Machine-Level Programming II: Control

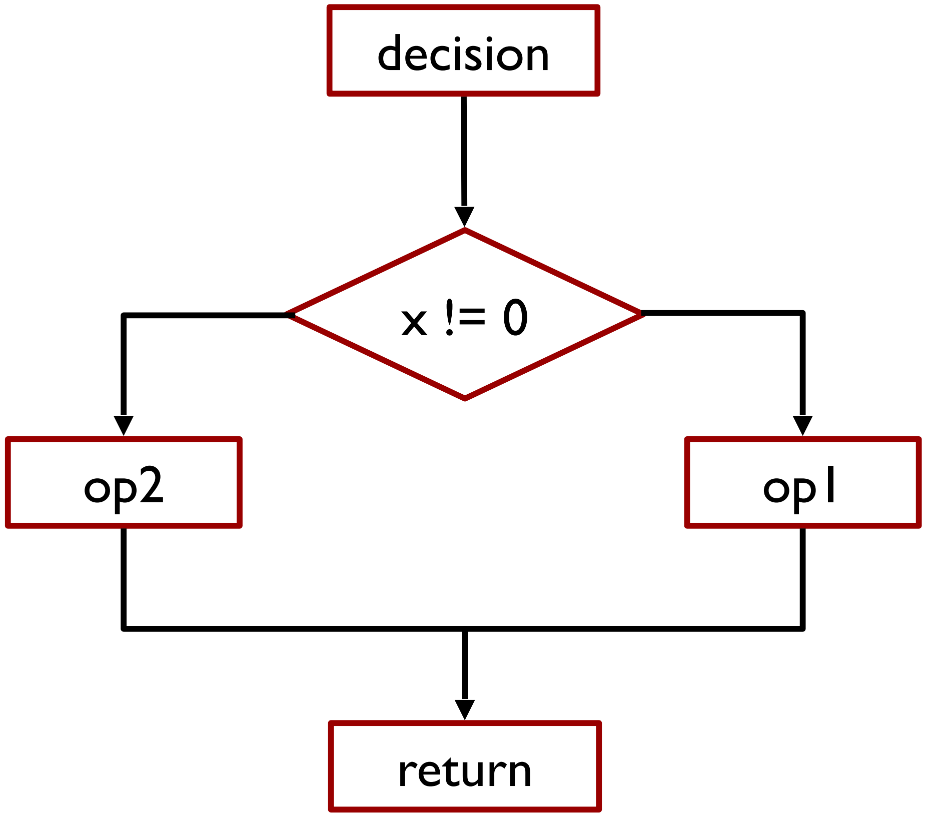
CS2011: Introduction to Computer Systems
Lecture 7 (3.6)

Machine-Level Programming II: Control

- Basics of control flow
- Condition codes
- Conditional operations
- Loops
- Switch statements

Control flow

```
extern void op1(void);
extern void op2(void);

void decision(int x) {
    if (x) {
        op1();
    } else {
        op2();
    }
}
```



