Control flow
Condition codes
Conditional and unconditional jumps
Loops
Switch statements

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 related control flow constructs
  - break, continue
- x86 calls branches “jumps” (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>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>ja</td>
<td>SF</td>
<td>Negative</td>
</tr>
<tr>
<td>jna</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>
**Condition Codes (flags)**

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

**Arithmetic instructions set flags implicitly**

**Example:** `addl` or `addq` `Src, Dest` $\leftrightarrow t = a + b$
- **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

`LEA` does not set flags.

*Full documentation (IA32):* [http://www.jegerlehner.ch/intel/intelCodeTable.pdf](http://www.jegerlehner.ch/intel/intelCodeTable.pdf)

---

**cmp:** compare values with subtraction

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

**Compare instructions set flags**
- `cmp` $\leftrightarrow t = a - b$
  - **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

---

**test:** compare values with &

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

**Test instructions set flags**

- `testl` or `testq` Src2, Src1
- `testl` $b,a$ computes $a \& b$, discards result
  - 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, -

---

**Inspecting 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>setsns</td>
<td>~SF</td>
<td>Nonnegative</td>
</tr>
<tr>
<td>setg</td>
<td>~SF^OF &amp; ~ZF</td>
<td>Greater (Signed)</td>
</tr>
<tr>
<td>setge</td>
<td>~SF^OF</td>
<td>Greater or Equal (Signed)</td>
</tr>
<tr>
<td>setl</td>
<td>(SF^OF)</td>
<td>Less (Signed)</td>
</tr>
<tr>
<td>setle</td>
<td>(SF^OF)</td>
<td>Less or Equal (Signed)</td>
</tr>
<tr>
<td>seta</td>
<td>~CF &amp; ZF</td>
<td>Above (unsigned)</td>
</tr>
<tr>
<td>setb</td>
<td>CF</td>
<td>Below (unsigned)</td>
</tr>
</tbody>
</table>
Inspecting Condition Codes

**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

```
movl 12(%ebp),%eax
```

```
push %ebp
movl %esp, %ebp
movl 8(%ebp), %edx
movl 12(%ebp), %eax
cmp %eax, %edx
jle .L7
subl %eax, %edx
movl %edx, %eax
```

```
absdiff:
```

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

```
Jump!

**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>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>SF</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>
```

```
.body: y at 12(%ebp), x at 8(%ebp)
```

```
Conditional Branch Example
```

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

```
Conditional Branch Example
```

```
.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
- Bad style
Conditional Branch Example

```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;
}
```

Typical control flow code in assembly

```assembly
abdiff:
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**

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

**Goto Version**

```c
nt = Test;
if (nt) goto Else;
val = Then-Expr;
else
val = Else-Expr;
```

Test is expression returning integer
10 interpreted as false
0 interpreted as true
Create separate code regions for then and else expressions
Execute appropriate one
PC Relative Addressing

- Jump instruction encodes offset from next instruction to destination PC.
  - (Not the absolute address of the destination.)
  - PC relative branches are relocatable.
  - Absolute branches are not (or they take a lot work to relocate).

```
0x100  cmp  r2, r3  0x1000
0x102  je   0x70   0x1002
0x104  ...  0x1004
...    ...    ...
0x174  add  r3, r4  0x1074
```

Compiling Loops

C/Java code:

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

Machine code:

```
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

“Do-While” Loop Example

C Code

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

Goto Version

```
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

```
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 (%ebp),%edx

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

Register | Variable
--- | ---
%edx | x
%eax | result

Why put the loop condition at the end?
General “Do-While” Translation

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

**Body:**  
- `Statement;`  
- `Statement;`  
- ...  
- `Statement_n;`  

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

---

“While” Loop Translation

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

```
Goto Version
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 and x86-64**  
Test at end, first iteration jumps over body to test.

---

“While” Loop Example

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

---

“For” Loop Example: Square-and-Multiply

```
/* 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:  
\[ p = p_n \cdot 2^n + p_{n-1} \cdot 2^{n-1} + \ldots + p_0 \cdot 2^0 \]

**Example**  
\[ 3^{10} = 3^2 \cdot 3^3 \]

\[ = 3^2 \cdot (3^3)^2 \]

Complexity \( O(\log p) = O(\text{sizeof}(p)) \)
**ipwr Computation**

```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;
}
```

**Before Iteration**

<table>
<thead>
<tr>
<th>Iteration</th>
<th><code>result</code></th>
<th><code>x</code></th>
<th><code>p</code></th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>3</td>
<td>1010</td>
</tr>
<tr>
<td>2</td>
<td>9</td>
<td>9</td>
<td>101</td>
</tr>
<tr>
<td>3</td>
<td>81</td>
<td>9</td>
<td>101</td>
</tr>
<tr>
<td>4</td>
<td>6561</td>
<td>9</td>
<td>1</td>
</tr>
<tr>
<td>5</td>
<td>59049</td>
<td>43046721</td>
<td>0</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;
}
```

**“For” Loop Example (General Form)**

<table>
<thead>
<tr>
<th>Init</th>
<th>Test</th>
<th>Update</th>
<th>Body</th>
</tr>
</thead>
</table>
| result = 1 | p != 0 | p = p >> 1 | if (p & 0x1) {
|               |        |        |     result *= x;
|               |        |        |     x = x*x;    |
```

**“For” → “While”**

**For Version**

For (Initialize; Test; Update) 
Body

**While Version**

while (Test) {
    Body
    Update;
}

**Goto Version**

```
Initialize:
goto middle;
loop:
    Body
    middle:
    if (Test)
        goto loop;
    done:
```

**For-Loop: Compilation**

**For Version**

for (Initialize; Test; Update) 
Body

**Goto Version**

