

# Machine-Level Programming II: Control

COMP400727: Introduction to Computer Systems

Danfeng Shan  
Xi'an Jiaotong University

# Today

**Review of a few tricky bits from last time**

Basics of control flow

Condition codes

Conditional operations

Loops

If we have time: switch statements

# Reminder: Address Modes

## Most General Form

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

D: Constant “displacement” 1, 2, or 4 bytes

Rb: Base register: Any of 16 integer registers

Ri: Index register: Any, except for `%rsp`

S: Scale: 1, 2, 4, or 8 (*why these numbers?*)

## Special Cases

$$(Rb, Ri) \quad \text{Mem}[\text{Reg}[Rb] + \text{Reg}[Ri]]$$

$$D(Rb, Ri) \quad \text{Mem}[\text{Reg}[Rb] + \text{Reg}[Ri] + D]$$

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

# Memory operands and LEA

In most instructions, a memory operand accesses memory

Assembly	C equivalent
<code>mov 6(%rbx,%rdi,8), %ax</code>	<code>ax = *(rbx + rdi*8 + 6)</code>
<code>add 6(%rbx,%rdi,8), %ax</code>	<code>ax += *(rbx + rdi*8 + 6)</code>
<code>xor %ax, 6(%rbx,%rdi,8)</code>	<code>*(rbx + rdi*8 + 6) ^= ax</code>

LEA is special: it *doesn't* access memory

Assembly	C equivalent
<code>lea 6(%rbx,%rdi,8), %rax</code>	<code>rax = rbx + rdi*8 + 6</code>

# Why use LEA?

## ■ CPU designers' intended use: calculate a pointer to an object

- An array element, perhaps
- For instance, to pass just one array element to another function

### Assembly

```
lea (%rbx,%rdi,8), %rax
```

### C equivalent

```
rax = &rbx[rdi]
```

## ■ Compiler authors like to use it for ordinary arithmetic

- It can do complex calculations in one instruction
- It's one of the only three-operand instructions the x86 has
- It doesn't touch the condition codes (we'll come back to this)

### Assembly

```
lea (%rbx,%rbx,2), %rax
```

### C equivalent

```
rax = rbx * 3
```

# Today

Review of a few tricky bits from yesterday

**Basics of control flow**

**Condition codes**

Conditional operations

Loops

If we have time: switch statements

# 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` )

Current stack top

## Registers

<code>%rax</code>	<code>%r8</code>
<code>%rbx</code>	<code>%r9</code>
<code>%rcx</code>	<code>%r10</code>
<code>%rdx</code>	<code>%r11</code>
<code>%rsi</code>	<code>%r12</code>
<code>%rdi</code>	<code>%r13</code>
<code>%rsp</code>	<code>%r14</code>
<code>%rbp</code>	<code>%r15</code>

`%rip` Instruction pointer

<code>CF</code>	<code>ZF</code>	<code>SF</code>	<code>OF</code>
-----------------	-----------------	-----------------	-----------------

Condition codes

# 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 (as side effect) of arithmetic operations

Example: `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

`(a>0 && b>0 && t<0) || (a<0 && b<0 && t>=0)`

**Not set by `leaq` instruction**



# ZF set when

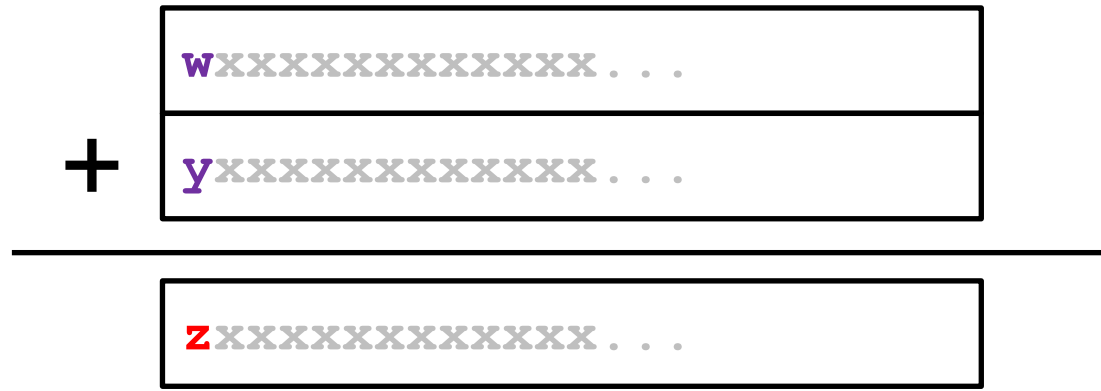
000000000000...000000000000



# CF set when

$$\begin{array}{r} \boxed{y \text{xxxxxxxxxxxxxxxx} \dots} \\ + \boxed{y \text{xxxxxxxxxxxxxxxx} \dots} \\ \hline \mathbf{1} \boxed{z \text{xxxxxxxxxxxxxxxx} \dots} \end{array}$$

