CS429: Computer Organization and Architecture
Instruction Set Architecture III

Dr. Bill Young
Department of Computer Sciences
University of Texas at Austin

Last updated: February 23, 2018 at 14:49
We can now generate programs that execute linear sequences of instructions: access registers and memory, perform computations.

But what about loops, conditions, etc.? Need ISA support for:

- comparing and testing data values
- directing program control
  - jump to some instruction that isn’t just the next one in sequence
- Do so based on some condition that has been tested.

```c
#include <stdio.h>

int main(void)
{
    int count;
    for (count = 1; count <= 500; count++)
        printf("I will not throw paper airplanes in class.");
    return 0;
}```
Processor State (x86-64, Partial)

Information about currently executing program.

- Temporary data (%rax, ...)
- Location of runtime stack (%rsp)
- Location of current code control point (%rip)
- Status of recent tests (CF, ZF, SF, OF)

<table>
<thead>
<tr>
<th>Registers</th>
</tr>
</thead>
<tbody>
<tr>
<td>%rax</td>
</tr>
<tr>
<td>%rbx</td>
</tr>
<tr>
<td>%rcx</td>
</tr>
<tr>
<td>%rdx</td>
</tr>
<tr>
<td>%rsi</td>
</tr>
<tr>
<td>%rdi</td>
</tr>
<tr>
<td>%rsp</td>
</tr>
<tr>
<td>%rbp</td>
</tr>
</tbody>
</table>

%rip Instruction pointer

CF | ZF | SF | OF   | Condition codes
PC-relative Addressing

Don’t use `%rip` as a general purpose register.

However, the compiler may generate *PC-relative addressing*.

\[
\text{jmp } 0\times10(\%\text{rip})
\]

The effective address for a PC-relative instruction address is the offset parameter added to the address of the next instruction. This offset is signed to allow reference to code both before and after the instruction.

Can you guess why the compiler might generate such code?
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 by arithmetic operations*

E.g., addq Src, Dest

C analog: \( t = a + b \);

- 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 overflow:
  \( (a>0 && b>0 && t<0) || (a<0 && b<0 && t >=0) \)

Condition codes not set by lea instruction.
Explicitly set by Compare instruction
\texttt{cmpq Src2, Src1}

- \texttt{cmpq b, a} is like computing \((a - b)\) without setting destination.
- CF set if carry out from most significant bit; used for unsigned computations.
- ZF set if \(a == b\)
- SF set if \((a-b) < 0\)
- OF set if two’s complement (signed) overflow:
  \[(a>0 \&\& b>0 \&\& (a-b)<0) \mid\mid (a<0 \&\& b<0 \&\& (a-b)>=0)\]
Explicitly set by Test instruction

testq Src2, Src1

- Sets condition codes based on value of (Src1 & Src2).
- Often useful to have one of the operands be a mask.
- testq b, a is like computing a&b, without setting a destination.
- ZF set iff (a & b) == 0
- SF set iff (a & b) < 0
- CF and OF are set to 0.

How could you use testq to jump if the value in %rbx is even?
Explicitly set by Test instruction
testq Src2, Src1

- Sets condition codes based on value of (Src1 & Src2).
- Often useful to have one of the operands be a mask.
- testq b, a is like computing a&b, without setting a destination.
- ZF set iff (a & b) == 0
- SF set iff (a & b) < 0
- CF and OF are set to 0.

How could you use testq to jump if the value in %rbx is even?