```
Initialize:
goto middle;
loop:
    if (Test)
        goto loop;
    middle:
```

```c
result = 1;
goto middle;
loop:
    if (p & 0x1) {
        result *= x;
    }
    x = x*x;
    p = p >> 1;
    middle:
    if (p != 0)
        goto loop;
    done:
```
Switch Statements

Multiple case labels
Here: 5, 6
Fall through cases
Here: 2
Missing cases
Here: 4

Lots to manage, we need a **jump table**

```c
long switch_eg (unsigned long x, long y, long z) {
    long w = 1;
    switch(x) {
        case 1:  
            w = y*z;
            break;
        case 2:  
            w = y/z;
            /* Fall Through */
        case 3:  
            w += z;
            break;
        case 4:  
            w -= z;
            break;
        default:  
            w = 2;
    }
    return w;
}
```

Jump Table Structure

**Switch Form**

```c
switch(x) {
    case val_0:  // .L56
        Block 0
        w = y*z;  
        break;
    case val_1:  // .L57
        Block 1
        w = y/z;
        /* Fall Through */
    case val_n-1:  // .L60
        Block n-1
        w += z;
        break;
    default:      // .L61
        w = 2;
}
```

**Jump Table**

```
   Target = JTab[x];
   goto Target;
```

**Jump Targets**

- Targ0: Code Block 0
- Targ1: Code Block 1
- Targ2: Code Block 2
- Targ3: Code Block 3
- Targ4: Code Block 4
- Targ5: Code Block 5
- Targ6: Code Block 6
- Targ7: Code Block 7
- Targ8: Code Block 8
- Targ9: Code Block 9
- Targ10: Code Block 10
- Targ11: Code Block 11
- Targ12: Code Block 12

**Jump Table (IA32)**

```assembly
.switch(x) {
    .L62:
    .L61 # x = 0
    w = y*z;
    break;
    .L66 # x = 1
    w = y/z;
    /* Fall Through */
    .L67 # x = 3
    w += z;
    break;
    .L68 # x = 4
    break;
    .L65 # x = 5
    w -= z;
    break;
    case 6:    // .L60
        w = 2;
        break;
    default:    // .L61
        w = 2;
}
```

**Jump Targets**

- Target0: Code Block 0
- Target1: Code Block 1
- Target2: Code Block 2
- Target3: Code Block 3
- Target4: Code Block 4
- Target5: Code Block 5
- Target6: Code Block 6
- Target7: Code Block 7
- Target8: Code Block 8
- Target9: Code Block 9
- Target10: Code Block 10
- Target11: Code Block 11
- Target12: Code Block 12

Declaring data, not instructions.
Switch Statement Example (IA32)

```assembly
long switch_eg(unsigned long x, long y, long z) {  
    long w = 1;
    switch(x) {
        case 0:  // .L62:  // Default case
            w = 2;    
            break;
        . . .
        default: // .L61  
            w = 2;
    }
    return w;
}
```

Assembly Setup Explanation (IA32)

Table each target requires 4 bytes
Base address at .L62

Jump target address modes
- Direct: jmp L61
- Indirect: jmp *.L62(,%edx,4)

Start of jump table: .L62
Must scale by factor of 4 (labels are 32-bits = 4 bytes on IA32)
Fetch target from effective address .L62 + edx*4
target = JTab[x]; goto target; (only for 0 ≤ x ≤ 6)

Code Blocks (Partial)

```assembly
switch(x) {
    . . .
    case 2:  // .L57  
        w = y/z;
        /* Fall Through */
    case 3:  // .L58
        w += z;
        break;
    . . .
    default: // .L61
        w = 2;
} 
return w;
```
The compiler might choose to pull the return statement in to each relevant case rather than jumping out to it.

The compiler might choose to pull the return statement in to each relevant case rather than jumping out to it.

The compiler might choose to pull the return statement in to each relevant case rather than jumping out to it.

The compiler might choose to pull the return statement in to each relevant case rather than jumping out to it.

The compiler might choose to pull the return statement in to each relevant case rather than jumping out to it.
Switch machine code

Jump Table

Doesn't show up in disassembled code

Can inspect using GDB if we know its address.

```
(gdb) x/7xw 0x080488dc
```

Examine 7 hexadecimal format "words" (4 bytes each)

Use command "help x" to get format documentation

```
0x080488dc:
  0x08048630
  0x08048650
  0x0804863a
  0x08048642
  0x08048630
  0x08048649
  0x08048649
```

Matching Disassembled Targets

```
0x080488dc:
  0x08048630
  0x08048650
  0x0804863a
  0x08048642
  0x08048630
  0x08048649
  0x08048649
```

Question

Would you implement this with a jump table?

```
switch(x) {
    case 0: <some code>
    break;
    case 10: <some code>
    break;
    case 52000: <some code>
    break;
    default: <some code>
    break;
}
```
Quick Review

Control
1-bit condition code/flag registers
Set by arithmetic instructions (addl, shll, etc.), cmp, test
Access flags with setg, setle,... instructions

Conditional jumps use flags for decisions (jle .L4, je .L10, ..)
Unconditional jumps always jump: jmp
Direct or indirect jumps

Standard Techniques
Loops converted to do-while form
Large switch statements use jump tables

Do-While loop
C Code
Goto Version

While-Do loop

Do-While Version

While version

Goto Version

while (Test)
  Body
  while (Test);
  Goto loop

Do-While

if (Test)
  goto done;

Body
done:
  goto middle;

loop:
  if (Test)
    goto loop;

  Body
middle:
  if (Test)
    goto loop;

  Body
  Goto Version