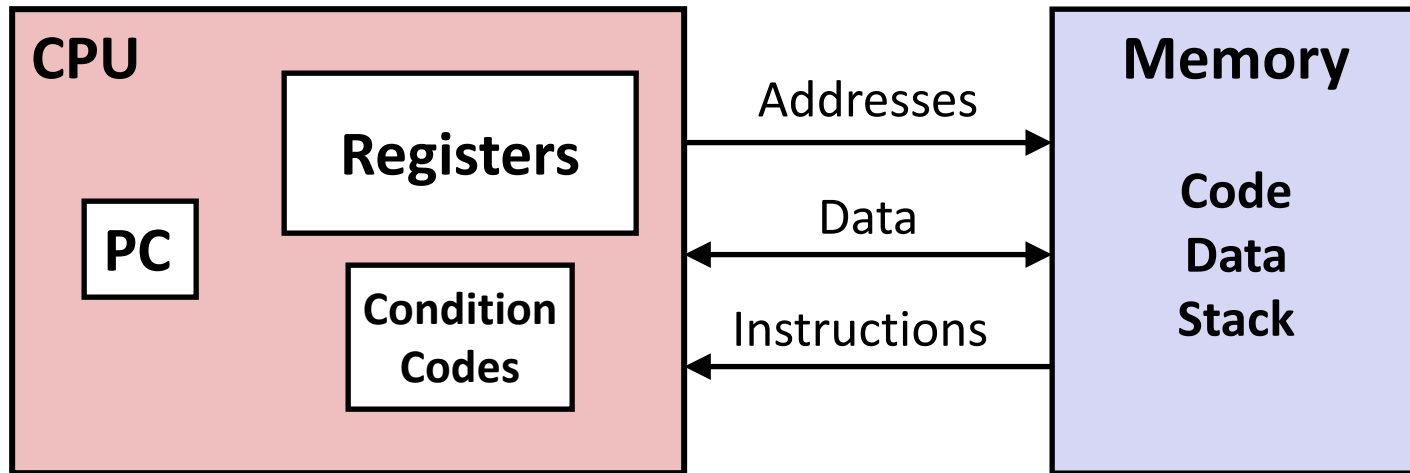# Machine-Level Programming II: Control

15-213/18-213/14-513/15-513: Introduction to Computer Systems
6th Lecture, May 28, 2020

Today

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

Recall: ISA = Assembly/Machine Code View
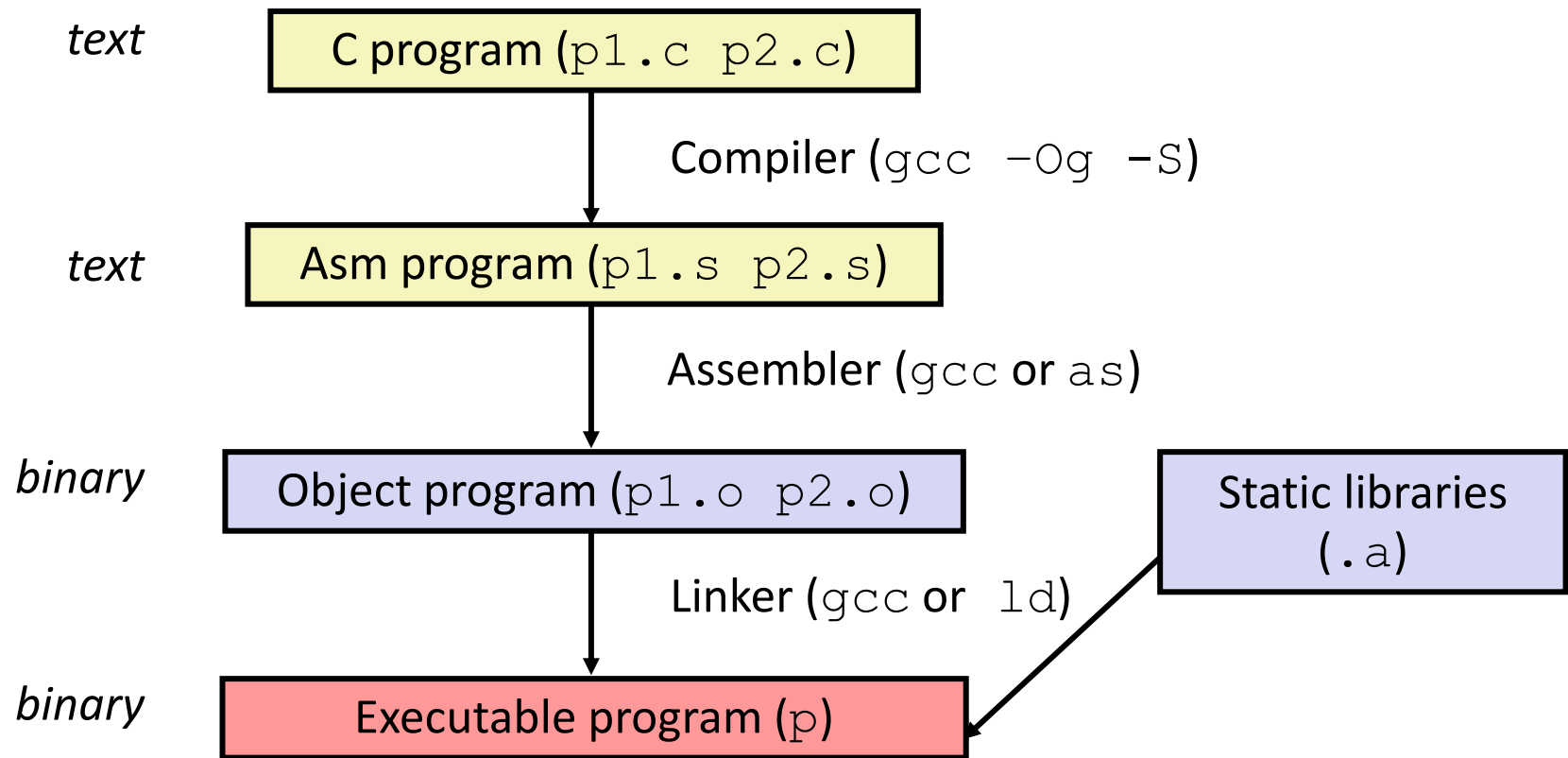