```
testq $1, %rbx
je    even
odd:
```
**SetX Instructions**: Set low order bytes of destination to 0 or 1, based on combinations of condition codes.

Does not alter remaining 7 bytes.

<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>
### x86-64 Registers: Least Significant Byte

<table>
<thead>
<tr>
<th>%rax</th>
<th>%al</th>
<th>%r8</th>
<th>%r8b</th>
</tr>
</thead>
<tbody>
<tr>
<td>%rbx</td>
<td>%bl</td>
<td>%r9</td>
<td>%r9b</td>
</tr>
<tr>
<td>%rcx</td>
<td>%cl</td>
<td>%r10</td>
<td>%r10b</td>
</tr>
<tr>
<td>%rdx</td>
<td>%dl</td>
<td>%r11</td>
<td>%r11b</td>
</tr>
<tr>
<td>%rsi</td>
<td>%sil</td>
<td>%r12</td>
<td>%r12b</td>
</tr>
<tr>
<td>%rdi</td>
<td>%dil</td>
<td>%r13</td>
<td>%r13b</td>
</tr>
<tr>
<td>%rsp</td>
<td>%spl</td>
<td>%r14</td>
<td>%r14b</td>
</tr>
<tr>
<td>%rbp</td>
<td>%bpl</td>
<td>%r15</td>
<td>%r15b</td>
</tr>
</tbody>
</table>

Can reference the low-order byte.
SetX instructions

- Set single byte based on combinations of conditions codes.
- Argument is one of addressable byte registers.
- Does not alter remaining bytes;
- Typically use movzbl to finish the job (will also zero 4 high order bytes).

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

<table>
<thead>
<tr>
<th>Register</th>
<th>Use(s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>%rdi</td>
<td>Argument x</td>
</tr>
<tr>
<td>%rsi</td>
<td>Argument y</td>
</tr>
<tr>
<td>%rax</td>
<td>return value</td>
</tr>
</tbody>
</table>

```
cmpq %rsi, %rdi # compare x : y
setg %al     # Set if >
movzbl %al, %eax # Zero rest of %rax
retq         
```
**jX Instructions:** Jump to different parts of the code depending on condition codes.

<table>
<thead>
<tr>
<th>jX</th>
<th>Condition</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>jmp</td>
<td>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 (Old Style)**

Generation: `gcc -Og -fno-if-conversion control.c`

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

```assembly
absdiff:
    cmpq %rsi, %rdi        # x:y
    jle .L4
    movq %rdi, %rax
    subq %rsi, %rax
    retq
.L4:
    movq %rsi, %rax
    subq %rdi, %rax
    retq
```

<table>
<thead>
<tr>
<th>Register</th>
<th>Use(s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>%rdi</td>
<td>Argument x</td>
</tr>
<tr>
<td>%rsi</td>
<td>Argument y</td>
</tr>
<tr>
<td>%rax</td>
<td>return value</td>
</tr>
</tbody>
</table>
A common compilation strategy is to take a C construct and rewrite it into an equivalent C version that is closer to assembly, as an intermediate step toward assembly.
C allows “goto” as a means of transferring control.
Jump to position designated by label.
Generally considered bad coding style in high level language.

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

long absdiff_j (long x, long y)
{
    long result;
    int ntest = x <= y;
    if (ntest) goto Else;
    result = x - y;
    goto Done;
Else:
    result = y - x;
Done:
    return result;
}
```
General Conditional Expression Translation

C Code:

```c
val = Test ? Then.Expr : Else.Expr;
```

Example:

```c
val = x>y ? x−y : y−x;
```

Goto Version:

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

- Create separate code regions for then and else expressions.
- Execute the appropriate one.
Conditional Move Instructions

- Refer to generically as “cmovXX”
- Based on values of condition codes
- Conditionally copy value from source to destination.
- Can be used to eliminate conditional jump.
### Conditional Move Instructions

<table>
<thead>
<tr>
<th>Inst.</th>
<th>Synonym</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>cmove</td>
<td>cmovz</td>
<td>Equal / zero</td>
</tr>
<tr>
<td>cmovne</td>
<td>cmovnz</td>
<td>Not equal / not zero</td>
</tr>
<tr>
<td>cmovs</td>
<td></td>
<td>Negative</td>
</tr>
<tr>
<td>cmovns</td>
<td></td>
<td>Not negative</td>
</tr>
<tr>
<td>cmovg</td>
<td>cmovnle</td>
<td>Greater (signed)</td>
</tr>
<tr>
<td>cmovge</td>
<td>cmovnl</td>
<td>Greater or equal (signed)</td>
</tr>
<tr>
<td>cmovl</td>
<td>cmovnge</td>
<td>Less (signed)</td>
</tr>
<tr>
<td>cmove</td>
<td>cmovng</td>
<td>Less or equal (signed)</td>
</tr>
<tr>
<td>cmova</td>
<td>cmovnbe</td>
<td>Above (unsigned)</td>
</tr>
<tr>
<td>cmovae</td>
<td>cmovnb</td>
<td>Above or equal (unsigned)</td>
</tr>
<tr>
<td>cmovb</td>
<td>cmovnae</td>
<td>Below (unsigned)</td>
</tr>
<tr>
<td>cmovbe</td>
<td>cmovna</td>
<td>Below or equal (unsigned)</td>
</tr>
</tbody>
</table>
Using Conditional Moves

