The Hardware/Software Interface
CSE351 Spring 2015

Lecture 8

Instructor:
Katelin Bailey

Teaching Assistants:
Announcements

• Non-majors: would access to the labs be useful?
• Midterm
  • Quick poll: when works for a midterm review session?
  • Anyone also enrolled in 333?
• Homework 1
  • Due today!
  • Floating point question: ungraded
  • Negatives on question 2
• Lab 2
  • Out today (after class)!
  • Due next Friday
• First annual CSE women's day_mini-Grace Hopper (May 9)
Roadmap

C:
car *c = malloc(sizeof(car));
c->miles = 100;
c->gals = 17;
float mpg = get_mpg(c);
free(c);

Java:
Car c = new Car();
c.setMiles(100);
c.setGals(17);
float mpg = c.getMPG();

Assembly language:
get_mpg:
pushq %rbp
movq %rsp, %rbp
... popq %rbp
ret

Machine code:
0111010000011000
100011010000010000000010
1000100111000010
110000011111101000011111

Computer system:

› Memory, data, & addressing
› Integers & floats
› Machine code & C
› x86 assembly
› Procedures & stacks
› Arrays & structs
› Memory & caches
› Processes
› Virtual memory
› Memory allocation
› Java vs. C

OS:

Windows 8
Mac
Today

• **Review**
  • Move instructions, registers, and operands
  • Memory addressing modes

• **x86 Arithmetic operations**

• **x86 Control Flow**
  • Condition Codes
  • Conditional and Unconditional Branches
  • Loops
Three Basic Kinds of Instructions (review)

- Transfer data between memory and register
  - **Load** data from memory into register
    - \( \% \text{reg} = \text{Mem}[\text{address}] \)
  - **Store** register data into memory
    - \( \text{Mem}[\text{address}] = \% \text{reg} \)

- Perform arithmetic function on register or memory data
  - \( c = a + b; \quad z = x \ll y; i = h \& g; \)

- Transfer control: what instruction to execute next
  - Unconditional jumps to/from procedures
  - Conditional branches

Remember:
memory is indexed just like an array[] of bytes!
Moving Data: IA32 (review)

- **Moving Data**
  - \texttt{movl Source, Dest:}

- **Operand Types**
  - **Immediate**: Constant integer data
    - Example: $0\times400$, $-533$
    - Like C constant, but prefixed with ‘$’
    - Encoded with 1, 2, or 4 bytes
  - **Register**: One of 8 integer registers
    - Example: \%eax, \%edx
    - But \%esp and \%ebp reserved for special use
    - Others have special uses for particular instructions
  - **Memory**: 4 consecutive bytes of memory at address given by register
    - Simplest example: (\%eax)
    - Various other “address modes”
### movl Operand Combinations (review)

<table>
<thead>
<tr>
<th>Source</th>
<th>Dest</th>
<th>Src,Dest</th>
<th>C Analog</th>
</tr>
</thead>
<tbody>
<tr>
<td>Imm</td>
<td>Reg</td>
<td>movl $0x4,%eax</td>
<td>var_a = 0x4;</td>
</tr>
<tr>
<td></td>
<td>Mem</td>
<td>movl $-147,(%eax)</td>
<td>*p_a = -147;</td>
</tr>
<tr>
<td>Reg</td>
<td>Reg</td>
<td>movl %eax,%edx</td>
<td>var_d = var_a;</td>
</tr>
<tr>
<td></td>
<td>Mem</td>
<td>movl %eax,(%edx)</td>
<td>*p_d = var_a;</td>
</tr>
<tr>
<td>Mem</td>
<td>Reg</td>
<td>movl (%eax),%edx</td>
<td>var_d = *p_a;</td>
</tr>
</tbody>
</table>

Cannot do memory-memory transfer with a single instruction. How would you do it?
Complete Memory Addressing Modes (review)

- Remember, the addresses used for accessing memory in mov (and other) instructions can be computed in several different ways

- Most General Form:
  \[ D(Rb,Ri,S) \text{ Mem}[\text{Reg}[Rb] + S*\text{Reg}[Ri] + D] \]
  - \( D \): Constant “displacement” value represented in 1, 2, or 4 bytes
  - \( Rb \): Base register: Any of the 8/16 integer registers
  - \( Ri \): Index register: Any, except for %esp or %rsp; %ebp unlikely
  - \( S \): Scale: 1, 2, 4, or 8 (why these numbers?)