Programmer-Visible State

- **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`

text   C program (`p1.c p2.c`)

         ↓   Compiler (`gcc –Og –S`)

text   Asm program (`p1.s p2.s`)

         ↓   Assembler (`gcc` or `as`)

binary   Object program (`p1.o p2.o`)   Static libraries (`.a`)

         ↓   Linker (`gcc` or `ld`)

binary   Executable program (`p`)

Recall: Move & Arithmetic Operations

- ## Some Two Operand Instructions:

Format		*Computation*	
`movq`	*Src,Dest*	Dest = Src (Src can be $const)	
`leaq`	*Src,Dest*	Dest = address computed by expression Src	
`addq`	*Src,Dest*	Dest = Dest + Src	
`subq`	*Src,Dest*	Dest = Dest − Src	
`imulq`	*Src,Dest*	Dest = Dest * Src	
`salq`	*Src,Dest*	Dest = Dest << Src	***Also called shlq***
`sarq`	*Src,Dest*	Dest = Dest >> Src	***Arithmetic***
`shrq`	*Src,Dest*	Dest = Dest >> Src	***Logical***
`xorq`	*Src,Dest*	Dest = Dest ^ Src	
`andq`	*Src,Dest*	Dest = Dest & Src	
`orq`	*Src,Dest*	Dest = Dest | Src	

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]**

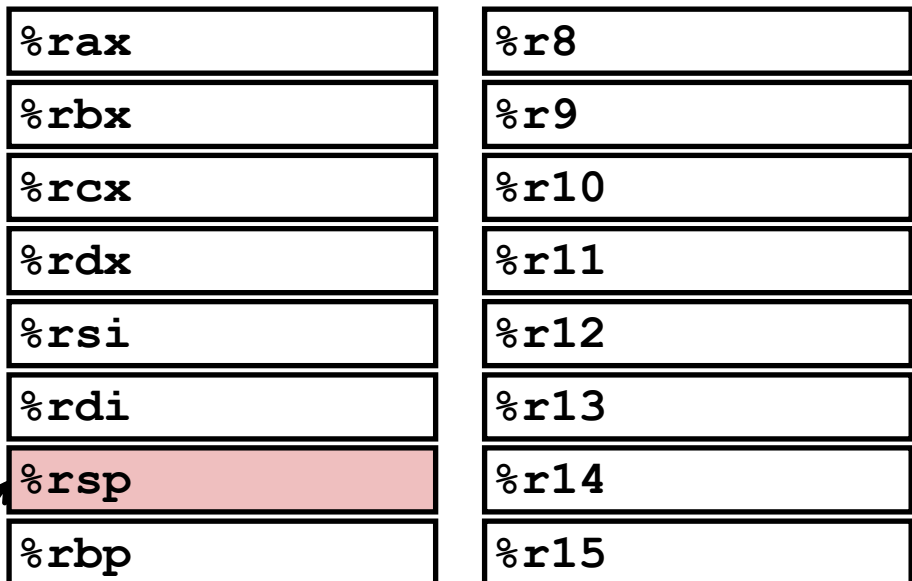**(Rb,Ri,S)**   **Mem[Reg[Rb]+S*Reg[Ri]]**