Conditional Move Instructions

- Instruction supports:
  - if (Test) Dest ← Src

- Supported in post-1995 x86 processors

- GCC tries to use them, but only when safe

Why?

- Branches are very disruptive to instruction flow through pipelines.

- Conditional moves do not require control transfer.

C Code

```c
val = Test
? Then_Expr
: Else_Expr
```

Goto Version

```c
result = Then_Expr;
eval = Else_Expr;
nt = !Test;
if (nt) result = eval;
return result;
```
Conditional Move Example

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

<table>
<thead>
<tr>
<th>Register</th>
<th>Use(s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>%rdi</td>
<td>Argument x</td>
</tr>
<tr>
<td>%rsi</td>
<td>Argument y</td>
</tr>
<tr>
<td>%rax</td>
<td>return value</td>
</tr>
</tbody>
</table>

absdiff:

```
movq  %rdi, %rax  # x
subq  %rsi, %rax  # result = x - y
movq  %rsi, %rdx
subq  %rdi, %rdx  # eval = y - x
cmpq  %rsi, %rdi  # x:y
cmovle %rdx, %rax  # if <=, result = eval
retq
```
Bad Cases for Conditional Move

Expensive Computations:

\[
\text{val} = \text{Test}(x) \ ? \ \text{Hard1}(x) : \ \text{Hard2}(x);
\]

- Both values get computed
- Only makes sense when computations are very simple

Risky Computations:

\[
\text{val} = p \ ? \ p : 0;
\]

- Both values get computed
- May have undesirable side effects.

Computations with Side Effects:

\[
\text{val} = x > 0 \ ? \ x *= 7 : x += 3;
\]

- Both values get computed
- Must be side effect free
Do-While Loop Example

Following our strategy of rewriting a C construct into a semantically equivalent C version that is closer to assembly.

C Code:

```c
long pcoun_t_do( unsigned long x ) {
    long result = 0;
    do {
        result += x & 0x1;
        x >>= 1;
    } while( x )
    return result ;
}
```

Goto Version:

```c
long pcoun_t_goto( unsigned long x ) {
    long result = 0;
    loop:
        result += x & 0x1;
        x >>= 1;
        if ( x ) goto loop;
    return result ;
}
```

- Count number of 1’s in argument x (“popcount”)
- Use conditional branch to either continue looping or to exit loop
Do-While Loop Compilation

Goto Version:

```c
long pcound_goto
    (unsigned long x) {
loop:
    result += x & 0x1;
    x >>= 1;
    if (x) goto loop;
return result;
}
```

<table>
<thead>
<tr>
<th>Register</th>
<th>Use(s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>%rdi</td>
<td>Argument x</td>
</tr>
<tr>
<td>%rax</td>
<td>return value</td>
</tr>
</tbody>
</table>

**Assembly Code**

```
movl $0 , %eax        # result = 0
.L2: 
movq %rdi , %rdx
andl $1 , %edx        # t = x & 0x1
addq %rdx , %rax      # result += t
shrq $1 , %rdi        # x >>= 1
jne .L2               # if (x) goto loop
retq
```

CS429 Slideset 8: 23 Instruction Set Architecture III
**General Do-While Translation**

**C Code:**

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

**Goto Version:**

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

- **Body** can be any C statement, typically is a compound statement.
- **Test** is an expression returning an integer.
  - If it evaluates to 0, that’s interpreted as false.
  - If it evaluates to anything but 0, that’s interpreted as true.
“Jump-to-middle” translation
- Used with -Og