- Special Cases: can use any combination of \( D, Rb, Ri \) and \( S \)
  - \( (Rb,Ri) \text{ Mem}[\text{Reg}[Rb]+\text{Reg}[Ri]] \) (\( S=1, D=0 \))
  - \( D(Rb,Ri) \text{ Mem}[\text{Reg}[Rb]+\text{Reg}[Ri]+D] \) (\( S=1 \))
  - \( (Rb,Ri,S) \text{ Mem}[\text{Reg}[Rb]+S*\text{Reg}[Ri]] \) (\( D=0 \))
Address Computation Examples (review)

<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 (review)

- **leal** $Src,Dest$
  - $Src$ is address mode expression
  - Set $Dest$ to address computed by expression
    - (lea stands for load effective address)
  - Example: `leal (%edx,%ecx,4), %eax`

- **Uses**
  - Computing addresses without a memory reference
    - E.g., translation of $p = \&x[i]$;
  - Computing arithmetic expressions of the form $x + k*i$
    - $k = 1, 2, 4, \text{ or } 8$
Today

• Review
  • Move instructions, registers, and operands
  • Memory addressing modes
• **x86 Arithmetic operations**
• **x86 Control Flow**
  • Condition Codes
  • Conditional and Unconditional Branches
  • Loops
### Some Arithmetic Operations

**Two Operand (Binary) Instructions:**

<table>
<thead>
<tr>
<th>Format</th>
<th>Computation</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>addl</code></td>
<td><code>Dest = Dest + Src</code></td>
</tr>
<tr>
<td><code>subl</code></td>
<td><code>Dest = Dest - Src</code></td>
</tr>
<tr>
<td><code>imull</code></td>
<td><code>Dest = Dest * Src</code></td>
</tr>
<tr>
<td><code>shll</code></td>
<td><code>Dest = Dest &lt;&lt;&lt; Src</code> <strong>Also called sall</strong></td>
</tr>
<tr>
<td><code>sarl</code></td>
<td><code>Dest = Dest &gt;&gt;&gt; Src</code> <strong>Arithmetic</strong></td>
</tr>
<tr>
<td><code>shrl</code></td>
<td><code>Dest = Dest &gt;&gt;&gt; Src</code> <strong>Logical</strong></td>
</tr>
<tr>
<td><code>xorl</code></td>
<td><code>Dest = Dest ^ Src</code></td>
</tr>
<tr>
<td><code>andl</code></td>
<td><code>Dest = Dest &amp; Src</code></td>
</tr>
<tr>
<td><code>orl</code></td>
<td>`Dest = Dest</td>
</tr>
</tbody>
</table>

- **Watch out for argument order!** (especially `subl`)
- **No distinction between signed and unsigned int (why?)**
  - except arithmetic vs. logical shift right
Some Arithmetic Operations

• One Operand (Unary) Instructions

- **incl** Dest  \( \text{Dest} = \text{Dest} + 1 \)  increment
- **decl** Dest  \( \text{Dest} = \text{Dest} - 1 \)  decrement
- **negl** Dest  \( \text{Dest} = -\text{Dest} \)  negate
- **notl** Dest  \( \text{Dest} = \sim\text{Dest} \)  bitwise complement

• See textbook section 3.5.5 for more instructions: mull, cltd, idivl, divl
Using `leal` for Arithmetic Expressions (IA32)

```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;
}
```

```
arith:
    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
```