# OF set when



`w == y && w != z`

# Compare Instruction

**cmp a, b**

Computes  $b - a$  (just like **sub**)

Sets condition codes based on result, but...

**Does not change  $b$**

**CF set** if carry out from most significant bit (used for unsigned comparisons) (when  $b < a$ )

**ZF set** if  $b == a$

**SF set** if  $(b - a) < 0$  (as signed)

**OF set** if two's-complement (signed overflow)

$(b > 0 \ \&\& \ a < 0 \ \&\& \ (b - a) < 0) \ || \ (b < 0 \ \&\& \ a > 0 \ \&\& \ (b - a) > 0)$

Used for **if (a < b) { ... }**

whenever  $a - b$  isn't needed for anything else

# Test Instruction

**test a, b**

Computes  $b \& a$  (just like **and**)

Sets condition codes (only SF and ZF) based on result, but...

**Does not change  $b$**

**ZF Set** when  $a \& b == 0$

**SF Set** when  $a \& b < 0$

Most common use: `test %rX, %rX`  
to compare `%rX` to zero

# Today

Review of a few tricky bits from yesterday

Basics of control flow

Condition codes

**Conditional operations**

Loops

If we have time: switch statements

# Reading Condition Codes

## SetX a

Set **low-order byte** of **a** to 0 or 1 based on *combinations* of condition codes

Does not alter remaining 7 bytes

SetX	Condition	Description
<b>sete</b>	<b>ZF</b>	<b>Equal / Zero</b>
<b>setne</b>	<b>~ZF</b>	<b>Not Equal / Not Zero</b>
<b>sets</b>	<b>SF</b>	<b>Negative</b>
<b>setns</b>	<b>~SF</b>	<b>Nonnegative</b>
<b>setg</b>	<b>~ (SF^OF) &amp; ~ZF</b>	<b>Greater (Signed)</b>
<b>setge</b>	<b>~ (SF^OF)</b>	<b>Greater or Equal (Signed)</b>
<b>setl</b>	<b>(SF^OF)</b>	<b>Less (Signed)</b>
<b>setle</b>	<b>(SF^OF)   ZF</b>	<b>Less or Equal (Signed)</b>
<b>seta</b>	<b>~CF &amp; ~ZF</b>	<b>Above (unsigned)</b>
<b>setb</b>	<b>CF</b>	<b>Below (unsigned)</b>



# x86-64 Integer Registers

<code>%rax</code>	<code>%al</code>
<code>%rbx</code>	<code>%bl</code>
<code>%rcx</code>	<code>%cl</code>
<code>%rdx</code>	<code>%dl</code>
<code>%rsi</code>	<code>%sil</code>
<code>%rdi</code>	<code>%dil</code>
<code>%rsp</code>	<code>%spl</code>
<code>%rbp</code>	<code>%bpl</code>

<code>%r8</code>	<code>%r8b</code>
<code>%r9</code>	<code>%r9b</code>
<code>%r10</code>	<code>%r10b</code>
<code>%r11</code>	<code>%r11b</code>
<code>%r12</code>	<code>%r12b</code>
<code>%r13</code>	<code>%r13b</code>
<code>%r14</code>	<code>%r14b</code>
<code>%r15</code>	<code>%r15b</code>

SetX argument is always a low byte (`%al`, `%r8b`, etc.)

