This week

- **Lab 1 due 11pm today**
  - Frustrating? Awesome? Both? Neither?

- **Lab 2 out later today, due 11pm Wednesday July 17**
  - Disassembly, reverse engineering machine code

- **For those fascinated by floating point:**
  - Lecture 1pm Wednesday July 10 in CSE 305 by William Kahan, main designer of the IEEE floating point standard and Turing Award winner.

- **HW 2 due 11pm Thursday July 11**
Swap Ints in 32-bit Mode

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

```
swap:
    pushl %ebp
    movl %esp,%ebp
    pushl %ebx
    movl 12(%ebp),%ecx
    movl 8(%ebp),%edx
    movl (%ecx),%eax
    movl (%edx),%ebx
    movl %eax,(%edx)
    movl %ebx,(%ecx)
    movl -4(%ebp),%ebx
    movl %ebp,%esp
    popl %esp
    ret
```

Body

Finish

Setup

Offset

-4  Old %ebx
  0   Old %ebp
  4   Rtn adr
  8   xp
 12   yp
32-bit vs. 64-bit operands

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

- New instruction forms:
  - `movl` → `movq`
  - `addl` → `addq`
  - `sall` → `salq`
  - etc.

- x86-64 can still use 32-bit instructions that generate 32-bit results
  - Higher-order bits of destination register are just set to 0
  - Example: `addl`
Swap Ints in 64-bit Mode

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

- **Arguments passed in registers (why useful?)**
  - First (`xp`) in `%rdi`, second (`yp`) in `%rsi`
  - 64-bit pointers

- **No stack operations required: faster**

- **32-bit data**
  - Data held in registers `%eax` and `%edx`
  - `movl` operation (the `l` refers to data width, not address width)
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;
}

swap_l:
    movq (%rdi), %rdx
    movq (%rsi), %rax
    movq %rax, (%rdi)
    movq %rdx, (%rsi)
    retq
```

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

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

**Most General Form:**

\[ D(Rb,Ri,S) \rightarrow \text{Mem}[\text{Reg}[Rb] + S*\text{Reg}[Ri] + D] \]

- **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

1. \((Rb,Ri)\) \rightarrow \text{Mem}[\text{Reg}[Rb]+\text{Reg}[Ri]] \quad (S=1, \text{D}=0)
2. \(D(Rb,Ri)\) \rightarrow \text{Mem}[\text{Reg}[Rb]+\text{Reg}[Ri]+D] \quad (S=1)
3. \((Rb,Ri,S)\) \rightarrow \text{Mem}[\text{Reg}[Rb]+S*\text{Reg}[Ri]] \quad (D=0)

...
**Address Computation Examples**

<table>
<thead>
<tr>
<th>%edx</th>
<th>0xf000</th>
</tr>
</thead>
<tbody>
<tr>
<td>%ecx</td>
<td>0x100</td>
</tr>
</tbody>
</table>

(Rb,Ri)   Mem[Reg[Rb]+Reg[Ri]]
D(Ri,S)   Mem[S*Reg[Ri]+D]
(Rb,Ri,S) Mem[Reg[Rb]+S*Reg[Ri]]
D(Rb)     Mem[Reg[Rb]+D]

<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

- Load Effective Address

1. `leal Src, Dest`  
2. `leaq Src, Dest`
   - `Src` is address mode expression
   - Set `Dest` to address computed by expression
   - 32-bit Example: `leal (%edx,%ecx,4), %eax`
   - 64-bit Example: `leaq (%rdx,%rcx,4), %rax`

- 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, or 8`
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>Dest = Dest + Src</td>
</tr>
<tr>
<td><code>subl</code></td>
<td>Dest = Dest – Src</td>
</tr>
<tr>
<td><code>imull</code></td>
<td>Dest = Dest * Src</td>
</tr>
<tr>
<td><code>shll</code></td>
<td>Dest = Dest &lt;&lt; Src</td>
</tr>
<tr>
<td><code>sarl</code></td>
<td>Dest = Dest &gt;&gt; Src</td>
</tr>
<tr>
<td><code>shrl</code></td>
<td>Dest = Dest &gt;&gt; Src</td>
</tr>
<tr>
<td><code>xorl</code></td>
<td>Dest = Dest ^ Src</td>
</tr>
<tr>
<td><code>andl</code></td>
<td>Dest = Dest &amp; Src</td>
</tr>
<tr>
<td><code>orl</code></td>
<td>Dest = Dest</td>
</tr>
</tbody>
</table>

- **Watch out for argument order! (especially `subl`)**
- **Few distinctions between signed and unsigned int (why?)**
  - except `sarl` vs. `shrl`, see CS:APP 3.5.5 about extra case for `imull`
Some Arithmetic Operations

- **One Operand (Unary) Instructions**
  
<table>
<thead>
<tr>
<th>Instruction</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>incl Dest</code></td>
<td><code>Dest = Dest + 1</code> increment</td>
</tr>
<tr>
<td><code>decl Dest</code></td>
<td><code>Dest = Dest - 1</code> decrement</td>
</tr>
<tr>
<td><code>negl Dest</code></td>
<td><code>Dest = -Dest</code> negate</td>
</tr>
<tr>
<td><code>notl Dest</code></td>
<td><code>Dest = ~Dest</code> bitwise complement</td>
</tr>
</tbody>
</table>

- 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;
}
```

```assembly
arith:
    pushl %ebp
    movl %esp,%ebp

    movl 8(%ebp),%eax
    movl 12(%ebp),%edx
    leal (%edx,%eax),%ecx
    leal (%edx,%edx,2),%edx
    sal $4,%edx
    addl 16(%ebp),%ecx
    leal 4(%edx,%eax),%eax
    imull %ecx,%eax

    movl %ebp,%esp
    popl %ebp
    ret
```
Understanding \textit{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;
}
```

\textbf{Stack}

- Offset 16 \rightarrow z
- Offset 12 \rightarrow y
- Offset 8 \rightarrow x
- Offset 4 \rightarrow Rtn adr
- Offset 0 \rightarrow Old `%ebp`

\textbf{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
- `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)
### 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
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)
```

