ...
        Else:
            val = Else-Expr;
            goto Done;
```

- Create separate code regions for then & else expressions
- Execute appropriate one

```
"Do-While" Loop Example

```c
int fact_goto(int x) {
    int result = 1;
    loop:
        result *= x;
        x = x-1;
    if (x > 1) goto loop;
    return result;
}
```

- Use backward branch to continue looping
- Only take branch when “while” condition holds
“Do-While” Loop Compilation

Goto Version

```c
int fact_goto(int x)
{
    int result = 1;

    loop:
        result *= x;
        x = x - 1;
        if (x > 1)
            goto loop;
    return result;
}
```

Assembly

```assembly
fact_goto:
    pushl %ebp # Setup
    movl %esp,%ebp # Setup
    movl $1,%eax # eax = 1
    movl 8(%ebp),%edx # edx = x
.
.L11:
    imull %edx,%eax # result *= x
    decl %edx # x--
    cmpl $1,%edx # Compare x : 1
    jg .L11 # if > goto loop
    movl %ebp,%esp # Finish
    popl %ebp # Finish
    ret # Finish
```

General “Do-While” Translation

C Code

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

Goto Version

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

“While” Loop Example

C Code

```c
int fact_while(int x)
{
    int result = 1;
    while (x > 1) {
        result *= x;
        x = x - 1;
    }
    return result;
}
```

Goto Version #1

```c
int fact_while_goto(int x)
{
    int result = 1;
    loop:
        if (!(x > 1))
            goto done;
        result *= x;
        x = x - 1;
    goto loop;
    done:
    return result;
}
```

Is this code equivalent to the do-while version?
- Must jump out of loop if test fails

Alternative “While” Loop Translation

C Code

```c
int fact_while(int x)
{
    int result = 1;
    while (x > 1) {
        result *= x;
        x = x - 1;
    }
    return result;
}
```

Goto Version #2

```c
int fact_while_goto2(int x)
{
    int result = 1;
    if (!(x > 1))
        goto done;
    loop:
        result *= x;
        x = x - 1;
    if (x > 1)
        goto loop;
    done:
    return result;
}
```

- Historically used by GCC
- Uses same inner loop as do-while version
- Guards loop entry with extra test

New Style “While” Loop Translation

C Code

```c
int fact_while(int x)
{
    int result = 1;
    while (x > 1) {
        result *= x;
        x = x - 1;
    }
    return result;
}
```

Goto Version

```c
int fact_while_goto3(int x)
{
    int result = 1;
    if (!(x > 1))
        goto middle;
    loop:
        result *= x;
        x = x - 1;
    if (x > 1)
        goto loop;
    middle:
    return result;
}
```

- Recent technique for GCC
  - Both IA32 & x86-64
  - First iteration jumps over body computation within loop
Jump-to-Middle While Translation

C Code

```
while (Test)
    Body
```

- Avoids duplicating test code
- Unconditional goto incurs no performance penalty
- for loops compiled in similar fashion

Goto Version

```
goto middle;
loop:         # Body
middle:
    if (Test)
        goto loop;
```

Goto (Previous) Version

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

Jump-to-Middle Example

```
int fact_while(int x)
{
    int result = 1;
    while (x > 1) {
        result *= x;
        x--;
    }
    return result;
}
```

```
# x in %edx, result in %eax
jmp .L34      # goto Middle
.L35:         # Loop:
imull %edx, %eax # result *= x
decl %edx      # x--
.L34:         # Middle:
cmpl $1, %edx  # x:1
jg .L35       # if >, goto Loop
```

Implementing Loops

- IA32
  - All loops translated into form based on “do-while”
- x86-64
  - Also make use of “jump to middle”
- Why the difference
  - IA32 compiler developed for machine where all operations costly
  - x86-64 compiler developed for machine where unconditional branches incur (almost) no overhead

Next time

- For loops
- Switch statements
- Procedures