Machine-Level Programming II: Control

Today

- **Review of a few tricky bits from last time**
- Basics of control flow
- Condition codes
- Conditional operations
- Loops
- If we have time: switch statements

Reminder: Machine Instructions

```
*dest = t;
```

- **C**
  - Store value **`t`** where designated by **`dest`**

```
movq %rax, (%rbx)
```

- **Assembly**
  - Move 8-byte value to memory
    - Quad words in x86-64 parlance
  - Operands:
    - **`t`**: Register **`%rax`**
    - **`dest`**: Register **`%rbx`**
    - **`*dest`**: Memory **`M[%rbx]`**

```
0x40059e:  48 89 03
```

- **Machine**
  - 3 bytes at address **`0x40059e`**
  - Compact representation of the assembly instruction
  - (Relatively) easy for hardware to interpret

Reminder: Address 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 (*why these numbers?*)

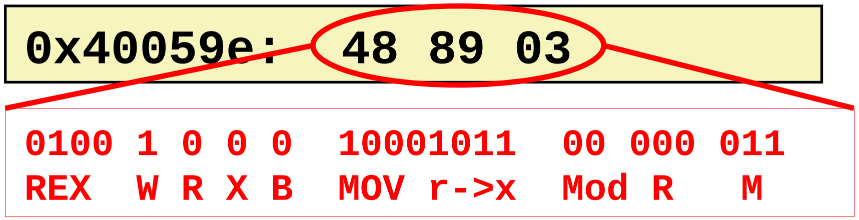
Special Cases

(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]]**

Reminder: Machine Instructions

```
*dest = t;
```

```
movq %rax, (%rbx)
```



```
0x40059e:  48 89 03
```

```
0100 1 0 0 0  10001011  00 000 011
REX  W R X B  MOV r->x  Mod R   M
```

- **C**
  - Store value **`t`** where designated by **`dest`**
- **Assembly**
  - Move 8-byte value to memory
    - Quad words in x86-64 parlance
  - Operands:
    - **`t`**: Register **`%rax`**
    - **`dest`**: Register **`%rbx`**
    - **`*dest`**: Memory **`M[%rbx]`**
- **Machine**
  - 3 bytes at address **`0x40059e`**
  - Compact representation of the assembly instruction
  - (Relatively) easy for hardware to interpret

Memory operands and LEA

- **In most instructions, a memory operand accesses memory**

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

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

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

Why use LEA?

- **CPU designers’ intended use: calculate a pointer to an object**
  - An array element, perhaps
  - For instance, to pass just one array element to another function

Assembly	C equivalent
lea (%rbx,%rdi,8), %rax	rax = &rbx[rdi]

- **Compiler authors like to use it for ordinary arithmetic**
  - It can do complex calculations in one instruction
  - It’s one of the only three-operand instructions the x86 has
  - It doesn’t touch the condition codes (we’ll come back to this)

Assembly	C equivalent
lea (%rbx,%rbx,2), %rax	rax = rbx * 3

Which numbers are pointers?

- They aren't labeled
- You have to figure it out from context

```
(gdb) info registers
rax      0x40057d          4195709
rbx      0x0               0
rcx      0x4005e0          4195808
rdx      0x7fffffffdc28    140737488346152
rsi      0x7fffffffdc18    140737488346136
rdi      0x1               1
rbp      0x0               0x0
rsp      0x7fffffffdb38    0x7fffffffdb38
r8       0x7ffff7dd5e80    140737351868032
r9       0x0               0
r10      0x7fffffffd7c0    140737488345024
r11      0x7ffff7a2f460    140737348039776
r12      0x400490          4195472
r13      0x7fffffffdc10    140737488346128
r14      0x0               0
r15      0x0               0
rip      0x40057d          0x40057d
```

Which numbers are pointers?

- They aren’t labeled
- You have to figure it out from context

- %rsp and %rip always hold pointers

```
(gdb) info registers
rax      0x40057d         4195709
rbx      0x0              0
rcx      0x4005e0         4195808
rdx      0x7fffffffdc28   140737488346152
rsi      0x7fffffffdc18   140737488346136
rdi      0x1              1
rbp      0x0              0x0
rsp      0x7fffffffdb38   0x7fffffffdb38
r8       0x7ffff7dd5e80   140737351868032
r9       0x0              0
r10      0x7fffffffd7c0   140737488345024
r11      0x7ffff7a2f460   140737348039776
r12      0x400490         4195472
r13      0x7fffffffdc10   140737488346128
r14      0x0              0
r15      0x0              0
rip      0x40057d         0x40057d
```

Which numbers are pointers?

