

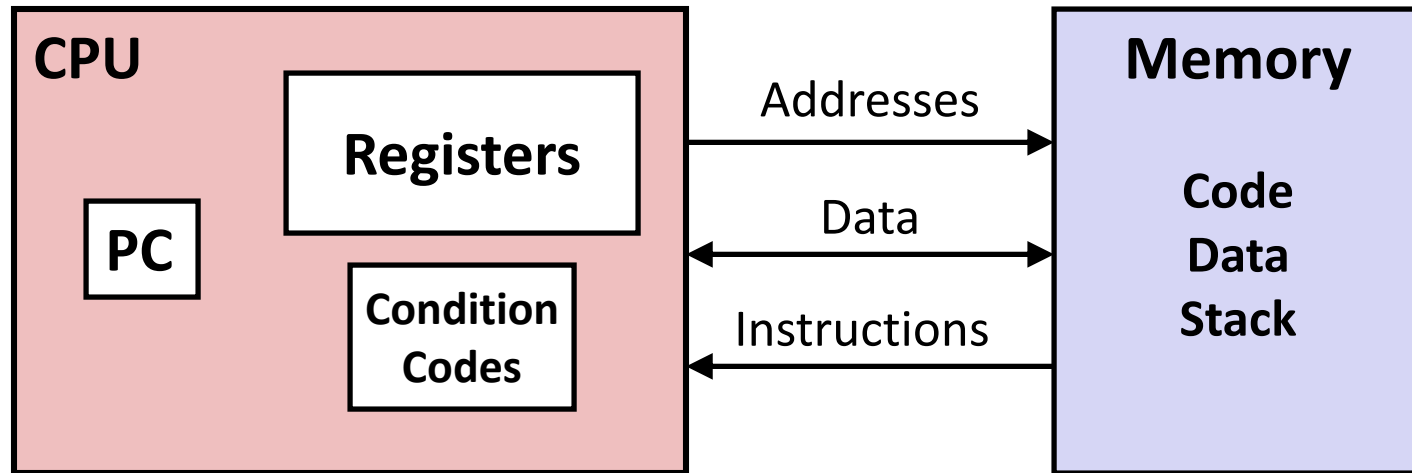
# Machine-Level Programming II: Control

**15-213/18-213/14-513/15-513/18-613: Introduction to Computer Systems**  
**6<sup>th</sup> Lecture, January 30, 2020**

# Today

- **Control: Condition codes**
- **Conditional branches**
- **Loops**
- **Switch Statements**