# Reading Condition Codes (Cont.)

## SetX Instructions:

Set single byte based on combination of condition codes

## One of addressable byte registers

Does not alter remaining bytes

Typically use `movzbl` to finish job

32-bit instructions also set upper 32 bits to 0

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

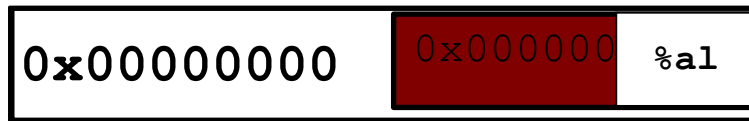
```
cmpq    %rsi, %rdi    # Compare x:y
setg    %al           # Set when >
movzbl  %al, %eax     # Zero rest of %rax
ret
```

Register	Use(s)
<code>%rdi</code>	Argument <b>x</b>
<code>%rsi</code>	Argument <b>y</b>
<code>%rax</code>	Return value

# Reading Condition Codes (Cont.)

Beware weirdness `movzbl` (and others)

```
movzbl %al, %eax
```



Zapped to all  
0's

**Use(s)**

Argument **x**

Argument **y**

Return value

```
cmpq    %rsi, %rdi    # Compare x:y
setg    %al           # Set when >
movzbl  %al, %eax     # Zero rest of %rax
ret
```

# Jumping

## jX Instructions

Jump to different part of code depending on condition codes

jX	Condition	Description
jmp	1	Unconditional
je	ZF	Equal / Zero
jne	$\sim ZF$	Not Equal / Not Zero
js	SF	Negative
jns	$\sim SF$	Nonnegative
jg	$\sim (SF \wedge OF) \ \& \ \sim ZF$	Greater (Signed)
jge	$\sim (SF \wedge OF)$	Greater or Equal (Signed)
jl	$(SF \wedge OF)$	Less (Signed)
jle	$(SF \wedge OF) \   \ ZF$	Less or Equal (Signed)
ja	$\sim CF \ \& \ \sim ZF$	Above (unsigned)
jb	CF	Below (unsigned)

# Conditional Branch Example (Old Style)

Generation

ICSServer> gcc -Og -S -fno-if-conversion

I'll get to this shortly.

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

```
absdiff:
    cmpq    %rsi, %rdi    # x:y
    jle    .L4
    movq    %rdi, %rax
    subq    %rsi, %rax
    ret
.L4:      # x <= y
    movq    %rsi, %rax
    subq    %rdi, %rax
    ret
```

Register	Use(s)
%rdi	Argument <b>x</b>
%rsi	Argument <b>y</b>
%rax	Return value

# Expressing with Goto Code

C allows goto statement

Jump to position designated by label

```
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 (Using Branches)

## C Code

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

```
val = x > y ? x - y : y - x;
```

## Goto Version

```
    ntest = !Test;  
    if (ntest) goto  
Else;  
    val = Then_Expr;  
    goto Done;  
Else:  
    val = Else_Expr;  
Done:  
    . . .
```

Create separate code regions for  
then & else expressions

Execute appropriate one

# Using Conditional Moves

## Conditional Move Instructions

Instruction supports:

if (Test) Dest  $\leftarrow$  Src

Supported in post-1995 x86 processors

GCC tries to use them

But, only when known to be safe

## Why?

Branches are very disruptive to instruction flow through pipelines

Conditional moves do not require control transfer

## C Code

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

## Goto Version

```
result = Then_Expr;  
eval = Else_Expr;  
nt = !Test;  
if (nt) result = eval;  
return result;
```



# Conditional Move Example

```

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

```

Register	Use(s)
%rdi	Argument <b>x</b>
%rsi	Argument <b>y</b>
%rax	Return value

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
ret

```

# Bad Cases for Conditional Move

## Expensive Computations

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

Both values get computed

Only makes sense when computations  
are very simple

Bad Performance

## Risky Computations

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

Both values get computed

May have undesirable effects

Unsafe

## Computations with side effects

```
val = x > 0 ? x*=7 : x+=3;
```

Both values get computed

Must be side-effect free

Illegal

# Today

Review of a few tricky bits from yesterday

Basics of control flow

Condition codes

Conditional operations

**Loops**

If we have time: switch statements

# “Do-While” Loop Example

## C Code

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

## Goto Version

```
long pcount_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

```

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

```

Register	Use(s)
%rdi	Argument <b>x</b>
%rax	<b>result</b>

```

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

```

# General “Do-While” Translation

## C Code

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

```
Body: {  
    Statement1;  
    Statement2;  
    ...  
    Statementn;  
}
```

## Goto Version

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

# General “While” Translation #1

“Jump-to-middle” translation

Used with -Og

**While version**

```
while (Test)  
  Body
```



**Goto Version**

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

# While Loop Example #1

## C Code

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

## Jump to Middle