Control flow in assembly language

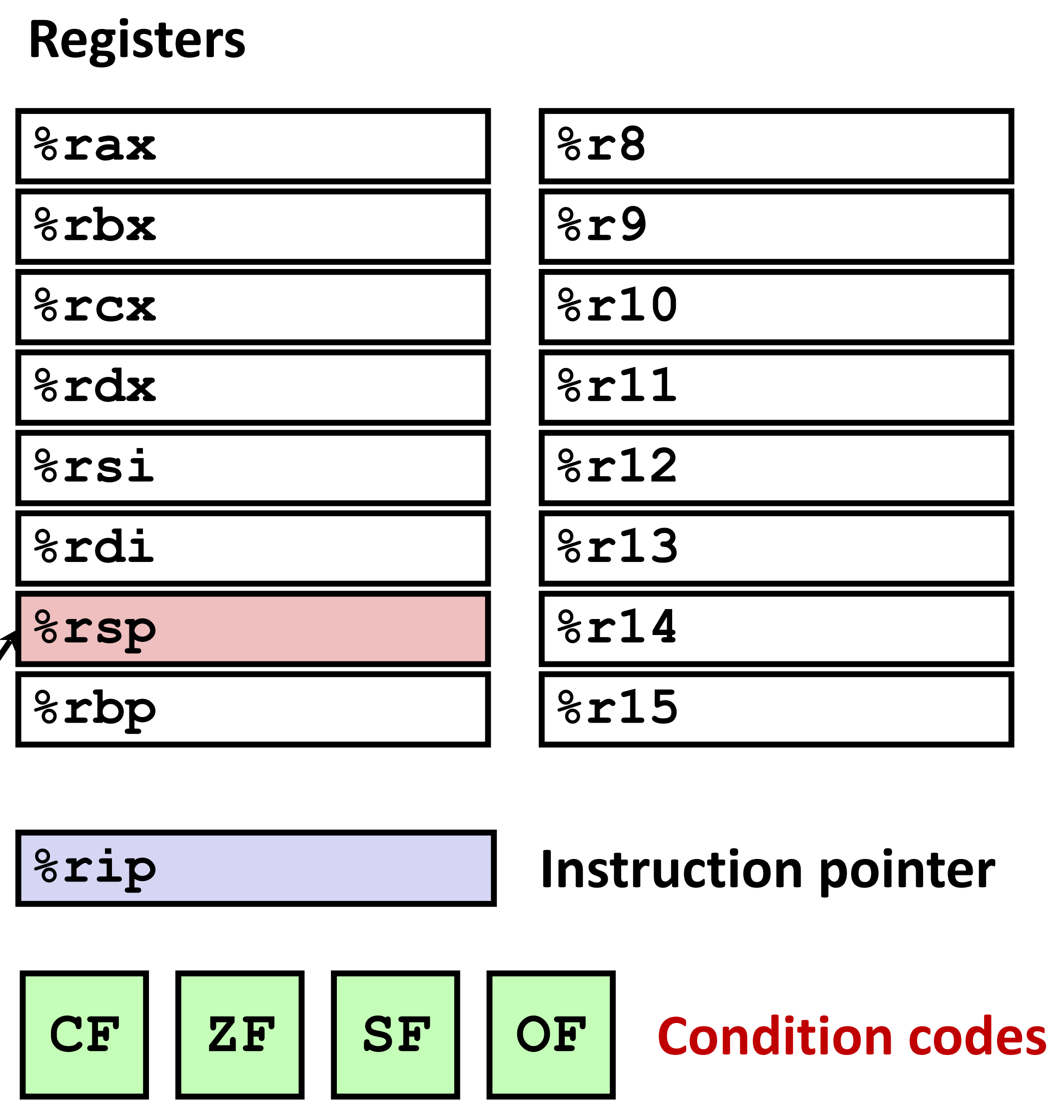
```
extern void op1(void);
extern void op2(void);

void decision(int x) {
    if (x) {
        op1();
    } else {
        op2();
    }
}
```

```
decision:
        subq    $8, %rsp
        testl   %edi, %edi
        je      .L2
        call    op1
        jmp     .L1
.L2:
        call    op2
.L1:
        addq    $8, %rsp
        ret
```

It's all done with GOTO!

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**)



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**)
  - **GDB prints these as one "eflags" register**

`eflags 0x246 [ PF ZF IF ]` *ZF is set, CSO clear*

```
(gdb) break 6
Breakpoint 1 at 0x1182: file main.c, line 6.
(gdb) run
(gdb) info registers eflags
eflags         0x246               [ PF ZF IF ]
```

Contents of the **32-bit e**flags register is 0x246 (if any of the interesting flags are set, it will show inside [ ])

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) after arithmetic operations**

  Example: `addq` *Src,Dest* ↔ `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)
  a and b positive -> t negative, or,
  a and b negative -> t positive
  ```
- **Not set by `leaq` instruction**

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 after logical operations**
  - **CF and OF are always set to 0**
- **Implicitly set after shift operations**
  - **CF: set to the last bit shifted out**
  - **OF: always set to 0**
- **Implicitly set after INC/DEC operations (will not discuss why)**
  - **ZF and OF are always set according to the result**
  - **CF: stays unchanged**

ZF set when

000000000000…00000000000

SF set when

`1xxxxxxxxxxxx...xxxxxxxxxxxx`

For **signed** arithmetic, this means result is negative
For unsigned arithmetic, this does not tell us much

CF set when