- They aren’t labeled
- You have to figure it out from context
- %rsp and %rip always hold pointers
  - Register values that are “close” to %rsp or %rip are *probably* also pointers

```
(gdb) info registers
rax      0x40057d        4195709
rbx      0x0             0
rcx      0x4005e0        4195808
rdx      0x7fffffffdc28  140737488346152
rsi      0x7fffffffdc18  140737488346136
rdi      0x1             1
rbp      0x0             0x0
rsp      0x7fffffffdb38  0x7fffffffdb38
r8       0x7ffff7dd5e80  140737351868032
r9       0x0             0
r10      0x7fffffffd7c0  140737488345024
r11      0x7ffff7a2f460  140737348039776
r12      0x400490        4195472
r13      0x7fffffffdc10  140737488346128
r14      0x0             0
r15      0x0             0
rip      0x40057d        0x40057d
```

Which numbers are pointers?

- **If a register is being *used* as a pointer…**

```
Dump of assembler code for function main:
=> 0x40057d <+0>:  sub   $0x8,%rsp
   0x400581 <+4>:  mov   (%rsi),%rsi
   0x400584 <+7>:  mov   $0x400670,%edi
   0x400589 <+12>: mov   $0x0,%eax
   0x40058e <+17>: call  0x400460
```

Which numbers are pointers?

- **If a register is being *used* as a pointer…**
  - mov (%rsi), %rsi
  - …Then its value is *expected* to be a pointer.
    - There might be a bug that makes its value incorrect.

```
Dump of assembler code for function main:
=> 0x40057d <+0>:  sub   $0x8,%rsp
   0x400581 <+4>:  mov   (%rsi),%rsi
   0x400584 <+7>:  mov   $0x400670,%edi
   0x400589 <+12>: mov   $0x0,%eax
   0x40058e <+17>: call  0x400460
```

Which numbers are pointers?

- **If a register is being *used* as a pointer…**
  - mov (%rsi), %rsi
  - …Then its value is *expected* to be a pointer.
    - There might be a bug that makes its value incorrect.

```
Dump of assembler code for function main:
=> 0x40057d <+0>:   sub    $0x8,%rsp
   0x400581 <+4>:   mov    (%rsi),%rsi
   0x400584 <+7>:   mov    $0x400670,%edi
   0x400589 <+12>:  mov    $0x0,%eax
   0x40058e <+17>:  call   0x400460
```

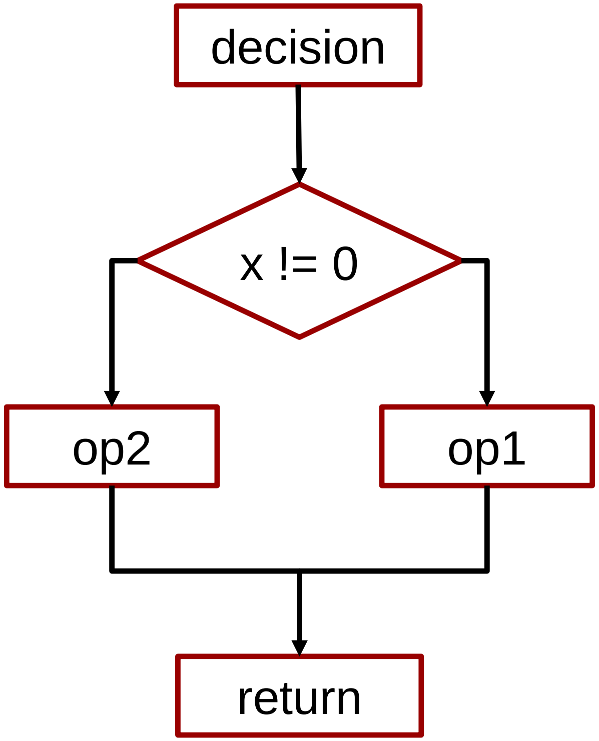
- **Not as obvious with complicated address “modes”**
  - (%rsi, %rbx) – *One* of these is a pointer, we don’t know which.
  - (%rsi, %rbx, 2) – %rsi is a pointer, %rbx isn’t (why?)
  - 0x400570(, %rbx, 2) – 0x400570 is a pointer, %rbx isn’t (why?)
  - lea (anything), %rax – (anything) *may or may not* be a pointer

