

# CS 105

"Tour of the Black Holes of Computing"

## Machine-Level Programming II: Control Flow

### Topics

- Condition codes
- Conditional branches
- Loops
- Switch statements

## Condition Codes (Implicit Setting)

### Single-bit registers

CF Carry Flag (for unsigned)      SF Sign Flag (for signed)  
ZF Zero Flag                              OF Overflow Flag (signed)

### Implicitly set (as side effect) by arithmetic operations

`addq Src, Dest`

C analog: `dest += src;`

- CF set if carry out from most significant bit
  - Detects unsigned overflow; also used for multiprecision arithmetic
- ZF set if `src+dest == 0`
- SF set if `src+dest < 0`
- OF set if two's complement overflow
  - `(src > 0 && dest > 0 && src + dest < 0) \`  
`|| (src < 0 && dest < 0 && src + dest >= 0)`

**Not set by `leaq` instruction**

CS 105

## Condition Codes (Explicit Setting)

### Explicit setting by Compare instruction

`cmpq Src2, Src1`

- `cmpq b, a` like computing `a-b` without setting destination
  - Note reversed operand order!
- CF set if carry out from most significant bit
  - Used for unsigned comparisons
  - Also good for multi-precision arithmetic (at assembly level)
- ZF set if `a == b`
- SF set if `(a-b) < 0`
- OF set if two's complement overflow
  - `(a > 0 && b < 0 && (a - b) < 0) || (a < 0 && b > 0 && (a - b) > 0)`

- 3 -

CS 105

## Condition Codes (Explicit Setting)

### Explicit setting by Test instruction

`testq Src1, Src2`

- Sets condition codes based on value of `Src1 & Src2`
  - Intel thought it useful to have one operand be a mask
  - Compiler usually sets `Src1` and `Src2` the same
- `testq a, b` like computing `a&b` without setting destination
- ZF set when `a&b == 0`
- SF set when `a&b < 0`
- CF, OF unaffected (not cleared!)
- Most common usage: `testq %rax, %rax`
  - Sets ZF if `%rax == 0`, SF if `%rax < 0`
  - i.e., "Is `%rax` zero, negative, or positive?"
- Usually immediately precedes jump instruction (covered in a few minutes)

This is the one that matters!

- 4 -

CS 105

## Reading Condition Codes

### SetX instructions

- Set single byte based on combinations of condition codes
- Remaining 7 bytes unaltered!

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

- 5 -

CS 105

## x86-64 Integer Registers

%rax	%al	%r8	%r8b
%rbx	%bl	%r9	%r9b
%rcx	%cl	%r10	%r10b
%rdx	%dl	%r11	%r11b
%rsi	%sil	%r12	%r12b
%rdi	%dil	%r13	%r13b
%rsp	%spl	%r14	%r14b
%rbp	%bpl	%r15	%r15b

- Can reference low-order byte

- 6 -

CS 105

## Reading Condition Codes (Cont.)

### SetX instructions

- Set single byte based on combinations of condition codes

### One of 8 addressable *byte* registers

- Does not alter remaining 7 bytes!
- Typically use `movzbl` to finish job
  - "1" instructions also set upper 32 bits (of 64) to 0

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

Register	Use(s)
%rdi	Argument x
%rsi	Argument y
%rax	Return value

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

Note inverted ordering!

- 7 -

CS 105

## Jumping

### jX instructions

- Jump to different part of code depending on condition codes

jX	Condition	Description
jmp	1	Unconditional
je/jz	ZF	Equal / Zero
jne/jnz	~ZF	Not Equal / Not Zero
js	SF	Negative
jns	~SF	Nonnegative
kg	~(SF^OF) & ~ZF	Greater (signed)
jge	~(SF^OF)	Greater or Equal (signed)
jl	(SF^OF)	Less (signed)
jle	(SF^OF)   ZF	Less or Equal (signed)
ja	~CF & ~ZF	Above (unsigned)
jb	CF	Below (unsigned)

- 8 -

CS 105

## Conditional-Branch Example (Old Style)



### Generation

wilkes> gcc -Og -S -fno-if-conversion control.c

```
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 x
%rsi	Argument y
%rax	Return value

- 9 -

CS 105

## Expressing with Goto Code



C allows goto statement

Jump to position designated by label

Sinful!