# Recall: ISA = Assembly/Machine Code View

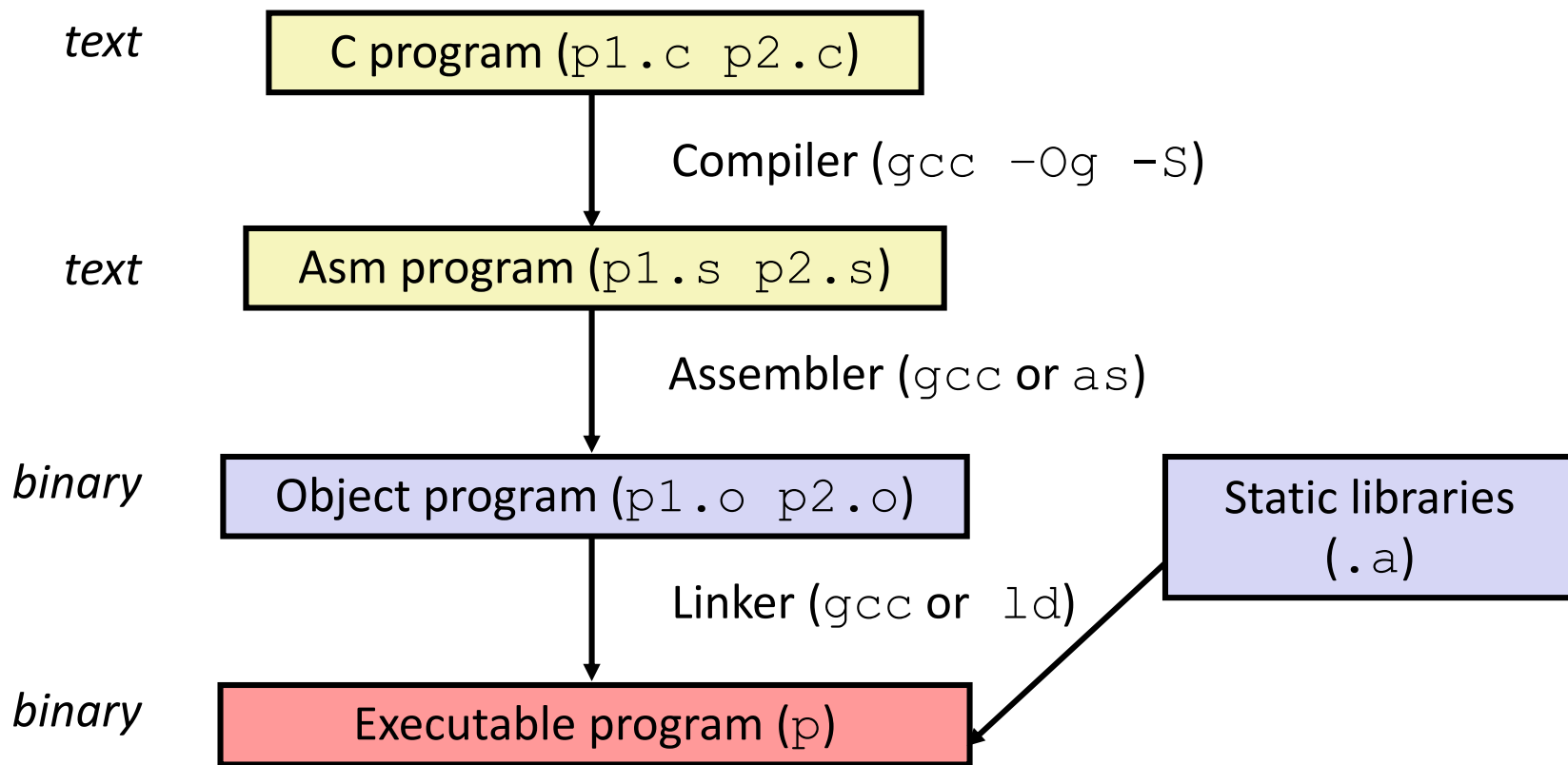


## Programmer-Visible State

- **PC: Program counter**
    - Address of next instruction
  - **Register file**
    - Heavily used program data
  - **Condition codes**
    - Store status information about most recent arithmetic or logical operation
    - Used for conditional branching
- **Memory**
    - Byte addressable array
    - Code and user data
    - Stack to support procedures

# Recall: Turning C into Object Code

- Code in files `p1.c` `p2.c`
- Compile with command: `gcc -Og p1.c p2.c -o p`
  - Use basic optimizations (`-Og`) [New to recent versions of GCC]
  - Put resulting binary in file `p`



# Recall: Move & Arithmetic Operations

## ■ Some Two Operand Instructions:

<i><b>Format</b></i>	<i><b>Computation</b></i>	
<code>movq</code>	<i>Src, Dest</i>	$\text{Dest} = \text{Src}$ (Src can be <code>\$const</code> )
<code>leaq</code>	<i>Src, Dest</i>	$\text{Dest} = \text{address computed by expression Src}$
<code>addq</code>	<i>Src, Dest</i>	$\text{Dest} = \text{Dest} + \text{Src}$
<code>subq</code>	<i>Src, Dest</i>	$\text{Dest} = \text{Dest} - \text{Src}$
<code>imulq</code>	<i>Src, Dest</i>	$\text{Dest} = \text{Dest} * \text{Src}$
<code>salq</code>	<i>Src, Dest</i>	$\text{Dest} = \text{Dest} \ll \text{Src}$
<code>sarq</code>	<i>Src, Dest</i>	$\text{Dest} = \text{Dest} \gg \text{Src}$
<code>shrq</code>	<i>Src, Dest</i>	$\text{Dest} = \text{Dest} \gg \text{Src}$
<code>xorq</code>	<i>Src, Dest</i>	$\text{Dest} = \text{Dest} \wedge \text{Src}$
<code>andq</code>	<i>Src, Dest</i>	$\text{Dest} = \text{Dest} \& \text{Src}$
<code>orq</code>	<i>Src, Dest</i>	$\text{Dest} = \text{Dest}   \text{Src}$