Set Up

Body

Finish
Understanding \texttt{arith} (IA32)

```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;
}
```

movl 8(%ebp),%eax  # eax = x
movl 12(%ebp),%edx  # edx = y
leal (%edx,%eax),%ecx  # ecx = x+y (t1)
leal (%edx,%edx,2),%edx  # edx = y + 2*y = 3*y
sal $4,%edx  # edx = 48*y (t4)
addl 16(%ebp),%ecx  # ecx = z+t1 (t2)
leal 4(%edx,%eax),%eax  # eax = 4+t4+x (t5)
imull %ecx,%eax  # eax = t5*t2 (rval)
Understanding `arith` (IA32)

```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;
}
```

```
movl 8(%ebp),%eax  # eax = x
movl 12(%ebp),%edx  # edx = y
leal (%edx,%eax),%ecx  # ecx = x+y (t1)
leal (%edx,%edx,2),%edx  # edx = y + 2*y = 3*y
sall $4,%edx  # edx = 48*y (t4)
addl 16(%ebp),%ecx  # ecx = z+t1 (t2)
leal 4(%edx,%eax),%eax  # eax = 4+t4+x (t5)
imull %ecx,%eax  # eax = t5*t2 (rval)
```
int arith (IA32)

```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

- Offset 0: Old %ebp
- Offset 4: Rtn adr
- Offset 8: x
- Offset 12: y
- Offset 16: z

#### Assembly Code

```
movl 8(%ebp),%eax  # eax = x
movl 12(%ebp),%edx  # edx = y
leal (%edx,%eax),%ecx  # ecx = x+y  (t1)
leal (%edx,%edx,2),%edx  # edx = y + 2*y = 3*y
sal $4,%edx  # edx = 48*y  (t4)
addl 16(%ebp),%ecx  # ecx = z+t1  (t2)
leal 4(%edx,%eax),%eax  # eax = 4+t4+x  (t5)
imull %ecx,%eax  # eax = t5*t2  (rval)
```
Understanding `arith` (IA32)

```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),%eax  # eax = x
movl 12(%ebp),%edx  # edx = y
lea (%edx,%eax),%ecx  # ecx = x+y  (t1)
lea (%edx,%edx,2),%edx  # edx = y + 2*y = 3*y
sall $4,%edx  # edx = 48*y  (t4)
addl 16(%ebp),%ecx  # ecx = z+t1  (t2)
lea 4(%edx,%eax),%eax  # eax = 4+t4+x  (t5)
imull %ecx,%eax  # eax = t5*t2  (rval)
```
Observations about `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;
}
```

- Instructions in different order from C code
- Some expressions require multiple instructions
- Some instructions cover multiple expressions
- Get exact same code when compile:
  - `(x + y + z) * (x + 4 + 48 * y)`

---

movl 8(%ebp),%eax  # eax = x
movl 12(%ebp),%edx  # edx = y
leal (%edx,%eax),%ecx  # ecx = x+y (t1)
leal (%edx,%edx,2),%edx  # edx = y + 2*y = 3*y
sal $4,%edx  # edx = 48*y (t4)
addl 16(%ebp),%ecx  # ecx = z+t1 (t2)
leal 4(%edx,%eax),%eax  # eax = 4+t4+x (t5)
imull %ecx,%eax  # eax = t5*t2 (rval)
Another Example (IA32)

```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;
}
```

### Logical Function

- **Set Up**
  - `pushl %ebp`
  - `movl %esp,%ebp`

- **Body**
  - `movl 8(%ebp),%eax`
  - `xorl 12(%ebp),%eax`
  - `sarl $17,%eax`
  - `andl $8185,%eax`

- **Finish**
  - `movl %ebp,%esp`
  - `popl %ebp`
  - `ret`
Another Example (IA32)

```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;
}
```

**logical:**
- **Set Up**
  - `pushl %ebp`
  - `movl %esp,%ebp`
- **Body**
  - `movl 8(%ebp),%eax`
  - `xorl 12(%ebp),%eax`
  - `sarl $17,%eax`
  - `andl $8185,%eax`
- **Finish**
  - `movl %ebp,%esp`
  - `popl %ebp`
  - `ret`

**Example Assembly Code:**
- `movl 8(%ebp),%eax`  
  - `eax = x`
- `xorl 12(%ebp),%eax`  
  - `eax = x^y`  
  - `(t1)`
- `sarl $17,%eax`  
  - `eax = t1>>17`  
  - `(t2)`
- `andl $8185,%eax`  
  - `eax = t2 & 8185`
Another Example (IA32)

```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;
}
```

documented:

```
movl 8(%ebp),%eax  eax = x
xorl 12(%ebp),%eax  eax = x^y  (t1)
sarl $17,%eax  eax = t1>>17  (t2)
andl $8185,%eax  eax = t2 & 8185
```
Another Example (IA32)

```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;
}
```

```
2^{\text{13}} = 8192, \quad 2^{\text{13}} - 7 = 8185
...0010000000000000, \ldots001111111111001
```