Processor State (x86-64, Partial)
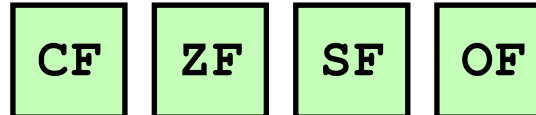
- **Information about currently executing program**
  - Temporary data (**%rax**, …)
  - Location of runtime stack (**%rsp**)
  - Location of current code control point (**%rip**, …)
  - Status of recent tests (**CF, ZF, SF, OF**)

Registers

`%rax`		`%r8`	
`%rbx`		`%r9`	
`%rcx`		`%r10`	
`%rdx`		`%r11`	
`%rsi`		`%r12`	
`%rdi`		`%r13`	
`%rsp`		`%r14`	
`%rbp`		`%r15`	

Current stack top

`%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* ⟷ `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…00000000000

SF set when

```
   yxxxxxxxxxxxxx...

+  yxxxxxxxxxxxxx...
_____

   1xxxxxxxxxxxxx...
```

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

CF set when

```
  1xxxxxxxxxxxx...
+ 1xxxxxxxxxxxxx...
─────────────────────
1 xxxxxxxxxxxxxx...
```
Carry

```
1 0xxxxxxxxxxxx...
- 1xxxxxxxxxxxx...
─────────────────────
  1xxxxxxxxxxxx...
```
Borrow