Today

- Review of a few tricky bits from yesterday
- Basics of control flow
- Condition codes
- Conditional operations
- Loops
- If we have time: 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

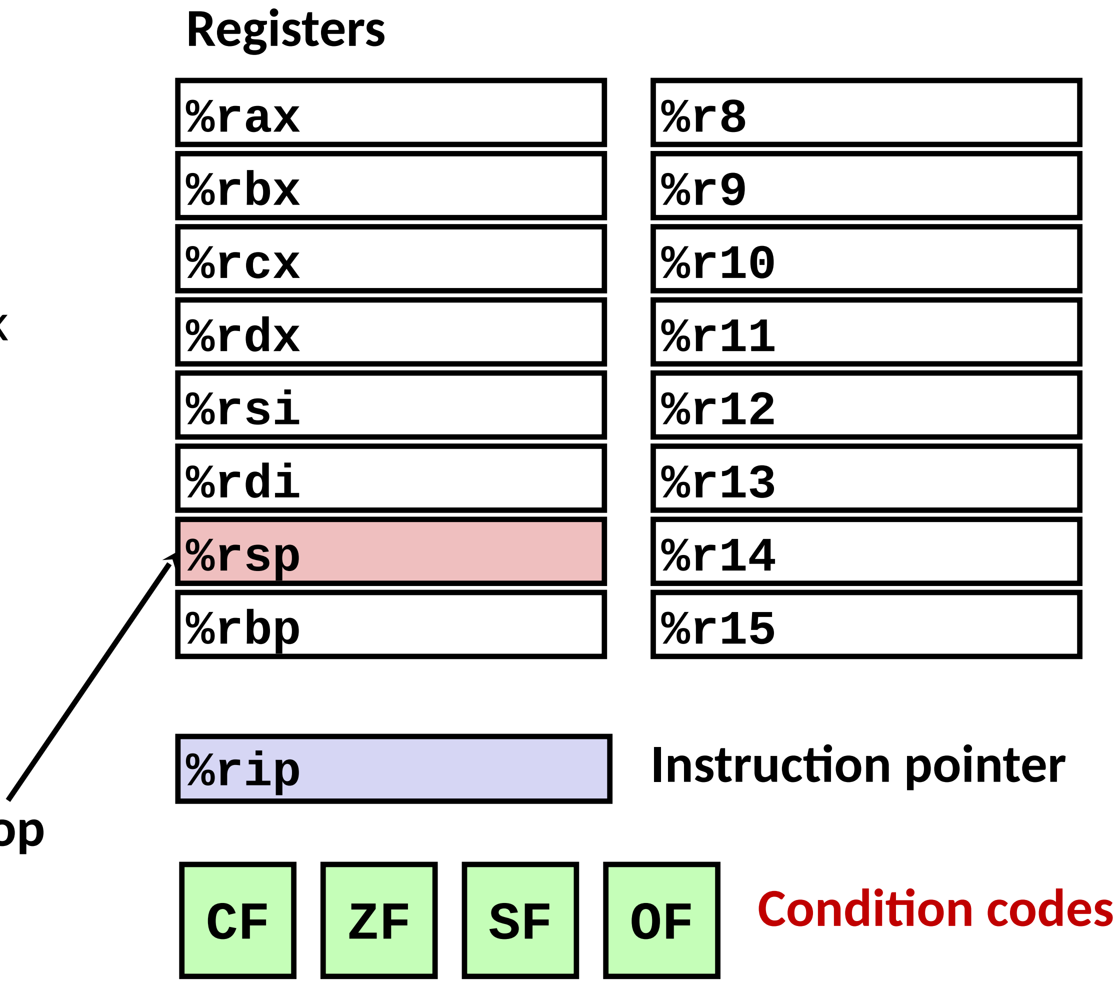
```
extern void op1(void);
extern void op2(void);

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 ]` *Z set, CSO clear*

- **Implicitly set (as side effect) of 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)`

- **Not set by `leaq` instruction**

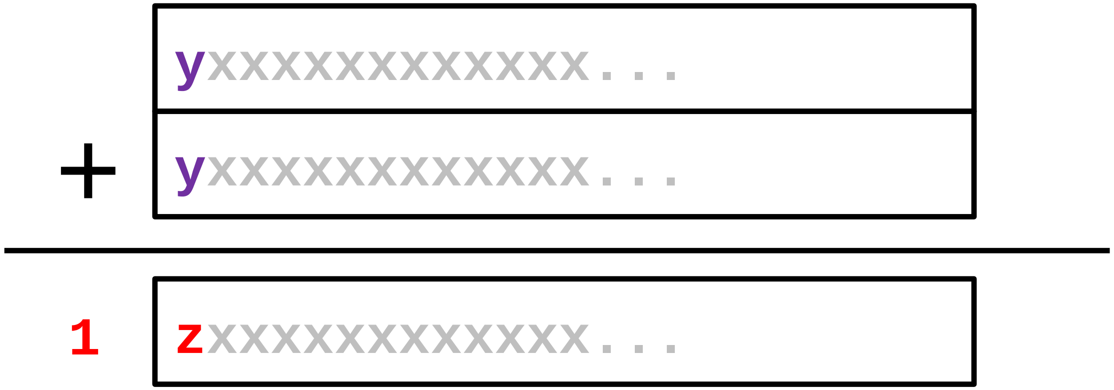
ZF set when