```
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;
    return result;
Else:
    result = y-x;
    return result;
}
```

- 10 -

CS 105

## 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

- 11 -

CS 105

## 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 known to be safe

### Why?

- Branches are disruptive to instruction flow through pipelines
- Branches can kill performance
- 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;
```

- 12 -

CS 105

## 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 x
%rsi	Argument y
%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
```

- 13 -

CS 105

## Bad Cases for Conditional Move



### Expensive Computations

```
val = test(x) ? hard_1(x) : hard_2(x);
```

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

### Risky Computations

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

- Both values get computed
- May have undesirable effects

### Computations with side effects

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

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

- 14 -

CS 105

## “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” or “bitcount”)

Use conditional branch to either continue looping or to exit loop

- 15 -

CS 105

## “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 x
%rax	result

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

- 16 -

CS 105

## General “Do-While” Translation



C Code

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

Body: {

```
  Statement1;
  Statement2;
  ...
  Statementn;
}
```

Goto Version

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

- 17 -

CS 105

## 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:
```

- 18 -

CS 105

## 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 Version

```
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

- 19 -

CS 105

## 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` and above

Goto Version

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

- 20 -

CS 105

## 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(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;
}
```

### 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;
}
```

```
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;
        result += bit; Body
    }
    i++; Update
    if (i < WSIZE)
        goto loop; Test
done:
    return result;
}
```

Initial test can often be optimized away

- 25 -

CS 105

## Switch Statement Example

```
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:
        w -= z;
        break;
    case 6:
        w = 2;
        break;
    default:
        w = 2;
        break;
    }
    return w;
}
```

Multiple case labels

■ Here: 5 & 6

Fall-through cases

■ Here: 2

Missing cases

■ Here: 4

- 26 -

CS 105

## Jump Table Structure

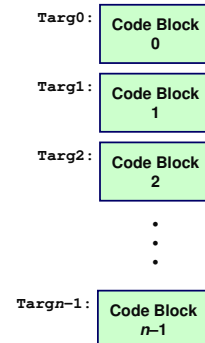
Switch Form

```
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



Approximate Translation

```
goto *Jtab[x];
```

- 27 -

CS 105

## 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 option?

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

Note that w is not initialized here!

- 28 -

CS 105

## 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)
```

Indirect  
jump

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
```

- 29 -

CS 105

## 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
```

- 30 -

CS 105

## 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;
}
```

- 31 -

CS 105

## Code Blocks (x == 1)

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

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

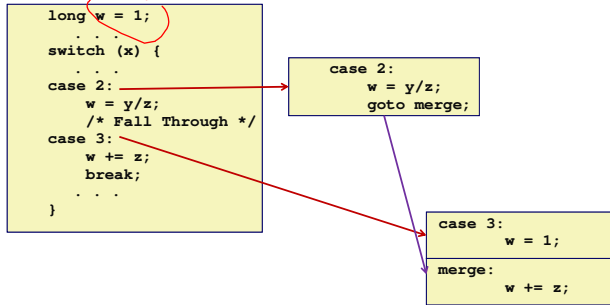
Register	Use(s)
<code>%rdi</code>	Argument <code>x</code>
<code>%rsi</code>	Argument <code>y</code>
<code>%rdx</code>	Argument <code>z</code>
<code>%rax</code>	Return value

- 32 -

CS 105



## Handling Fall-Through



- 33 -

CS 105

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

```

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

```

```

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

```

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

- 34 -

CS 105

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

```

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

```

```

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

```

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

- 35 -

CS 105

## Sparse Switches

What if jump table is too large?

- Compiler uses if-then-else structure
- Ternary tree gives  $O(\log n)$  performance

- 36 -

CS 105