**Compiler Optimization**
- **Set Up**
  - `pushl %ebp`
  - `movl %esp,%ebp`
- **Body**
  - `movl 8(%ebp),%eax`
  - `xorl 12(%ebp),%eax`
  - `sarl $17,%eax`
  - `andl $8185,%eax`
- **Finish**
  - `movl %ebp,%esp`
  - `popl %ebp`
  - `ret`

```
movl 8(%ebp),%eax  \quad \text{eax} = x
xorl 12(%ebp),%eax  \quad \text{eax} = x^y \quad (t1)
sarl $17,%eax  \quad \text{eax} = t1\gg\!17 \quad (t2)
andl $8185,%eax  \quad \text{eax} = t2 \& 8185
```
Today

• Review
  • Move instructions, registers, and operands
  • Memory addressing modes
• x86 Arithmetic operations
• x86 Control Flow
  • Condition Codes
  • Conditional and Unconditional Branches
  • Loops
Jumping

- **jX Instructions**
  - Jump to different part of code depending on condition codes
  - Takes address as argument

<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>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)&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>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>
Conditionals and Control Flow

• A conditional branch is sufficient to implement most control flow constructs offered in higher level languages
  • if (condition) then {...} else {...}
  • while (condition) {...}
  • do {...} while (condition)
  • for (initialization; condition; iterative) {...}

• Unconditional branches implement some related control flow constructs
  • break, continue

• In x86, we’ll refer to branches as “jumps” (either conditional or unconditional)
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: %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)
  
- **Implicitly set (think of it as side effect) by arithmetic operations**
  
  - **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 (beware!)**
  
- **Full documentation (IA32):** [http://www.jegerlehner.ch/intel/IntelCodeTable.pdf](http://www.jegerlehner.ch/intel/IntelCodeTable.pdf)

Example: \( \text{addl/addq Src, Dest} \leftrightarrow t = a+b \)
Condition Codes (Explicit Setting: Compare)

• Single-bit registers
  
  | CF  | Carry Flag (for unsigned) | SF  | Sign Flag (for signed) |
  | ZF  | Zero Flag                | OF  | Overflow Flag (for signed) |

• Explicit Setting by Compare Instruction

```
cmpl/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)

- **Single-bit registers**
  
<table>
<thead>
<tr>
<th>Register</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>CF</td>
<td>Carry Flag (for unsigned)</td>
</tr>
<tr>
<td>SF</td>
<td>Sign Flag (for signed)</td>
</tr>
<tr>
<td>ZF</td>
<td>Zero Flag</td>
</tr>
<tr>
<td>OF</td>
<td>Overflow Flag (for signed)</td>
</tr>
</tbody>
</table>

- **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** if a&b == 0
  - **SF set** if a&b < 0
  
  testl %eax, %eax
  - Sets SF and ZF, check if eax is +,0,-
  ```
## Reading Condition Codes

- **SetX Instructions**
  - Set a single byte to 0 or 1 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>sets</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>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>
SetX Instructions:
Set single byte to 0 or 1 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: y at 12(%ebp), x at 8(%ebp)

```
movl 12(%ebp),%eax
cmpl %eax,8(%ebp)
setg %al
movzbl %al,%eax
```

What does each of these instructions do?
Reading Condition Codes (Cont.)

- **SetX Instructions:**
  Set single byte to 0 or 1 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:** y at 12(%ebp), x at 8(%ebp)

```assembly
movl 12(%ebp),%eax      # eax = y
cmpl %eax,8(%ebp)       # Compare x and y
setg %al                # al = x > y
movzbl %al,%eax         # Zero rest of %eax
```

(x - y)
Today

• Review
  • Move instructions, registers, and operands
  • Memory addressing modes
• x86 Arithmetic operations
• x86 Control Flow
  • Condition Codes
  • Conditional and Unconditional Branches
  • Loops
Jumping