Stack

<table>
<thead>
<tr>
<th>Offset</th>
<th>Variable</th>
</tr>
</thead>
<tbody>
<tr>
<td>16</td>
<td>z</td>
</tr>
<tr>
<td>12</td>
<td>y</td>
</tr>
<tr>
<td>8</td>
<td>x</td>
</tr>
<tr>
<td>4</td>
<td>Rtn adr</td>
</tr>
<tr>
<td>0</td>
<td>Old %ebp</td>
</tr>
</tbody>
</table>

%ebp
Understanding arith (IA32)

```
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)
```
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)
```
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:

  \[(x+y+z) \times (x+4+48*y)\]
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;
}

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

logical:
pushl %ebp
movl %esp,%ebp

movl 8(%ebp),%eax
xorl 12(%ebp),%eax
sarl $17,%eax
andl $8185,%eax
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;
}
```

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

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
```

Set Up

Body

Finish
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:**

```
pushl %ebp
movl %esp,%ebp

movl 8(%ebp),%eax
xorl 12(%ebp),%eax
sarq $17,%eax
andl $8185,%eax

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

- **Set Up**
  - movl 8(%ebp),%eax
  - xorl 12(%ebp),%eax
  - sarq $17,%eax
  - andl $8185,%eax

- **Body**
  - eax = x
  - eax = x^y (t1)
  - eax = t1>>17 (t2)
  - eax = t2 & 8185

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

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

**Notes**
- $2^{13} = 8192, 2^{13} - 7 = 8185$
- $...0010000000000000, ...0001111111111001$

---

**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`
Topics: control flow

- Condition codes
- Conditional and unconditional branches
- Loops
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)
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 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>
**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)
  - **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 (beware!)

- **Full documentation (IA32):** http://www.jegerlehner.ch/intel/IntelCodeTable.pdf
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**
  - `cmp1/cmpq Src2,Src1`
  - `cmp1 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
    - $(a > 0 \&\& b < 0 \&\& (a-b) < 0) \lor (a < 0 \&\& b > 0 \&\& (a-b) > 0)$
Condition Codes (Explicit Setting: Test)

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

- **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>settle</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>
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

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

**Body:** y at 12(%ebp), x at 8(%ebp)

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

```
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
```

`int gt (int x, int y)`

(x – y)
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>
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    .L7
    subl   %eax, %edx
    movl   %edx, %eax
    .L8:
    leave
    ret
    .L7:
    subl   %edx, %eax
    jmp    .L8
```

- **Setup**
- **Body1**
- **Finish**
- **Body2**
Conditional Branch Example (Cont.)

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

```
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;
}
```

- C allows “goto” as means of transferring control
  - Closer to machine-level programming style
- Generally considered bad coding style
http://xkcd.com/292/
Conditional Branch Example (Cont.)

