x86 Control Flow

(Part A, Part B)
Condition codes, comparisons, and tests

[Un]Conditional jumps and conditional moves
Translating if-else, loops, and switch statements

1. Compare and test: conditions

cmpq b,a computes a - b, sets flags, discards result

Which flags indicate that a < b? (signed? unsigned?)

testq b,a computes a & b, sets flags, discards result

Common pattern:

testq %rax, %rax

What do ZF and SF indicate?

(Aside) Saving conditions as Boolean values

\( \text{setg: set if greater} \)
stores byte:
\( 0x01 \) if \( \sim (SF ^ OF) & \sim ZF \)
\( 0x00 \) otherwise

gt:

cmpq %rsi,%rdi
setg %al
movzbq %al,%rax
retq

Zero-extend from Byte (8 bits) to Quadword (64 bits)

get__ comes in same flavors as j__ (next slide)
2. Jump: choose next instruction

*Jump/branch* to different part of code by setting $\%\text{rip}$.

<table>
<thead>
<tr>
<th>j___</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>$\text{ZF}$</td>
<td>Equal / Zero</td>
</tr>
<tr>
<td>jne</td>
<td>$\neg$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>$\neg$SF</td>
<td>Nonnegative</td>
</tr>
<tr>
<td>jg</td>
<td>$-(SF\cdot OF)$ &amp; $\neg$ZF</td>
<td>Greater (Signed)</td>
</tr>
<tr>
<td>jge</td>
<td>$-(SF\cdot OF)$</td>
<td>Greater or Equal (Signed)</td>
</tr>
<tr>
<td>jl</td>
<td>$(SF\cdot OF)$</td>
<td>Less (Signed)</td>
</tr>
<tr>
<td>jle</td>
<td>$(SF\cdot OF)$</td>
<td>Less or Equal (Signed)</td>
</tr>
<tr>
<td>ja</td>
<td>$\neg$CF &amp; $\neg$ZF</td>
<td>Above (unsigned)</td>
</tr>
<tr>
<td>jb</td>
<td>CF</td>
<td>Below (unsigned)</td>
</tr>
</tbody>
</table>

### Interpreting Conditional Jumps

It is easier to read conditional jumps in x86-64 by comparing b against a instead of looking at condition codes.

<table>
<thead>
<tr>
<th>cmp a,b</th>
<th>test a,b</th>
</tr>
</thead>
<tbody>
<tr>
<td>je</td>
<td>“Equal”</td>
</tr>
<tr>
<td>jne</td>
<td>“Not equal”</td>
</tr>
<tr>
<td>js</td>
<td>“Sign” (negative)</td>
</tr>
<tr>
<td>jns</td>
<td>(non-negative)</td>
</tr>
<tr>
<td>jg</td>
<td>“Greater”</td>
</tr>
<tr>
<td>jge</td>
<td>“Greater or equal”</td>
</tr>
<tr>
<td>jl</td>
<td>“Less”</td>
</tr>
<tr>
<td>jle</td>
<td>“Less or equal”</td>
</tr>
<tr>
<td>ja</td>
<td>“Above” (unsigned &gt;)</td>
</tr>
<tr>
<td>jb</td>
<td>“Below” (unsigned &lt;)</td>
</tr>
</tbody>
</table>

### Jump for control flow

Jump immediately follows comparison/test. Together, they make a decision:

"if $\%\text{rcx} == \%\text{rax}$ then jump to label."

```
cmpq %rax, %rcx
je label
... 
```

```
je label
... 
```

```
label: addq %rdx, %rax
```

**Label**

Name for address of following item.

### Conditional branch example

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

```
cmpq %rax, %rcx
je label
... 
```

```
je label
... 
```

```
label: addq %rdx, %rax
```

**Labels**

Name for address of following item.

**How did the compiler create this?**

```
cmpq %rsi, %rdi
jle .L7
subq %rsi, %rdi
movq %rdi, %rax
```

```
.absdiff:
cmpq %rsi, %rdi
jle .L7
subq %rdi, %rsi
movq %rsi, %rax
jmp .L8
```

```
.L8:
retq
```

```
.L7:
```

```
subq %rdi, %rsi
movq %rsi, %rax
jmp .L8
```
Control-Flow Graph
Code flowchart/directed graph.

Nodes = Basic Blocks:
Straight-line code always executed together in order.

Edges = Control Flow:
Which basic block executes next (under what condition).