```
000000000000…00000000000
```

SF set when

```
1xxxxxxxxxxx…xxxxxxxxxxx
```

CF set when

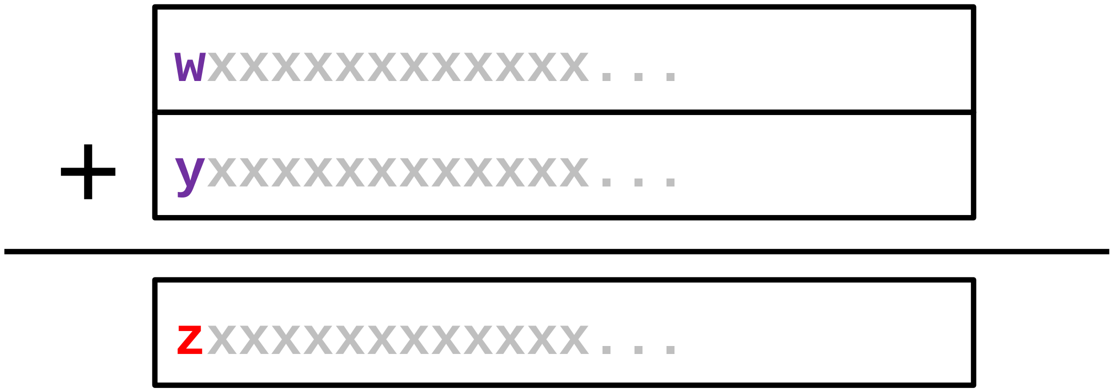


yxxxxxxxxxxxx...

+ yxxxxxxxxxxxx...

1 zxxxxxxxxxxxx...

OF set when



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

w == y && w != z

Compare Instruction

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

  - Used for **`if (a < b) { … }`**
    whenever isn’t needed for anything else

Test Instruction

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

  - 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)

Today

- Review of a few tricky bits from yesterday