***Also called `shlq`***

***Arithmetic***

***Logical***

# Recall: Addressing Modes

## ■ Most General Form

**$D(Rb, Ri, S)$**

**$Mem[Reg[Rb] + S * Reg[Ri] + D]$**

- 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

## ■ Special Cases

**$(Rb, Ri)$**

**$Mem[Reg[Rb] + Reg[Ri]]$**

**$D(Rb, Ri)$**

**$Mem[Reg[Rb] + Reg[Ri] + D]$**

**$(Rb, Ri, S)$**

**$Mem[Reg[Rb] + S * Reg[Ri]]$**

# 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

`CF`

`ZF`

`SF`

`OF`

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/borrow 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

# SF set when

$$\begin{array}{r} \text{+} \begin{array}{|l|} \hline yxxxxxxxxxxxxxxxxx \dots \\ \hline yxxxxxxxxxxxxxxxxx \dots \\ \hline \end{array} \\ \hline \begin{array}{|l|} \hline 1xxxxxxxxxxxxxxxxx \dots \\ \hline \end{array} \end{array}$$

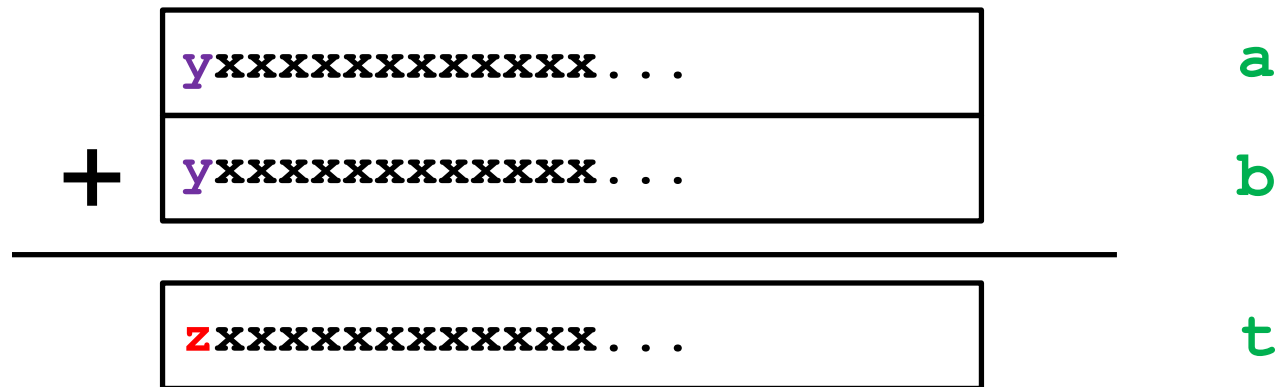
For signed arithmetic, this reports when result is a negative number

# CF set when



For unsigned arithmetic, this reports overflow

# OF set when



$$Z = \sim y$$

$(a > 0 \ \&\& \ b > 0 \ \&\& \ t < 0) \ || \ (a < 0 \ \&\& \ b < 0 \ \&\& \ t \geq 0)$

For signed arithmetic, this reports overflow

# Condition Codes (Explicit Setting: Compare)

## ■ Explicit Setting by Compare Instruction

- `cmpq Src2, Src1`
- `cmpq b, a` like computing `a-b` without setting destination
- **CF set** if carry/borrow 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) || (a<0 && b>0 && (a-b)>0)`