```
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

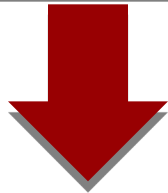
Initial goto starts loop at test



# General “While” Translation #2

## While version

```
while (Test)  
  Body
```



## Do-While Version

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



## “Do-while” conversion

Used with -O1

## Goto Version

```
if (!Test)  
  goto done;  
loop:  
  Body  
  if (Test)  
    goto loop;  
done:
```

# While Loop Example #2

## C Code

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

## Do-While Version

```
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

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

```
#define WSIZE (8*sizeof(int))  
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;  
}
```

## Init

```
i = 0
```

## Test

```
i < WSIZE
```

## Update

```
i++
```

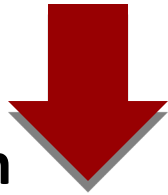
## Body

```
{  
    unsigned bit =  
        (x >> i) & 0x1;  
    result += bit;  
}
```

# “For” Loop → While Loop

## For Version

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



## While Version

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

# For-While Conversion

## Init

```
i = 0
```

## Test

```
i < WSIZE
```

## Update

```
i++
```

## Body

```
{  
    unsigned bit =  
        (x >> i) & 0x1;  
    result += bit;  
}
```

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

# “For” Loop Do-While Conversion

## Goto Version

### C Code

```
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;
}
```

Initial test can be optimized away

```
long pcount_for_goto_dw
(unsigned long x) {
    size_t i;
    long result = 0;
    i = 0;
if (!(i < WSIZE))
    goto done;
    loop:
    {
        unsigned bit =
            (x >> i) & 0x1;
        result += bit;
    }
    i++;
    if (i < WSIZE)
        goto loop;
done:
    return result;
}
```

*Ini*

*+ !Test*

*Body*

*Update*

*Test*

# Today

Review of a few tricky bits from yesterday

Basics of control flow

Condition codes

Conditional operations

Loops

**If we have time: switch statements**

```
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;
        /* Fall Through */
    case 3:
        w += z;
        break;
    case 5:
    case 6:
        w -= z;
        break;
    default:
        w = 2;
    }
    return w;
}
```

# Switch Statement Example

## Multiple case labels

Here: 5 & 6

## Fall through cases

Here: 2

## Missing cases

Here: 4

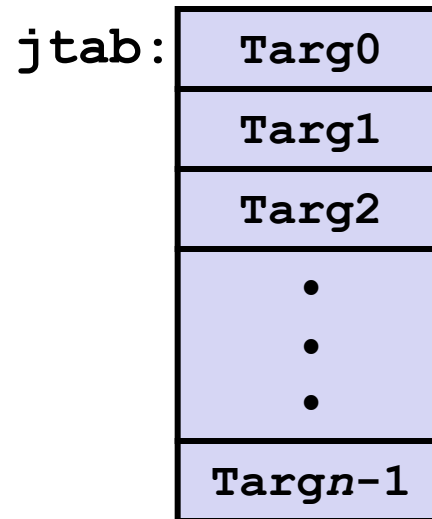


# Jump Table Structure

## Switch Form

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

## Jump Table



## Jump Targets

Targ0:

Code Block 0
-----------------

Targ1:

Code Block 1
-----------------

Targ2:

Code Block 2
-----------------

•  
•  
•

Targn-1:

Code Block n-1
-------------------

## Translation (Extended C)

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

# Switch Statement Example

```

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

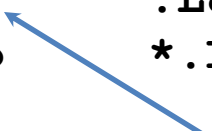
```

Setup:

```

switch_eg:
    movq    %rdx, %rcx
    cmpq    $6, %rdi    # x:6
    ja     .L8
    jmp     *.L4(, %rdi, 8)

```



What range of values  
takes default?

Register	Use(s)
%rdi	Argument <b>x</b>
%rsi	Argument <b>y</b>
%rdx	Argument <b>z</b>
%rax	Return value

Note that **w** not  
initialized here

# Switch Statement Example

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

## Setup:

```
switch_eg:
    movq    %rdx, %rcx
    cmpq    $6, %rdi      # x:6
    ja     .L8            # Use default
    jmp     *.L4(, %rdi, 8) # 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`