- Basics of control flow
- Condition codes
- **Conditional operations**
- Loops
- If we have time: switch statements

Jumping

jX Instructions

- Jump to different part of code 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)

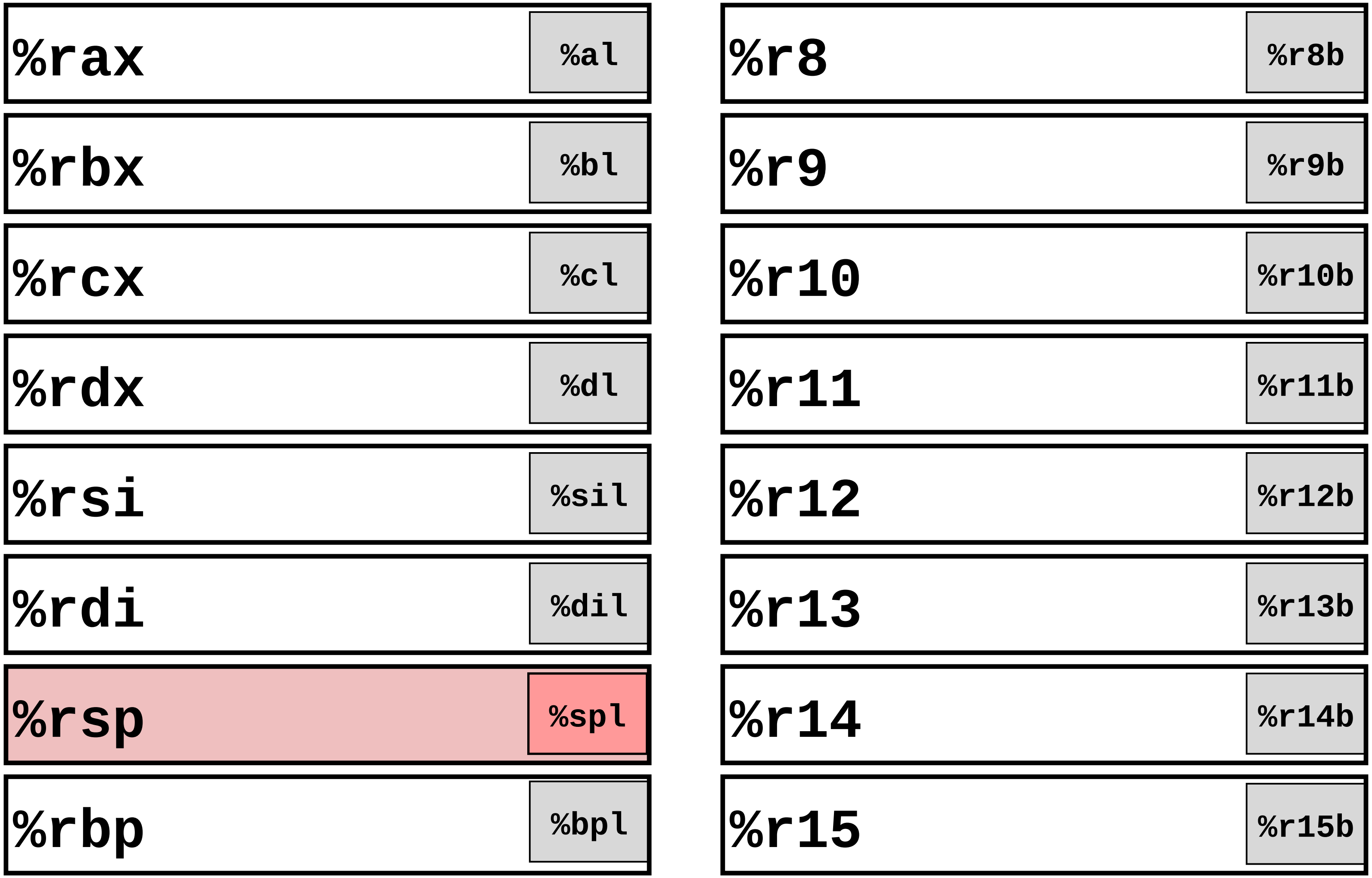
Reading Condition Codes

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

%rbx %bl

%rcx %cl

%rdx %dl

%rsi %sil

%rdi %dil

%rsp %spl

%rbp %bpl

%r8 %r8b

%r9 %r9b

%r10 %r10b

%r11 %r11b

%r12 %r12b

%r13 %r13b

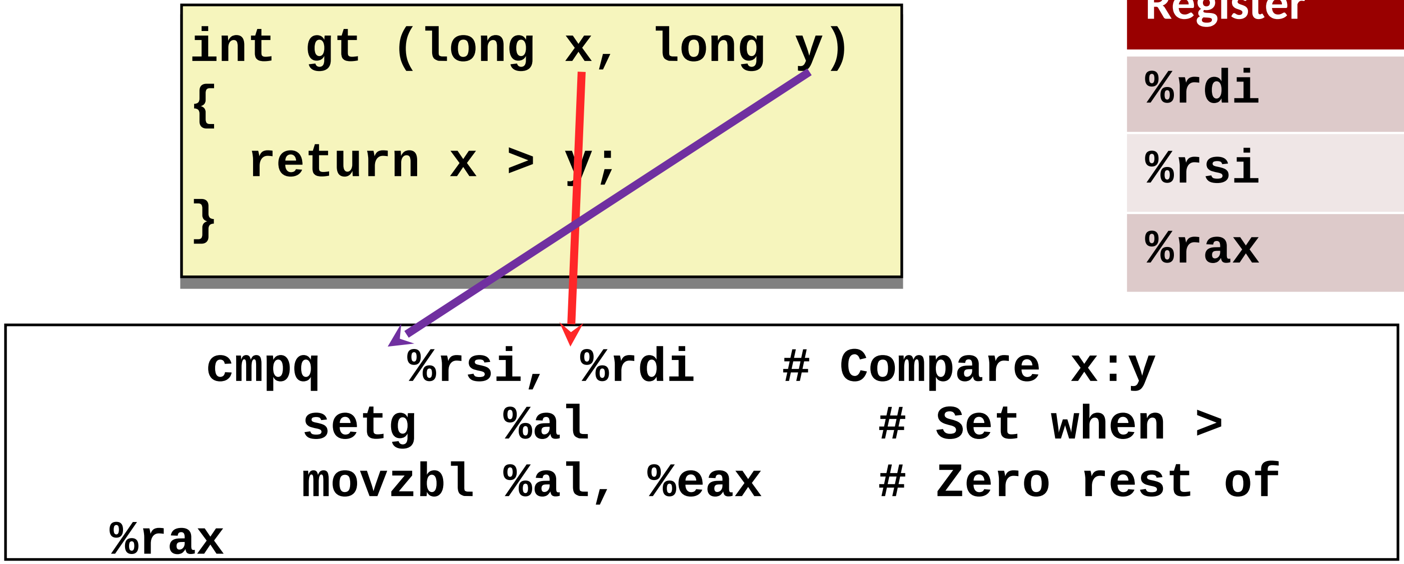
%r14 %r14b

%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;
}
```