Choose a linear order of basic blocks.

Choose a linear order of basic blocks.

Why might the compiler choose this basic block order instead of another valid order?

Translate basic blocks with jumps + labels

Why might the compiler choose this basic block order instead of another valid order?

Introduced by Fran Allen, et al. Won the 2006 Turing Award for her work on compilers.

introduced by fran allen, et al. won the 2006 turing award for her work on compilers.
Execute absdiff

```plaintext
<table>
<thead>
<tr>
<th>Registers</th>
</tr>
</thead>
<tbody>
<tr>
<td>%rax</td>
</tr>
<tr>
<td>%rdi 5</td>
</tr>
<tr>
<td>%rsi 3</td>
</tr>
</tbody>
</table>
```

End:

```plaintext
retq
```

Else:

```plaintext
subq %rdi, %rsi
movq %rsi, %rax
jmp End
```

Note: CSAPP shows translation with goto

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

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

Close to assembly code.

Execute absdiff

```plaintext
<table>
<thead>
<tr>
<th>Registers</th>
</tr>
</thead>
<tbody>
<tr>
<td>%rax</td>
</tr>
<tr>
<td>%rdi 4</td>
</tr>
<tr>
<td>%rsi 7</td>
</tr>
</tbody>
</table>
```

End:

```plaintext
retq
```

Else:

```plaintext
subq %rdi, %rsi
movq %rsi, %rax
jmp End
```

Note: CSAPP shows translation with goto

```c
long goto_ad(long x, long y){
    long result;
    if (x <= y) goto Else;
    result = x - y;
    End:
    return result;
    Else:
    result = y - x;
    goto End;
}
```
But never use goto in your source code!

http://xkcd.com/292/

Compile if-else

```c
long wacky(long x, long y){
    long result;
    if (x + y > 7) {
        result = x;
    } else {
        result = y + 2;
    }
    return result;
}
```

Assume x is available in %rdi, y is available in %rsi.

Place result in %rax for return.

Compile if-else (solution #1)

```c
long wacky(long x, long y){
    long result;
    if (x + y > 7) {
        result = x;
    } else {
        result = y + 2;
    }
    return result;
}
```

Assume x is available in %rdi, y is available in %rsi.

Place result in %rax for return.

Compile if-else (solution #2)

```c
long wacky(long x, long y){
    long result;
    if (x + y > 7) {
        result = x;
    } else {
        result = y + 2;
    }
    return result;
}
```

Assume x is available in %rdi, y is available in %rsi.

Place result in %rax for return.
Encoding jumps: PC-relative addressing

0x100  cmpq  %rax, %rbx  0x1000
0x102  je  0x70  0x1002
0x104  ...  0x1004
...  ...  ...
0x174  addq  %rax, %rbx  0x1074

PC-relative offsets support relocatable code.
Absolute branches do not (or it's hard).

x86 Control Flow

Condition codes, comparisons, and tests
[Un]Conditional jumps and conditional moves
Translating if-else, loops, and switch statements

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

Why put the loop condition at the end?
### while loop

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

This order is used by GCC for x86-64. Why?

### for loop translation

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

#### Initialize
- while (Test) {
  - Body;
  - Update;
}

#### Body
- if (p & 0x1) {
  - result = result * x;
  - x = x * x;
}

#### Update
- p = p >> 1;

#### Test
- (p != 0) ?

### for loop: square-and-multiply

```c
/* Compute x raised to nonnegative power p */
int power(int x, unsigned int p) {
    int result;
    for (result = 1; p != 0; p = p>>1) {
        if (p & 0x1) {
            result = 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: \( x^p = z_0 \cdot z_1^2 \cdot (z_2 2^2) \cdot \ldots \cdot (z_{n-1} 2^{n-1})^2 \)

\( z_i = 1 \) when \( p_i = 0 \)
\( z_i = x \) when \( p_i = 1 \)

\[ x^m \cdot x^n = x^{m+n} \]

Example

\[ 3^{31} = 3^1 \cdot 3^2 \cdot 3^4 = 3^1 \cdot 3^2 \cdot ((3^2)^2) \]

**Complexity** \( O(\log p) = O(\text{sizeof}(p)) \)
**for loop: power iterations**