- **jX Instructions**
  - Jump to different part of code depending on condition codes
  - Takes address as argument

<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>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) &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>~CF&amp;~ZF</td>
<td>Above (unsigned)</td>
</tr>
<tr>
<td>jb</td>
<td>CF</td>
<td>Below (unsigned)</td>
</tr>
</tbody>
</table>
int absdiff(int x, int y) {
    int result;
    if (x > y) {
        result = x-y;
    } else {
        result = y-x;
    }
    return result;
}

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

Body1

Body2

Setup

Finish
C allows “goto” as means of transferring control
  
  - Closer to machine-level programming style
  
  - Generally considered bad coding style
I could restructure
the program's flow.

OR use one little 'goto' instead.

Eh, screw good practice.
How bad can it be?

goto main_sub3;

*compile*

Dinosaur sitting on a desk.

Dinosaur:
I could restructure
the program's flow.

OR use one little 'goto' instead.

Eh, screw good practice.
How bad can it be?

goto main_sub3;

*compile*

Dinosaur:
I could restructure
the program's flow.

OR use one little 'goto' instead.

Eh, screw good practice.
How bad can it be?

goto main_sub3;

*compile*

Dinosaur:
I could restructure
the program's flow.

OR use one little 'goto' instead.

Eh, screw good practice.
How bad can it be?

goto main_sub3;

*compile*

Dinosaur:
I could restructure
the program's flow.

OR use one little 'goto' instead.

Eh, screw good practice.
How bad can it be?

goto main_sub3;

*compile*

Dinosaur:
I could restructure
the program's flow.

OR use one little 'goto' instead.

Eh, screw good practice.
How bad can it be?

goto main_sub3;

*compile*

Dinosaur:
I could restructure
the program's flow.

OR use one little 'goto' instead.

Eh, screw good practice.
How bad can it be?

goto main_sub3;

*compile*

Dinosaur:
I could restructure
the program's flow.

OR use one little 'goto' instead.

Eh, screw good practice.
How bad can it be?

goto main_sub3;

*compile*

Dinosaur:
I could restructure
the program's flow.

OR use one little 'goto' instead.

Eh, screw good practice.
How bad can it be?

goto main_sub3;

*compile*

Dinosaur:
I could restructure
the program's flow.

OR use one little 'goto' instead.

Eh, screw good practice.
How bad can it be?

goto main_sub3;

*compile*

Dinosaur:
I could restructure
the program's flow.

OR use one little 'goto' instead.

Eh, screw good practice.
How bad can it be?

goto main_sub3;

*compile*

Dinosaur:
I could restructure
the program's flow.

OR use one little 'goto' instead.

Eh, screw good practice.
How bad can it be?

goto main_sub3;

*compile*

Dinosaur:
I could restructure
the program's flow.

OR use one little 'goto' instead.

Eh, screw good practice.
How bad can it be?

goto main_sub3;

*compile*

Dinosaur:
I could restructure
the program's flow.

OR use one little 'goto' instead.

Eh, screw good practice.
How bad can it be?

goto main_sub3;

*compile*

Dinosaur:
I could restructure
the program's flow.

OR use one little 'goto' instead.

Eh, screw good practice.
How bad can it be?

goto main_sub3;

*compile*

Dinosaur:
I could restructure
the program's flow.

OR use one little 'goto' instead.

Eh, screw good practice.
How bad can it be?

goto main_sub3;

*compile*

Dinosaur:
I could restructure
the program's flow.

OR use one little 'goto' instead.

Eh, screw good practice.
How bad can it be?

goto main_sub3;

*compile*

Dinosaur:
I could restructure
the program's flow.

OR use one little 'goto' instead.

Eh, screw good practice.
How bad can it be?

goto main_sub3;

*compile*

Dinosaur:
I could restructure
the program's flow.

OR use one little 'goto' instead.

Eh, screw good practice.
How bad can it be?

goto main_sub3;

*compile*

Dinosaur:
I could restructure
the program's flow.

OR use one little 'goto' instead.

Eh, screw good practice.
How bad can it be?

goto main_sub3;

*compile*

Dinosaur:
I could restructure
the program's flow.

OR use one little 'goto' instead.