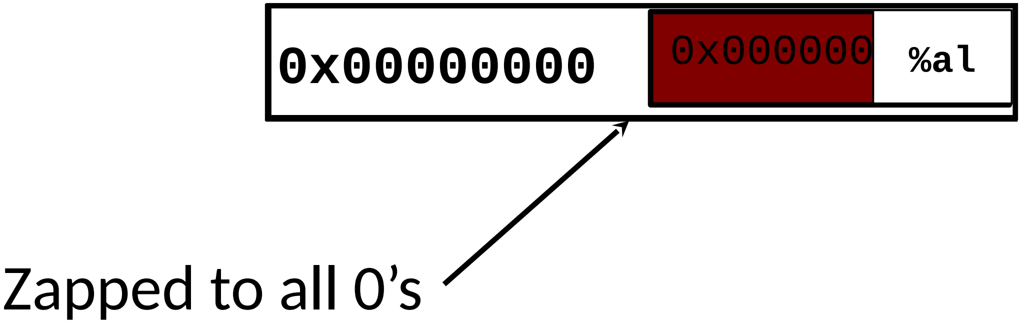
```
       cmpq   %rsi, %rdi   # Compare x:y
           setg   %al          # Set when >
           movzbl %al, %eax    # Zero rest of
   %rax
       ret
```

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

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
```

Conditional Branch Example (Old Style)

- **Generation**

```
shark> gcc –Og -S –fno-if-conversion cont
```

I’ll get to this shortly.

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

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

Register	Use(s)
`%rdi`	Argument **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
```

Bad Cases for Conditional Move

Expensive Computations

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

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

Bad Performance

Risky Computations

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

- Both values get computed
- May have undesirable effects

Unsafe

Computations with side effects

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

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

Illegal

Today

- Review of a few tricky bits from yesterday
- Basics of control flow
- Condition codes
- Conditional operations
- **Loops**
- If we have time: switch statements

“Do-While” Loop Example

C Code

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

Goto Version

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

- Count number of 1’s in argument x (“popcount”)
- Use conditional branch to either continue looping or to exit loop