```
    yxxxxxxxxxxxx...
+   yxxxxxxxxxxxx...
--------------------
1   zxxxxxxxxxxxx...
```

For **unsigned** arithmetic, this reports **overflow**
For signed arithmetic, this does not tell us much

CF set when

```
   1xxxxxxxxxxxxx...
+  1xxxxxxxxxxxxx...
--------------------
1  xxxxxxxxxxxxxx...
```

Carry

```
1  0xxxxxxxxxxxxx...
-  1xxxxxxxxxxxxx...
--------------------
   1xxxxxxxxxxxxx...
```

Borrow

OF set when

```
  wxxxxxxxxxxxx...
+ yxxxxxxxxxxxx...
------------------
  zxxxxxxxxxxxx...
```

w == y && w != z

For **signed** arithmetic, this reports **overflow**
For unsigned arithmetic, this does not tell us much

Compare Instruction

- `cmp a, b`
  - Computes $b - a$ (just like `sub`)
  - Explicitly **sets condition codes** based on result, but…
  - **Does not change b**

CF set if carry out from most significant bit (used for **unsigned** comparisons)

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

Test Instruction

- **`test a, b`**
  - Computes **b&a** (just like **`and`**)
  - Explicitly **sets condition codes** (**only SF and ZF**) based on result, but…
  - **Does not change *b***

ZF set if **b & a** == 0

SF set if **b & a** < 0

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

- Second most common use: `test %rX, %rY`
  tests if any of the 1-bits in `%rY` are also 1 in `%rX` (or vice versa)
  one of the operands is a **mask** indicating which bits should be tested

Machine-Level Programming II: Control

- Review of a few tricky bits from yesterday
- Basics of control flow
- Condition codes
- **Conditional operations**
- Loops
- Switch statements

Reading Condition Codes

SetX Instructions

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

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)

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

- 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;
}
```

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

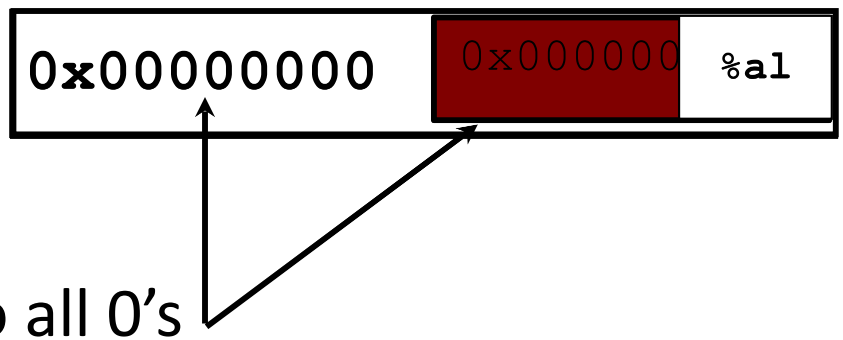
```
cmpq   %rsi, %rdi    # Compare x:y
setg   %al           # Set when >
movzbl %al, %eax     # Zero rest of %rax
ret
```

Reading Condition Codes (Cont.)

- SetX Instructions:

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 "using **labels**" depending on 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)

- Only **indirect jump** is jmp *Operand (e.g., jmp *%rax, jmp *(%rax))

Conditional Branch Example (Old Style)

- Generation

```
linux> gcc –Og -S –fno-if-conversion control.c
```

Not always needed

```
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

.L4 is a label in Assembly. Assembler determines address of .L4 and encodes the jump target (address of dest instruction) as part of **jle .L4** instruction

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;
}
```

Using goto function in C is not recommended. It can easily introduce vulnerable code and can make code very difficult to read and debug

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?