Eh, screw good practice.
How bad can it be?

goto main_sub3;

*compile*

Dinosaur:
I could restructure
the program's flow.

OR use one little 'goto' instead.

Eh, screw good practice.
How bad can it be?

goto main_sub3;

*compile*

Dinosaur:
I could restructure
the program's flow.

OR use one little 'goto' instead.

Eh, screw good practice.
How bad can it be?

goto main_sub3;

*compile*

Dinosaur:
I could restructure
the program's flow.

OR use one little 'goto' instead.

Eh, screw good practice.
How bad can it be?

goto main_sub3;

*compile*

Dinosaur:
I could restructure
the program's flow.

OR use one little 'goto' instead.

Eh, screw good practice.
How bad can it be?

goto main_sub3;

*compile*

Dinosaur:
I could restructure
the program's flow.

OR use one little 'goto' instead.

Eh, screw good practice.
How bad can it be?

goto main_sub3;

*compile*

Dinosaur:
I could restructure
the program's flow.

OR use one little 'goto' instead.

Eh, screw good practice.
How bad can it be?

goto main_sub3;

*compile*

Dinosaur:
I could restructure
the program's flow.

OR use one little 'goto' instead.

Eh, screw good practice.
How bad can it be?

goto main_sub3;

*compile*

Dinosaur:
I could restructure
the program's flow.

OR use one little 'goto' instead.

Eh, screw good practice.
How bad can it be?

goto main_sub3;

*compile*

Dinosaur:
I could restructure
the program's flow.

OR use one little 'goto' instead.

Eh, screw good practice.
How bad can it be?

goto main_sub3;

*compile*

Dinosaur:
I could restructure
the program's flow.

OR use one little 'goto' instead.

Eh, screw good practice.
How bad can it be?

goto main_sub3;

*compile*

Dinosaur:
I could restructure
the program's flow.

OR use one little 'goto' instead.

Eh, screw good practice.
How bad can it be?

goto main_sub3;

*compile*

Dinosaur:
I could restructure
the program's flow.

OR use one little 'goto' instead.

Eh, screw good practice.
How bad can it be?

goto main_sub3;

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

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

int x  %edx
int y  %eax

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

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

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

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

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

int x %edx
int y %eax

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
General Conditional Expression Translation

C Code

val = Test ? Then-Expr : Else-Expr;
result = x>y ? x-y : y-x;

Goto Version

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

if (Test)
val = Then-Expr;
else
val = Else-Expr;

• Test is expression returning integer
• = 0 interpreted as false
• ≠0 interpreted as true
• Create separate code regions for then & else expressions
• Execute appropriate one
• How might you make this more efficient?
Conditionals: x86-64

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

```
absdiff: # x in %edi, y in %esi
movl %edi, %eax  # eax = x
movl %esi, %edx  # edx = y
subl %esi, %eax  # eax = x-y
subl %edi, %edx  # edx = y-x
cmpl %esi, %edi  # x:y
cmovle %edx, %eax  # eax=edx if <=
ret
```

- **Conditional move instruction**
  - `cmovC src, dest`
  - Move value from src to dest if condition C holds
  - **Why is this good?**
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

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

```assembly
absdiff: # x in %edi, y in %esi
    movl %edi, %eax  # eax = x
    movl %esi, %edx  # edx = y
    subl %esi, %eax  # eax = x-y
    subl %edi, %edx  # edx = y-x
    cmpl %esi, %edi  # x:y
    cmovle %edx, %eax  # eax=edx if <=
    ret
```

Conditionals: x86-64
PC Relative Addressing

- PC relative branches are relocatable
- Absolute branches are not

```
0x100     cmp    r2, r3    0x1000
0x102     je     0x70     0x1002
0x104     ...     ...     0x1004
...       ...     ...     ...
0x172     add    r3, r4    0x1072
```
Today

• Review
  • Move instructions, registers, and operands
  • Memory addressing modes
• x86 Arithmetic operations
• x86 Control Flow
  • Condition Codes
  • Conditional and Unconditional Branches
• Loops
How to compile other loops should be straightforward

- The only slightly tricky part is to be sure where the conditional branch occurs: top or bottom of the loop

- How would `for(i=0; i<100; i++)` be implemented?