```c
/* Compute x raised to nonnegative power p */
int power(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>iterations</th>
<th>result</th>
<th>x</th>
<th>p</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>3</td>
<td>11</td>
</tr>
<tr>
<td>1</td>
<td>3</td>
<td>5</td>
<td>10</td>
</tr>
<tr>
<td>2</td>
<td>27</td>
<td>81</td>
<td>2</td>
</tr>
<tr>
<td>3</td>
<td>27</td>
<td>6561</td>
<td>1</td>
</tr>
<tr>
<td>4</td>
<td>177147</td>
<td>43046721</td>
<td>0</td>
</tr>
</tbody>
</table>

**optional**

**switch statement**

```c
long switch_eg (long x, long y, long z) {
    long w = 1;
    switch(x) {
        case 1:
            w = y * z;
            break;
        case 2:
            w = y - z;
        case 3:
            w += z;
            break;
        case 5:
        case 6:
            w -= z;
            break;
        default:
            w = 2;
    }
    return w;
}
```

**Aside**

**Conditional Move**

*Why?* Branch prediction in pipelined/OoO processors.

```c
cmov src, dest
if (Test) Dest ← Src
```

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

**Bad uses of conditional move**

**Expensive Computations**

```c
val = Test(x) ? Hard1(x) : Hard2(x);
```

**Risky Computations**

```c
val = p ? *p : 0;
```

**Computations with side effects**

```c
val = x > 0 ? x++ : x--; 
```
**switch jump table structure**

C code:

```c
switch(x) {
    case 1: <some code> break;
    case 2: <some code> break;
    case 3: <some code> break;
    case 5:
    case 6: <some code> break;
    default: <some code>
}
```

Translation sketch:

```c
if (0 <= x && x <= 6) 
    addr = jumptable[x];
    goto addr;
else 
    goto default;
```

**switch case dispatch**

```c
long switch_eg(long x, long y, long z) {
    long w = 1;
    switch(x) {
        ... 
        return w;
    }
}
```

Jump if above (unsigned, but...)

```
switch_eg:
    movl $1, %eax
    cmpq $6, %rdi
    ja .L8
    jmp *.L4(&%rdi,8)
```

**switch cases**

```c
switch(x) {
    case 1: // .L3
        w = y * z;
        break;
    case 2: // .L5
        w = y - z;
        break;
    case 3: // .L9
        w += z;
        break;
    case 5:
    case 6: // .L7
        w = z;
        break;
    default: // .L8
        w = 2;
    }
return w;
```

**switch jump table assembly declaration**

```asm
.section .rodata
.align 8
.L4:
    .quad .L8 # x == 0
    .quad .L3 # x == 1
    .quad .L5 # x == 2
    .quad .L9 # x == 3
    .quad .L8 # x == 4
    .quad .L7 # x == 5
    .quad .L7 # x == 6
```

```
switch(x) {
    case 1: // .L3
        w = y * z;
        break;
    case 2: // .L5
        w = y - z;
        break;
    case 3: // .L9
        w += z;
        break;
    case 5:
    case 6: // .L7
        w = z;
        break;
    default: // .L8
        w = 2;
    }
```

**Reg. Use**

<table>
<thead>
<tr>
<th>%rdi</th>
<th>x</th>
</tr>
</thead>
<tbody>
<tr>
<td>%rsi</td>
<td>y</td>
</tr>
<tr>
<td>%rdx</td>
<td>z</td>
</tr>
<tr>
<td>%rax</td>
<td>w</td>
</tr>
</tbody>
</table>

Aside: movl is used because 2 is a small positive value that fits in 32 bits. High order bits of %rax get set to zero automatically. It takes fewer bytes to encode a literal movl vs a movq.
Switch machine code

Assembly Code

```assembly
switch_eg:
  .
  cmpq $6, %rdi
  ja .L0
  jmp *.L4(,%rdi,8)
```

Disassembled Object Code

```
00000000004004f6 <switch_eg>:
  . .
4004fd:  77 2b                   ja 40052a <switch_eg+0x34>
4004ff:  ff 24 fd d0 05 40 00   jmpq *0x4005d0(,%rdi,8)
```

Inspect jump table contents using GDB.

Examine contents as 7 addresses

```
Address of code for case 0
Address of code for case 1
Address of code for case 6
```

Would you implement this with a jump table?

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

x86 Control Flow

(Part A, Part B)

Condition codes, comparisons, and tests
[Un]Conditional jumps and conditional moves
Translating if-else, loops, and switch statements

https://cs.wellesley.edu/~cs240/