For unsigned arithmetic, this reports overflow

OF set when

a

+ b

t
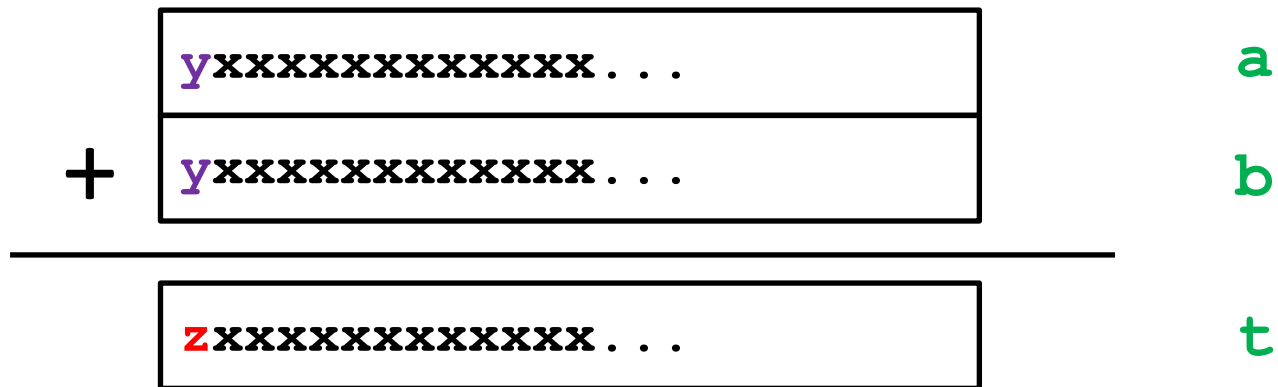
z = ~y

(a>0 && b>0 && t<0) || (a<0 && b<0 && t>=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
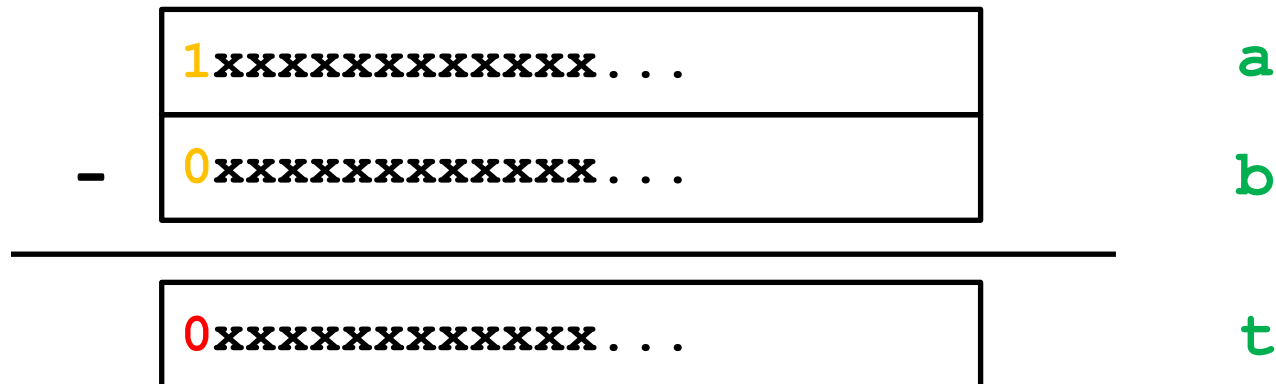 - Does not alter remaining 7 bytes of *Dest*

SetX	Condition	Description
`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)**

Example: setl (Signed <)

- **Condition: SF^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

```
    1xxxxxxxxxxxx...        a

  - 0xxxxxxxxxxxx...        b
  _____

    0xxxxxxxxxxxx...        t
```

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

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;
}
```

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

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

Explicit Reading Condition Codes (Cont.)

Beware weirdness **movzbl** (and others)

movzbl %al, %eax

0x00000000	0x000000	%al
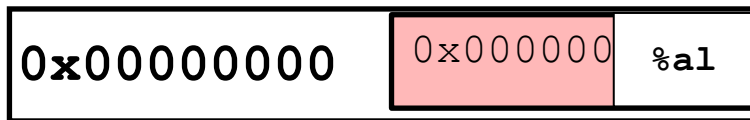
Zapped to all 0's

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
`jmp`	`1`	Unconditional
`je`	`ZF`	Equal / Zero
`jne`	`~ZF`	Not Equal / Not Zero
`js`	`SF`	Negative
`jns`	`~SF`	Nonnegative
`jg`	`~(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)

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 x
%rsi	Argument y
%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 ← 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 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
```

When is this bad?

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

- **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**

x86 being CISC has a popcount instruction

"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 **x**
`%rax`	`result`

```
        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);
```

Goto Version

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

- **Body:** {
  $Statement_1;$
  $Statement_2;$
  …
  $Statement_n;$
  }

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" conversion**
- **Used with −O1**

Do-While Version

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

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;
}
```

Bryant and O'Hallaron, Computer Systems: A Programmer's Perspective, Third Edition

"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 – why?**

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

Today

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

Switch Statement Example

```
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;
}
```
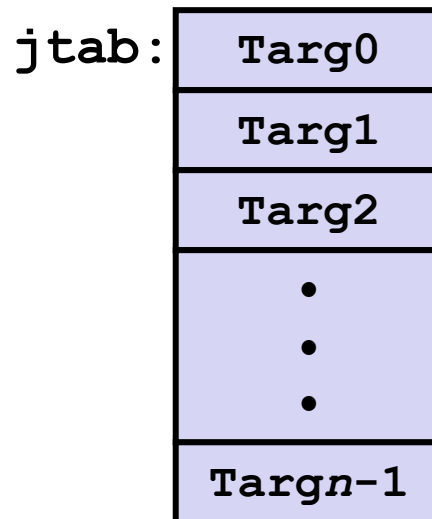
- **Multiple case labels**
  - Here: 5 & 6
- **Fall through cases**
  - Here: 2
- **Missing cases**
  - Here: 4

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)
```

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

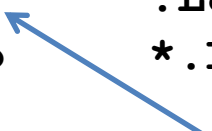**What range of values takes default?**

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 x
%rsi	Argument y
%rdx	Argument z
%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 x
%rsi	Argument y
%rcx	z
%rax	Return value

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

```c
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 x
%rsi	Argument y
%rdx	Argument z
%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:       0x0000000000400614      0x00000000004005f0
0x400800:       0x00000000004005f8      0x0000000000400602
0x400810:       0x0000000000400614      0x000000000040060b
0x400820:       0x000000000040060b      0x2c646c25203d2078
```

```
        . . .
    4005f0:         48 39 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
```