C/Java code:

```java
while ( sum != 0 ) {
    <loop body>
}
```

Machine code:

```assembly
loopTop:    cmpl  $0, %eax
    je    loopDone
    <loop body code>
    jmp    loopTop
loopDone:
```
“Do-While” Loop Example

C Code

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

Goto Version

```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
    movl %esp,%ebp
    movl $1,%eax
    movl 8(%ebp),%edx

.L11:
    imull %edx,%eax
    decl %edx
    cmpl $1,%edx
    jg .L11

    movl %ebp,%esp
    popl %ebp
    ret
```

**Registers:**

- `%edx`: `x`
- `%eax`: `result`
**“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

```
fact_goto:
    pushl %ebp
    movl %esp,%ebp
    movl $1,%eax
    movl 8(%ebp),%edx

.L11:
    imull %edx,%eax
    decl %edx
    cmpl $1,%edx
    jg .L11
    movl %ebp,%esp
    popl %ebp
    ret
```

---

**Registers:**

- `%edx` : `x`
- `%eax` : `result`

---

# Setup
# Setup
# eax = 1
# edx = x
# result *= x
# x--
# Compare x : 1
# if > goto loop
# Finish
# Finish
# Finish
General “Do-While” Translation

<table>
<thead>
<tr>
<th>C Code</th>
<th>Goto Version</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>do</code></td>
<td><code>loop:</code></td>
</tr>
<tr>
<td><code>  Body</code></td>
<td><code>  Body</code></td>
</tr>
<tr>
<td><code>while (Test);</code></td>
<td><code>if (Test)</code></td>
</tr>
<tr>
<td></td>
<td><code>  goto loop</code></td>
</tr>
</tbody>
</table>

- **Body:**
  ```
  
  ```

- **Test returns integer**
  - `= 0` interpreted as false
  - `≠ 0` interpreted as true
"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_goto(int x)
{
    int result = 1;
    goto middle;
    loop:
        result *= x;
        x = x-1;
    middle:
        if (x > 1)
            goto loop;
    return result;
}
```

- Used by GCC for both IA32 & x86-64
- First iteration jumps over body computation within loop straight to test
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
“For” Loop Example: Square-and-Multiply

```c
/* Compute x raised to nonnegative power p */
int ipwr_for(int x, unsigned int p)
{
    int result;
    for (result = 1; p != 0; p = p>>1) {
        if (p & 0x1)
            result *= x;
        x = x*x;
    }
    return result;
}
```

• **Algorithm**
  
  • Exploit bit representation: \( p = p_0 + 2p_1 + 2^2p_2 + \ldots + 2^{n-1}p_{n-1} \)
  
  • Gives: \( x^p = z_0 \cdot z_1^2 \cdot (z_2^2)^2 \cdot \ldots \cdot (((z_{n-1}^2)^2)^2)^2 \)
    - \( z_i = 1 \) when \( p_i = 0 \)
    - \( z_i = x \) when \( p_i = 1 \)
  
  • Complexity \( O(\log p) \)

Example

\[ 3^{10} = 3^2 \cdot 3^8 = 3^2 \cdot ((3^2)^2)^2 \]
/* Compute x raised to nonnegative power p */
int ipwr_for(int x, unsigned int p)
{
    int result;
    for (result = 1; p != 0; p = p>>1) {
        if (p & 0x1)
            result *= x;
        x = x*x;
    }
    return result;
}

<table>
<thead>
<tr>
<th>before iteration</th>
<th>result</th>
<th>x=3</th>
<th>p=10</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>3</td>
<td>10=1010₂</td>
</tr>
<tr>
<td>2</td>
<td>1</td>
<td>9</td>
<td>5= 101₂</td>
</tr>
<tr>
<td>3</td>
<td>9</td>
<td>81</td>
<td>2= 10₂</td>
</tr>
<tr>
<td>4</td>
<td>9</td>
<td>6561</td>
<td>1= 1₂</td>
</tr>
<tr>
<td>5</td>
<td>59049</td>
<td>43046721</td>
<td>0₂</td>
</tr>
</tbody>
</table>
### “For” Loop Example