# Condition Codes (Explicit Setting: Test)

## ■ Explicit Setting by Test instruction

- `testq Src2, Src1`
  - `testq 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** when `a&b == 0`
- **SF set** when `a&b < 0`

Very often:

```
testq    %rax, %rax
```

# Condition Codes (Explicit Reading: Set)

## ■ Explicit Reading by Set Instructions

- **setX** *Dest*: Set low-order byte of destination *Dest* to 0 or 1 based on combinations of condition codes
- Does not alter remaining 7 bytes of *Dest*

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>

## Example: setl (Signed <)

■ **Condition: SF<sup>^</sup>OF**

SF	OF	SF ^ OF	Implication
0	0	0	No overflow, so SF implies not <
1	0	1	No overflow, so SF implies <
0	1	1	Overflow, so SF implies negative overflow, i.e. <
1	1	0	Overflow, so SF implies positive overflow, i.e. not <

## negative overflow case

Diagram illustrating a subtraction operation:

Box 1:  $1 \text{xxxxxxxxxxxxxxxx} \dots$  (labeled 'a' in green)

Box 2:  $0 \text{xxxxxxxxxxxxxxxx} \dots$  (labeled 'b' in green)

Minus sign:  $-$

Box 3:  $0 \text{xxxxxxxxxxxxxxxx} \dots$  (labeled 't' in green)



# x86-64 Integer Registers

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

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

- Can reference low-order byte

# Explicit Reading Condition Codes (Cont.)

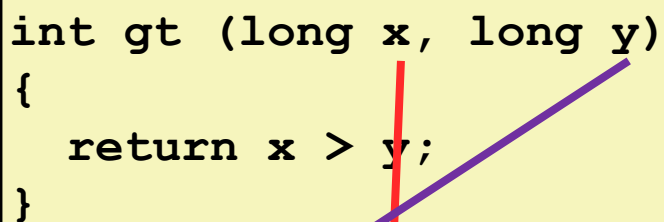
## ■ SetX Instructions:

- 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;
}
```



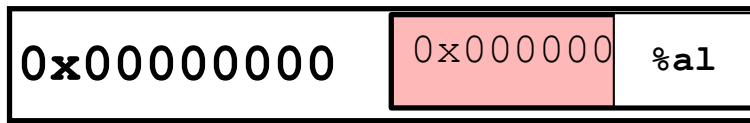
```
cmpq    %rsi, %rdi    # Compare x:y
setg     %al           # Set when >
movzbl  %al, %eax      # Zero rest of %rax
ret
```

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

# Explicit 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
```

# Today

- Control: Condition codes
- **Conditional branches**
- Loops
- Switch Statements

# Jumping

## ■ jX Instructions

- Jump to different part of code depending on condition codes
- Implicit reading of condition codes

jX	Condition	Description
<b>jmp</b>	<b>1</b>	<b>Unconditional</b>
<b>j<sub>e</sub></b>	<b>ZF</b>	<b>Equal / Zero</b>
<b>j<sub>ne</sub></b>	<b>~ZF</b>	<b>Not Equal / Not Zero</b>
<b>j<sub>s</sub></b>	<b>SF</b>	<b>Negative</b>
<b>j<sub>ns</sub></b>	<b>~SF</b>	<b>Nonnegative</b>
<b>j<sub>g</sub></b>	<b>~ (SF^OF) &amp; ~ZF</b>	<b>Greater (signed)</b>
<b>j<sub>ge</sub></b>	<b>~ (SF^OF)</b>	<b>Greater or Equal (signed)</b>
<b>j<sub>l</sub></b>	<b>SF^OF</b>	<b>Less (signed)</b>
<b>j<sub>le</sub></b>	<b>(SF^OF)   ZF</b>	<b>Less or Equal (signed)</b>
<b>j<sub>a</sub></b>	<b>~CF &amp; ~ZF</b>	<b>Above (unsigned)</b>
<b>j<sub>b</sub></b>	<b>CF</b>	<b>Below (unsigned)</b>

# Conditional Branch Example (Old Style)

## ■ Generation

shark> gcc -Og -S **-fno-if-conversion** control.c

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

```
n_test = !Test;  
if (n_test) 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

```

When is  
this bad?