```c
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;
}
```

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

.L7:
    leave
    ret

.L8:
    subl   %edx, %eax
    jmp    .L8
```

```
int x  %edx
int y  %eax
```
Conditional Branch Example (Cont.)

```c
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;
}
```

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

```c
int x %edx
int y %eax
```
### Conditional Branch Example (Cont.)

```c
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
```

### Pseudo-Assembly:
```
int x %edx
int y %eax
```
### Conditional Branch Example (Cont.)

```c
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;
}
```

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

.L7:
    subl %edx, %eax
    jmp .L8

.L8:
    leave
    ret
```

- `int x` `%edx`
- `int y` `%eax`
Conditional Branch Example (Cont.)

```c
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;
}
```

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

.L7:
    subl %edx, %eax
    jmp .L8
```

```
int x %edx
int y %eax
```
General Conditional Expression Translation

C Code

\[
\text{val} = \text{Test} \ ? \ \text{Then-Expr} \ : \ \text{Else-Expr};
\]

\[
\text{result} = x > y \ ? \ x - y : y - x;
\]

Goto Version

\[
\text{nt} = !\text{Test};
\]

\[
\text{if} (\text{nt}) \ \text{goto Else};
\]

\[
\text{val} = \text{Then-Expr};
\]

\[
\text{Done:}
\]

\[
\ldots
\]

\[
\text{Else:}
\]

\[
\text{val} = \text{Else-Expr};
\]

\[
\text{goto} \ \text{Done};
\]

\[
\begin{align*}
\text{if} & (\text{Test}) \\
\text{val} & = \text{Then-Expr}; \\
\text{else} & \\
\text{val} & = \text{Else-Expr};
\end{align*}
\]

- \text{Test} is expression returning integer
  - = 0 interpreted as false
  - \neq 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;
}
```

- **Conditional move instruction**
  - `cmovC src, dest`
  - Move value from src to dest if condition `C` holds
  - *Why is this good?*
Conditionals: x86-64

```
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
  - More efficient than conditional branching (simple control flow)
  - But overhead: both branches are evaluated
PC Relative Addressing

- PC relative branches are **relocatable**
- Absolute branches are not
Compiling Loops

C/Java code:

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

Machine code:

```c
loopTop:   cmpl  $0, %eax
          je    loopDone
          <loop body code>
          jmp   loopTop
loopDone:
```

- 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?
“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

Translation?
“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
    ret                  # Finish
```

Registers:

- `%edx` : `x`
- `%eax` : `result`
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 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
“While” Loop Example

```c
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: \( x^p = z_0 \cdot z_1^2 \cdot (z_2^2)^2 \cdot \ldots \cdot (\ldots((z_{n-1}^2)^2)^2)^2 \)
  
  \( z_i = 1 \) 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;
}
“For” Loop Example

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

General Form

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

<table>
<thead>
<tr>
<th>Init</th>
<th>Test</th>
<th>Update</th>
</tr>
</thead>
<tbody>
<tr>
<td>result = 1</td>
<td>p != 0</td>
<td>p = p &gt;&gt; 1</td>
</tr>
</tbody>
</table>

```c
{  
    if (p & 0x1)
        result *= x;
    x = x*x;
}
```
“For” → “While”

**For Version**

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

**While Version**

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

**Goto Version**

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

For Version

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

Goto Version

```plaintext
Init;
goto middle;
loop:
    Body
    Update ;
middle:
    if (Test)
        goto loop;
done:
```

```plaintext
for (result = 1; p != 0; p = p>>1) {
    if (p & 0x1)
        result *= x;
    x = x*x;
}
result = 1;
goto middle;
loop:
    if (p & 0x1)
        result *= x;
    x = x*x;
p = p >> 1;
middle:
    if (p != 0)
        goto loop;
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: \textbf{16 x 64-bit} vs. \textbf{8 x 32-bit}
- \texttt{movq}, \texttt{addq}, ... vs. \texttt{movl}, \texttt{addl}, ...
- \texttt{movq} -> “move quad word” or 4*16-bits
- x86-64: better support for passing function arguments in registers

**Control**

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

- **Do-While loop**
- **While-Do loop**

**C Code**

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

**Loop Version**

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

**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:
```

**or**

```c
goto middle;
loop:
    Body
    middle:
    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