**While version**

```
while (Test)
    Body
```

**Goto version**

```
goto test;
loop:
    Body
test:
    if (Test)
        goto loop;
done:
```
C Code

```c
long pcount_while (unsigned long s) {
    long result = 0;
    while (x) {
        result += x & 0x1;
        x >>= 1;
    }
    return result;
}
```

Jump to Middle

```c
long pcount_goto_jtm (unsigned long x) {
    long result = 0;
    goto test;
    loop:
        result += x & 0x1;
        x >>= 1;
    test:
        if (x) goto loop;
    return result;
}
```

- Compare to do-while version of function
- Initial goto starts loop at test
C Code

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

which is equivalent to:

Do-While Version

```c
if (!Test)
    goto done;

do
    Body
    while (Test);

done:
```

which gets compiled as if it were:

Goto Version

```c
if (!Test)
    goto done;

loop:
    Body
    if (Test)
        goto loop;

done:
```

Are all three versions semantically equivalent?
While Loop Example #2

C Code

```c
long pcount_while(
    unsigned long x)
{
    long result = 0;
    while (x) {
        result += x & 0x1;
        x >>= 1;
    }
    return result;
}
```

Do-While version

```c
long pcount_goto_dw(
    unsigned long x)
{
    long result = 0;
    if (!x) goto done;
    loop:
        result += x & 0x1;
        x >>= 1;
        if (x) goto loop;
    done:
        return result;
}
```

- Compare to do-while version of function
- Initial conditional guards entrance to loop
For Loop Form

General Form

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

Init

\[ i = 0 \]

Test

\[ i < \text{WSIZE} \]

Update

\[ i++ \]

Body

\#define WSIZE 8*sizeof(long)
long pcount_for
    (unsigned long x)
{
    size_t i;
    long result = 0;
    for (i=0; i<WSIZE; i++)
    {
        unsigned bit =
            (x >> i) & 0x1;
        result += bit;
    }
    return result;
}
For version

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

translates to:

While version

\[
\text{Init} \ ; \\
\text{while} \ (\text{Test}) \ \\ \\
\{ \\
\text{Body} \ ; \\
\text{Update} \ ; \\
\}
\]
For-While Conversion Example

Init

\[ i = 0 \]

Test

\[ i < \text{WSIZE} \]

Update

\[ i++ \]

Body

\[
\begin{align*}
\text{long} & \quad \text{pcount\_for\_while} \\
& \quad (\text{unsigned long} \; x) \\
& \quad \\
& \quad \{ \\
& \quad \quad \text{size\_t} \; i; \\
& \quad \quad \text{long} \; \text{result} = 0; \\
& \quad \quad i = 0; \\
& \quad \quad \textbf{while} \; (i < \text{WSIZE}) \\
& \quad \quad \{ \\
& \quad \quad \quad \text{unsigned} \; \text{bit} = \\
& \quad \quad \quad \quad (x \gg i) \& 0x1; \\
& \quad \quad \quad \text{result} += \text{bit}; \\
& \quad \quad \quad i++; \\
& \quad \quad \} \\
& \quad \} \\
& \quad \text{return} \; \text{result}; \\
\end{align*}
\]
C Code: Goto version

```c
long pcount_for_goto_dw
    (unsigned long x)
{
    size_t i;
    long result = 0;
    for (i=0; i<WSIZE; i++)
    {
        unsigned bit =
            (x >> i) & 0x1;
        result += bit;
    }
    return result;
}

Note that the initial test is not needed. Why?
```
long switch_eq (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 5:
    case 6:
        w -= z;
        break;
    default:
        w = 2;
    }
    return w;
}
Jump Table Structure

**Switch Form**

```
switch(x) {
    case val_0:  
        Block 0 
    case val_1:  
        Block 1 
        ...
    case val_n - 1:  
        Block n-1 
}
```

**Translation (Extended C)**

```
goto *JTab[x];
```

**Jump Table**

<table>
<thead>
<tr>
<th>JTab:</th>
<th>Targ0</th>
<th>Targ1</th>
<th>Targ2</th>
<th>...</th>
<th>Targn-1</th>
</tr>
</thead>
</table>