- Modern processors use **pipelining** to improve performance. They employ **branch prediction logic** to reliably **predict** "90% of the time" instructions to execute
  - Mis-prediction can cause **15-30 wasted clock cycles**.
- **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

cmov **(conditional** move**):** operands can be 16, 32, or 64 bits long. Single-byte is not supported. Assembler infers size of operand

```
absdiff:
   movq    %rdi, %rax   # x
   subq    %rsi, %rax   # result = x-y
   movq    %rsi, %rdx
   subq    %rdi, %rdx   # eval = y-x
   cmpq    %rsi, %rdi   # compare x:y
   cmovle  %rdx, %rax   # if <=, result = eval
   ret
```

Conditional Move Drawback (E.g., side effect)

```
long lt_cnt = 0;
long ge_cnt = 0;

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

Both **lt_cnt** and **ge_cnt** values will be updated as a side effect of using conditional move to implement conditional branches

All Bad Cases for Conditional Move

Expensive Computations

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

Bad Performance

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

Risky Computations

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

Unsafe

- Both values get computed/evaluated
- May have undesirable effects (if p is null, dereferencing null pointer error)

Computations with side effects

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

Illegal

- Both values get computed/evaluated —> x is updated twice
- Must be side-effect free

Exercise

`cmpq b,a` like computing `a-b` without setting dest

- **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

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)

```
xorq    %rax, %rax
subq    $1, %rax
cmpq    $2, %rax
setl    %al
movzblq %al, %eax
```

`%rax`	SF	CF	OF	ZF

Note: `setl` and `movzblq` do not modify condition codes

Exercise

`cmpq b,a` like computing `a-b` without setting dest

- **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

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)

```
xorq    %rax, %rax
subq    $1, %rax
cmpq    $2, %rax
setl    %al
movzblq %al, %eax
```

`%rax`	SF	CF	OF	ZF
`0x0000 0000 0000 0000`	0	0	0	1
`0xFFFF FFFF FFFF FFFF`	1	1	0	0
`0xFFFF FFFF FFFF FFFF`	1	0	0	0
`0xFFFF FFFF FFFF FF01`	1	0	0	0
`0x0000 0000 0000 0001`	1	0	0	0

Note: **`setl`** and **`movzblq`** do not modify condition codes

Machine-Level Programming II: Control

- Review of a few tricky bits from yesterday
- Basics of control flow
- Condition codes
- Conditional operations
- **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

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**

jne will always jump to .L2 as long as **ZF** is **not set / cleared (~ZF)** after each **shrq** Operation. Equivalent to **jnz**

```
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 or -O0`

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"/"guarded-do" 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/Guarded-Do 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

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/Guarded-Do Version

```
        testq  %rdi, %rdi
        je     .L4
        movl   $0, %eax
.L3:
        movq   %rdi, %rdx
        andl   $1, %edx
        addq   %rdx, %rax
        shrq   %rdi
        jne    .L3
        ret
.L4:
        movl   $0, %eax
        ret
```

- Initial conditional guards entrance to loop
  - Compiler can further use this to guard execution of potentially unnecessary initialization code

"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 To 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 To Jump-to-middle Conversion

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

Goto Version

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

"For" Loop To Do-While (Guarded-Do) Conversion

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 (compiler knows i = 0)

Goto Version

```
long pcount_for_gd_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;
}
```

Machine-Level Programming II: Control

- Review of a few tricky bits from yesterday
- Basics of control flow
- Condition codes
- Conditional operations
- Loops
- **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

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 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 **`x`**
`%rsi`	Argument **`y`**
`%rdx`	Argument **`z`**
`%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:

Indirect jump

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

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 $0 \le x \le 6$

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;
```

Compiler decides to place (w = 1) statement only under the cases that read w but not override w (here case 3 and 6 but not cases 1, 2, 4 and 6 and default)

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 # quad to octet (sign-extend
        # rax into 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**
`%rdx`	Argument **z**
`%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 **x**
`%rsi`	Argument **y**
`%rdx`	Argument **z**
`%rax`	Return value

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 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
```