```c
int result;
for (result = 1; p != 0; p = p >> 1)
{
    if (p & 0x1)
        result *= x;
    x = x * x;
}
```

#### General Form

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

<table>
<thead>
<tr>
<th>Init</th>
<th>Test</th>
<th>Update</th>
<th>Body</th>
</tr>
</thead>
<tbody>
<tr>
<td>result = 1</td>
<td>p != 0</td>
<td>p = p &gt;&gt; 1</td>
<td>{</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>if (p &amp; 0x1)</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>result *= x;</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>x = x * x;</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>}</td>
</tr>
</tbody>
</table>
"For" → "While"

For Version

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

While Version

```plaintext
Init;
while (Test) {
  Body
  Update;
}
```

Goto Version

```plaintext
Init;
goto middle;
loop:
  Body
  Update;
middle:
  if (Test)
    goto loop;
done:
```
For-Loop: Compilation

For Version

\[
\text{for (Init; Test; Update )}
\]

\[
\text{Body}
\]

Goto Version

\[
\text{Init;}
\]
\[
\text{goto middle;}
\]
\[
\text{loop:}
\]
\[
\text{Body}
\]
\[
\text{Update ;}
\]
\[
\text{middle:}
\]
\[
\text{if (Test)}
\]
\[
\text{goto loop;}
\]
\[
\text{done:}
\]

\[
\text{for (result = 1; p != 0; p = p>>1)}
\]
\[
\{\]
\[
\text{if (p & 0x1)}
\]
\[
\text{result *= x;}
\]
\[
\text{x = x*x;}
\]
\[
\}
\]

\[
\text{result = 1;}
\]
\[
\text{goto middle;}
\]
\[
\text{loop:}
\]
\[
\text{if (p & 0x1)}
\]
\[
\text{result *= x;}
\]
\[
\text{x = x*x;}
\]
\[
\text{p = p >> 1;}
\]
\[
\text{middle:}
\]
\[
\text{if (p != 0)}
\]
\[
\text{goto loop;}
\]
\[
\text{done:}
\]
Quick Review

• Complete memory addressing mode
  • (%eax), 17(%eax), 2(%ebx, %ecx, 8), ...

• Arithmetic operations that do set condition codes
  • subl %eax, %ecx        # ecx = ecx + eax
  • sall $4,%edx           # edx = edx << 4
  • addl 16(%ebp),%ecx     # ecx = ecx + Mem[16+ebp]
  • imull %ecx,%eax        # eax = eax * ecx

• Arithmetic operations that do NOT set condition codes
  • leal 4(%edx,%eax),%eax  # eax = 4 + edx + eax
Quick Review

- **x86-64 vs. IA32**
  - Integer registers: \(16 \times 64\)-bit vs. \(8 \times 32\)-bit
  - \texttt{movq, addq, ...} vs. \texttt{movl, addl, ...}
  - \texttt{movq} -> “move quad word” or \(4 \times 16\)-bits
  - x86-64: better support for passing function arguments in registers

- **Control**
  - Condition code registers
  - Set as side effect or by \texttt{cmp, test}
  - Used:
    - Read out by setx instructions (\texttt{setg, setle, ...})
    - Or by conditional jumps (\texttt{jle .L4, je .L10, ...})
    - Or by conditional moves (\texttt{cmovle %edx, %eax})
Quick Review

- **Do-While loop**

  ![Diagram showing C Code for Do-While loop]

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

- **While-Do loop**

  ![Diagram showing C Code for While-Do loop]

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

  ![Diagram showing Goto Version for While-Do loop]

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

- **Do-While Version**

  ![Diagram showing C Code for Do-While version]

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

  ![Diagram showing Goto Version for Do-While version]

  ```c
  done:
  goto middle;
  ```

  or

  ```c
  if (Test)
  goto loop;
  ```
Summarizing

• **C Control**
  - if-then-else
  - do-while
  - while, for
  - switch

• **Assembler Control**
  - Conditional jump
  - Conditional move
  - Indirect jump
  - Compiler
  - Must generate assembly code to implement more complex control

• **Standard Techniques**
  - Loops converted to do-while form
  - Large switch statements use jump tables
  - Sparse switch statements may use decision trees (see text)

• **Conditions in CISC**
  - CISC machines generally have condition code registers