**Jump Targets**

<table>
<thead>
<tr>
<th>Targ0:</th>
<th>Code Block 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>Targ1:</td>
<td>Code Block 1</td>
</tr>
<tr>
<td>Targ2:</td>
<td>Code Block 2</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>Targn-1:</td>
<td>Code Block n-1</td>
</tr>
</tbody>
</table>
long switch_eq(long x, long y, long z)
{
    long w = 1;
    switch(x) {
        ...
    }
    return w;
}

Setup:

<table>
<thead>
<tr>
<th>Register</th>
<th>Use(s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>%rdi</td>
<td>Argument x</td>
</tr>
<tr>
<td>%rsi</td>
<td>Argument y</td>
</tr>
<tr>
<td>%rdx</td>
<td>Argument z</td>
</tr>
<tr>
<td>%rax</td>
<td>return value</td>
</tr>
</tbody>
</table>

Note that w is not initialized here.
Switch Statement Example

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

Jump table

```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

Setup:

```asm
switch_eq:
    movq %rdx, %rcx
    cmpq $6, %rdi
    ja .L8
    jmp *.L4(, %rdi, 8)
```

# x:6
# use default
# goto *JTAB[x],
# indirect jump
**Assembly Setup Explanation**

**Table Structure**
- Each target requires 8 bytes
- Base address at .L4

**Jumping**
- **Direct:** `jmp .L8`
- Jump target is denoted by label `.L8`
- **Indirect:**
  - `jmp *.*L4(, %rdi, 8)`
- Start of jump table: `.L4`
- Must scale by factor of 8 (addresses are 8 bytes)
- Fetch target from effective address `.L4 + x*8`, but only for $0 \leq x \leq 6$

```
[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
```
### Jump Table

#### C Code:

```c
long switch_eq (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 5:
        case 6:
            w -= z;
            break;
        default:
            w = 2;
    }
    return w;
}
```

#### Assembly:

```
.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
```

---

**Jump Table:**

```
.long switch_eq
    (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 5:
        case 6:
            w -= z;
            break;
        default:
            w = 2;
    }
    return w;
}
```

---

**Jump Table:**

```
.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;
   ...
}

.L3:
   movq %rsi, %rax  # y
   imulq %rdx, %rax  # y*z
   retq

<table>
<thead>
<tr>
<th>Register</th>
<th>Use(s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>%rdi</td>
<td>Argument x</td>
</tr>
<tr>
<td>%rsi</td>
<td>Argument y</td>
</tr>
<tr>
<td>%rdx</td>
<td>Argument z</td>
</tr>
<tr>
<td>%rax</td>
<td>return value</td>
</tr>
</tbody>
</table>
Handling Fall-Through

```c
long w = 1;
...
switch (x) {
    ...
    case 2:
        w = y/z;
        / F all Through */
    case 3:
        w += z;
        break;
    ...
}
```

```c
    case 2:
        w = y/z;
        goto merge;
    ...
    case 3:
        w = 1;
    merge:
        w += z;
```
long  w = 1;
...
switch (x) {
  ...
  case 2:
    w = y/z;
    /* Fall Through */
  case 3:
    w += z;
    break;
  ...
}
switch (x) {
    ...
    case 5:  // .L7
    case 6:  // .L7
        w -= z;
        break;
    default: // .L8
        w = 2;
}
Suppose you have a set of switch labels that are “sparse” (widely separated).

In this case, it doesn’t make sense to use a jump table.

- If there are only a few labels, simply use a nested if structure.
- If there are many, build a balanced binary search tree.

The compiler decides the appropriate thresholds for what’s “sparse,” what are “a few,” etc.

```java
switch(x) {
    case 0:
        Block 0
    case 620:
        Block 620
    ...
    case 1040:
        Block 1040
}
```
C Control
- if-then-else
- do-while
- while, for
- switch

Assembler Control
- Conditional jump
- Conditional move
- Indirect jump (via jump tables)
- Compiler generates code sequence to implement more complex control

Standard Techniques
- Loops converted to do-while or jump-to-middle form
- Large switch statements use jump tables
- Sparse switch statements may use decision trees