Only for  $0 \leq x \leq 6$

## Jump table

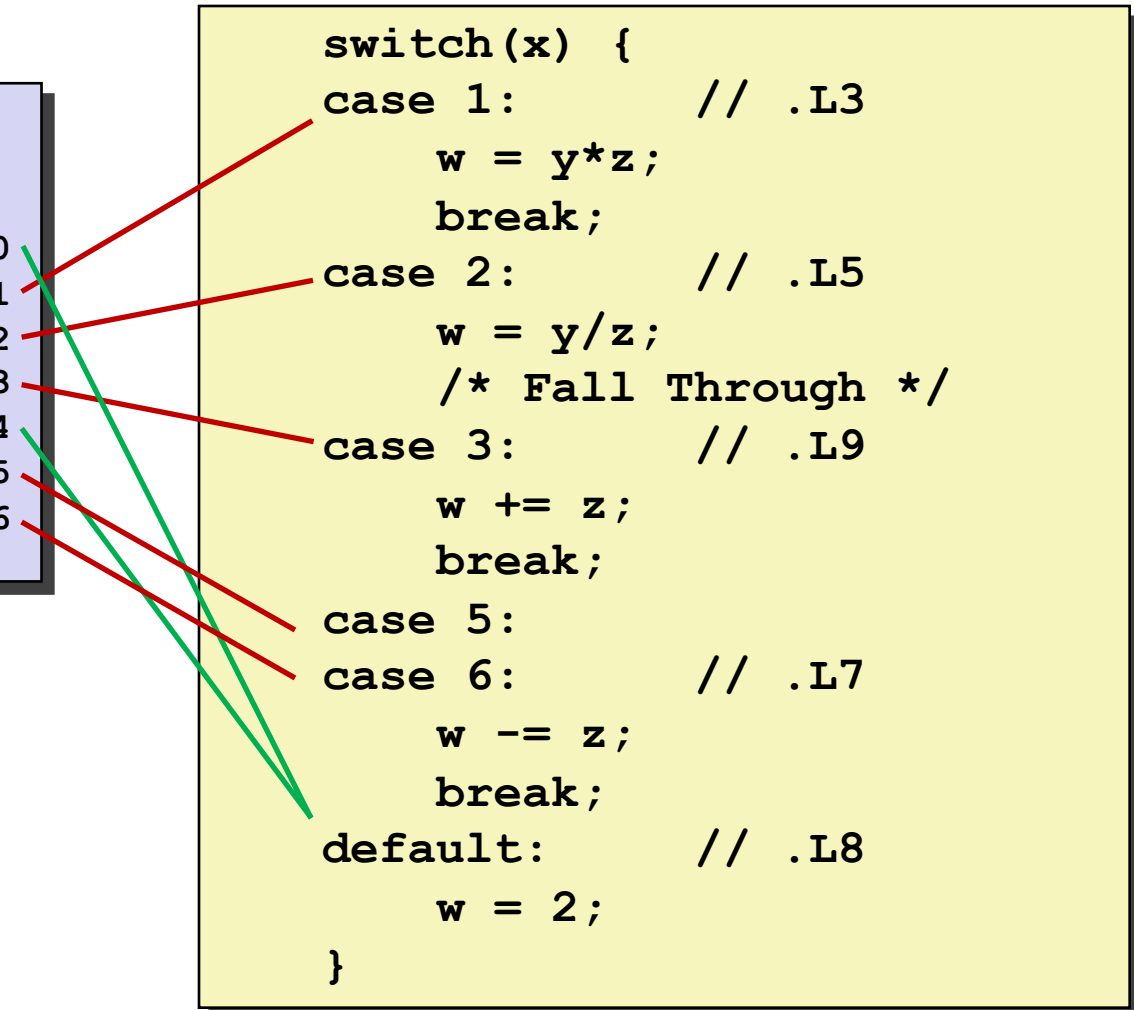
```
.section .rodata
        .align 8
.L4:
        .quad    .L8      # x = 0
        .quad    .L3      # x = 1
        .quad    .L5      # x = 2
        .quad    .L9      # x = 3
        .quad    .L8      # x = 4
        .quad    .L7      # x = 5
        .quad    .L7      # x = 6
```

# Jump Table

## 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;
case 2:      // .L5
    w = y/z;
    /* Fall Through */
case 3:      // .L9
    w += z;
    break;
case 5:
case 6:      // .L7
    w -= z;
    break;
default:    // .L8
    w = 2;
}
```



# Code Blocks (x == 1)

```

switch(x) {
case 1:      // .L3
    w = y*z;
    break;
    . . .
}

```

```

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

```

Register	Use(s)
%rdi	Argument <b>x</b>
%rsi	Argument <b>y</b>
%rdx	Argument <b>z</b>
%rax	Return value

# Handling Fall-Through

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