# Bad Cases for Conditional Move

## Expensive Computations

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

Bad Performance

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

## Risky Computations

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

Unsafe

- Both values get computed
- May have undesirable effects

## Computations with side effects

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

Illegal

- Both values get computed
- Must be side-effect free

# Today

- Control: Condition codes
- Conditional branches
- **Loops**
- 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

```

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

```

# Quiz Time!

Check out:

<https://canvas.cmu.edu/courses/13182/quizzes/31655>

# General “Do-While” Translation

## C Code

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

## Goto Version

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

■ **Body:** {  
    Statement<sub>1</sub>;  
    Statement<sub>2</sub>;  
    ...  
    Statement<sub>n</sub>;  
}



# 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;
}
```

- Initial conditional guards entrance to loop
- Compare to do-while version of function
  - Removes jump to middle. When is this good or bad?

# “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

## C Code

## Goto Version

```
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 – **why?**

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



# Today

- Control: Condition codes
- Conditional branches
- Loops
- **Switch Statements**

```
long my_switch
(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

jtab:	Targ0
	Targ1
	Targ2
	•
	•
	Targn-1

## Jump Targets

Targ0:

Code Block  
0

Targ1:

Code Block  
1

Targ2:

Code Block  
2

•  
•  
•

Targn-1:

Code Block  
n-1

## Translation (Extended C)

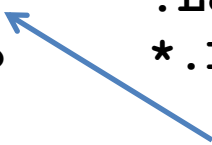
```
goto *JTab[x];
```

# Switch Statement Example

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

## Setup

```
my_switch:
    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 my_switch(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

```
my_switch:
    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;

    . . .
}
```

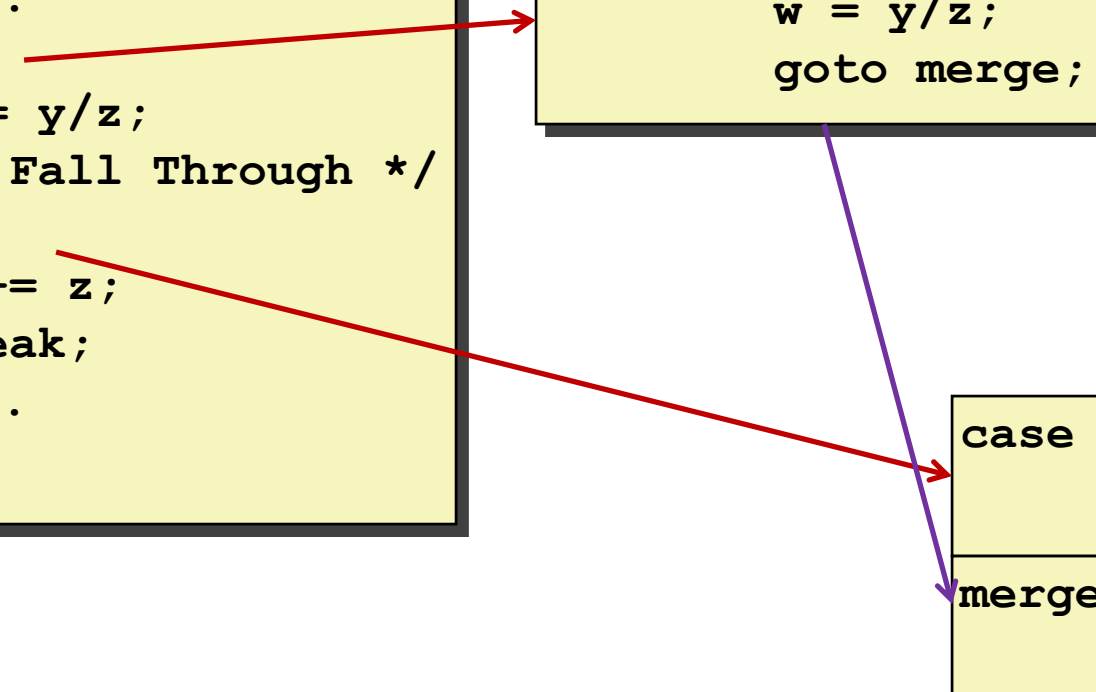
```
.L3:
    movq    %rsi, %rax    # y
    imulq   %rdx, %rax    # y*z
    ret
```

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                                # sign extend
                                # rax to rdx:rax
    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>
%rcx	<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)
%rdi	Argument <b>x</b>
%rsi	Argument <b>y</b>
%rdx	Argument <b>z</b>
%rax	Return value

# 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)

# Summary

## ■ Today

- Control: Condition codes
- Conditional branches & conditional moves
- Loops
- Switch statements

## ■ Next Time

- Stack
- Call / return
- Procedure call discipline

# Finding Jump Table in Binary

```

00000000004005e0 <switch_eg>:
4005e0:    48 89 d1                mov     %rdx,%rcx
4005e3:    48 83 ff 06             cmp     $0x6,%rdi
4005e7:    77 2b                   ja      400614 <switch_eg+0x34>
4005e9:    ff 24 fd f0 07 40 00    jmpq    *0x4007f0(,%rdi,8)
4005f0:    48 89 f0                mov     %rsi,%rax
4005f3:    48 0f af c2             imul    %rdx,%rax
4005f7:    c3                      retq
4005f8:    48 89 f0                mov     %rsi,%rax
4005fb:    48 99                   cqto
4005fd:    48 f7 f9                idiv    %rcx
400600:    eb 05                   jmp     400607 <switch_eg+0x27>
400602:    b8 01 00 00 00          mov     $0x1,%eax
400607:    48 01 c8                add     %rcx,%rax
40060a:    c3                      retq
40060b:    b8 01 00 00 00          mov     $0x1,%eax
400610:    48 29 d0                sub     %rdx,%rax
400613:    c3                      retq
400614:    b8 02 00 00 00          mov     $0x2,%eax
400619:    c3                      retq

```

# Finding Jump Table in Binary (cont.)

```
00000000004005e0 <switch_eg>:
```

```
. . .
```

```
4005e9:      ff 24 fd f0 07 40 00      jmpq    *0x4007f0(,%rdi,8)
```

```
. . .
```

```
% gdb switch
```

```
(gdb) x /8xg 0x4007f0
```

0x4007f0:	0x0000000000400614	0x00000000004005f0
0x400800:	0x00000000004005f8	0x0000000000400602
0x400810:	0x0000000000400614	0x000000000040060b
0x400820:	0x000000000040060b	0x2c646c25203d2078

```
(gdb)
```

# Finding Jump Table in Binary (cont.)

```
% gdb switch
(gdb) x /8xg 0x4007f0
0x4007f0:      0x000000000000400614      0x0000000000004005f0
0x400800:      0x0000000000004005f8      0x000000000000400602
0x400810:      0x000000000000400614      0x00000000000040060b
0x400820:      0x00000000000040060b      0x2c646c25203d2078
```

4005f0:	48 89 f0	mov	%rsi,%rax
4005f3:	48 0f af c2	imul	%rdx,%rax
4005f7:	c3	retq	
4005f8:	48 89 f0	mov	%rsi,%rax
4005fb:	48 99	cqto	
4005fd:	48 f7 f9	idiv	%rcx
400600:	eb 05	jmp	400607 <switch_eg+0x27>
400602:	b8 01 00 00 00	mov	\$0x1,%eax
400607:	48 01 c8	add	%rcx,%rax
40060a:	c3	retq	
40060b:	b8 01 00 00 00	mov	\$0x1,%eax
400610:	48 29 d0	sub	%rdx,%rax
400613:	c3	retq	
400614:	b8 02 00 00 00	mov	\$0x2,%eax
400619:	c3	retq	