```
case 2:
    w = y/z;
    goto merge;
```

```
case 3:
    w = 1;
merge:
    w += z;
```

# Code Blocks (x == 2, x == 3)

```

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

```

```

.L5:                                # Case 2
    movq    %rsi, %rax
    cqto
    idivq   %rcx                    # y/z
    jmp     .L6                     # goto merge
.L9:                                # Case 3
    movl    $1, %eax                # w = 1
.L6:                                # merge:
    addq    %rcx, %rax              # w += z
    ret

```

Register	Use(s)
%rdi	Argument <b>x</b>
%rsi	Argument <b>y</b>
%rdx	Argument <b>z</b>
%rax	Return value



# Code Blocks (x == 5, x == 6, default)

```

switch(x) {
    . . .
    case 5: // .L7
    case 6: // .L7
        w -= z;
        break;
    default: // .L8
        w = 2;
}

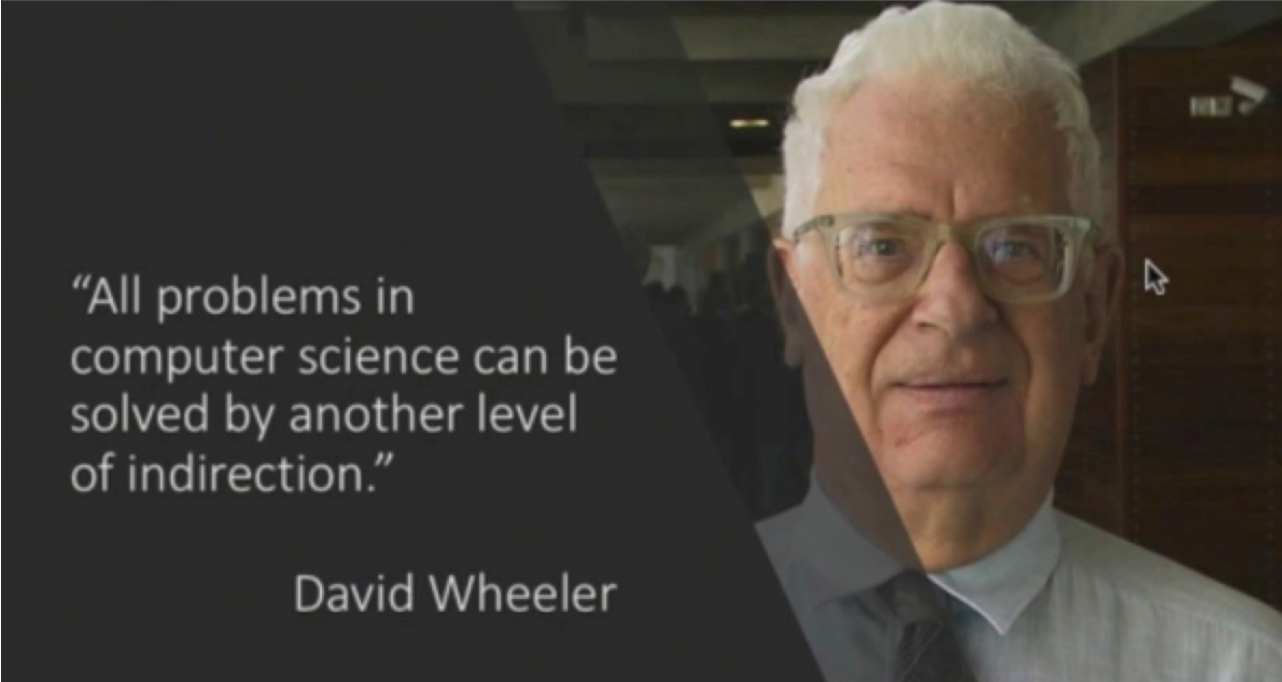
```

```

.L7:                                # Case 5,6
    movl    $1, %eax                # w = 1
    subq    %rdx, %rax              # w -= z
    ret
.L8:                                # Default:
    movl    $2, %eax                # 2
    ret

```

Register	Use(s)
<code>%rdi</code>	Argument <b>x</b>
<code>%rsi</code>	Argument <b>y</b>
<code>%rdx</code>	Argument <b>z</b>
<code>%rax</code>	Return value



“All problems in  
computer science can be  
solved by another level  
of indirection.”

David Wheeler

# Summarizing

## **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 (if-elseif-elseif-else)