“Do-While” Loop Compilation

Goto Version

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

Register	Use(s)
`%rdi`	Argument `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$_2$;

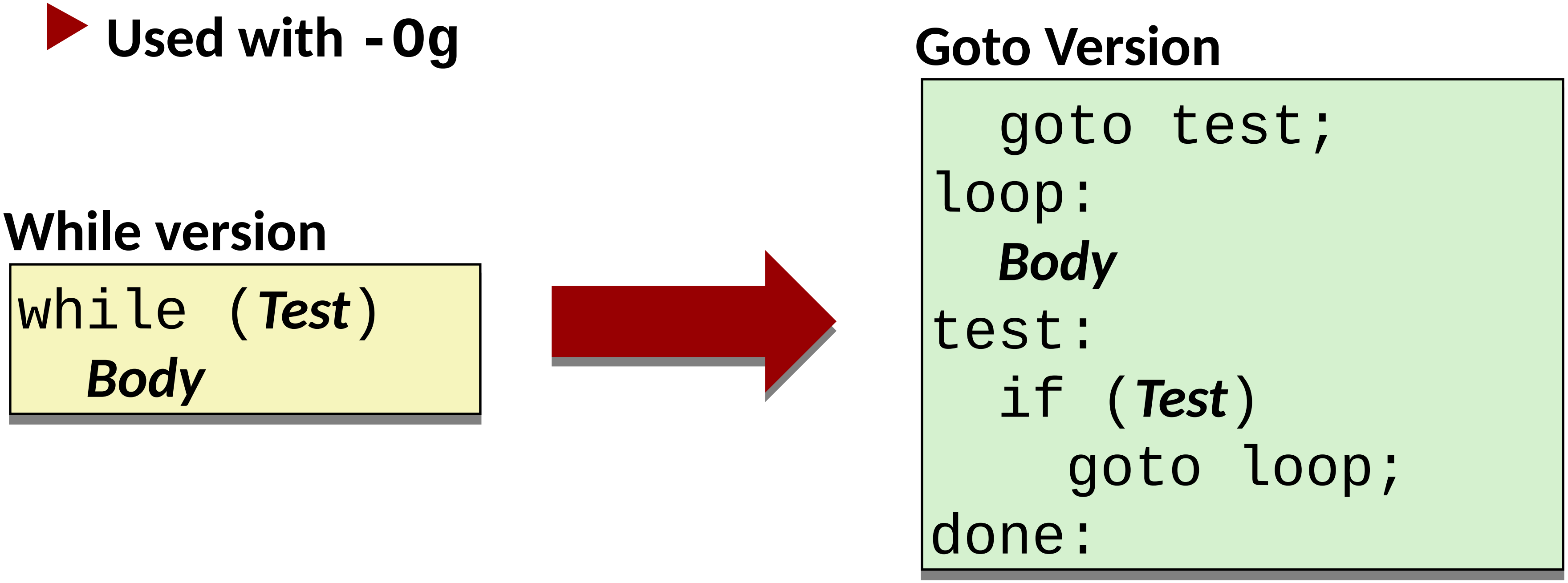
  …

  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;
}
```

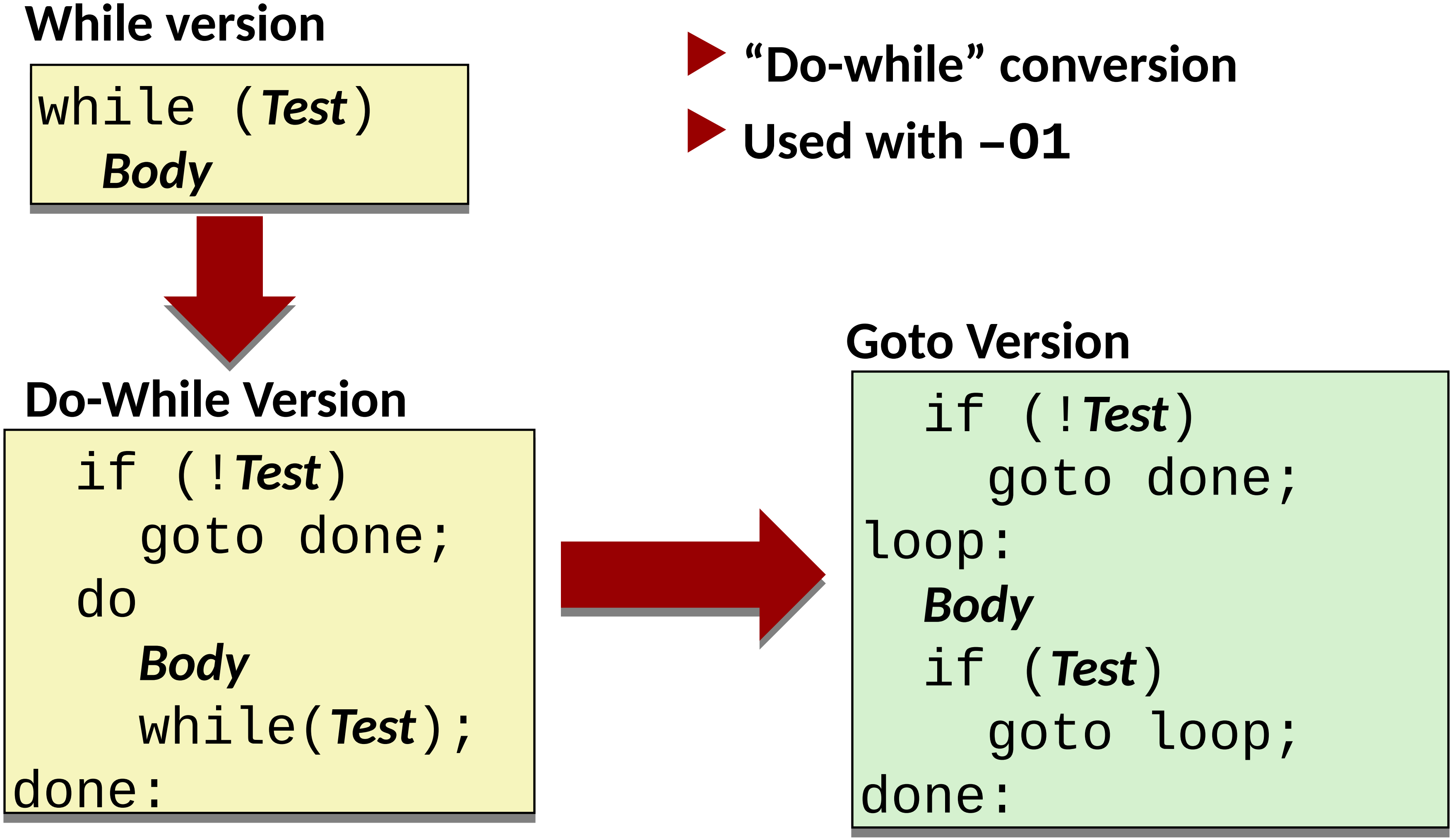
Jump to Middle

```
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

- “Do-while” conversion
- Used with –O1



While version

```
while (Test)
   Body
```

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;
}
```

- Compare to do-while version of function
- Initial conditional guards entrance to loop

“For” Loop Form

General Form

```
for (Init; Test; Update )

        Body
```

```
#define WSIZE 8*sizeof(int)
long pcount_for
  (unsigned long x)
{
  size_t i;
  long result = 0;
  for (i = 0; i < WSIZE; i++)
  {
    unsigned bit =
      (x >> i) & 0x1;
    result += bit;
  }
  return result;
}
```

Init

```
i = 0
```

Test

```
i < WSIZE
```

Update

```
i++
```

Body

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

“For” Loop → While Loop

For Version

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

While Version

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

For-While Conversion

Init

```
i = 0
```

Test

```
i < WSIZE
```

Update

```
i++
```

Body

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

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

“For” Loop Do-While Conversion

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

Today

- Review of a few tricky bits from yesterday
- Basics of control flow
- Condition codes
- Conditional operations
- Loops
- **If we have time: switch statements**

```
long switch_eg
   (long x, long y, long z)
{
    long w = 1;
    switch(x) {
    case 1:
        w = y*z;
        break;
    case 2:
        w = y/z;
        /* Fall Through */
    case 3:
        w += z;
        break;
    case 5:
    case 6:
        w -= z;
        break;
    default:
        w = 2;
    }
    return w;
}
```

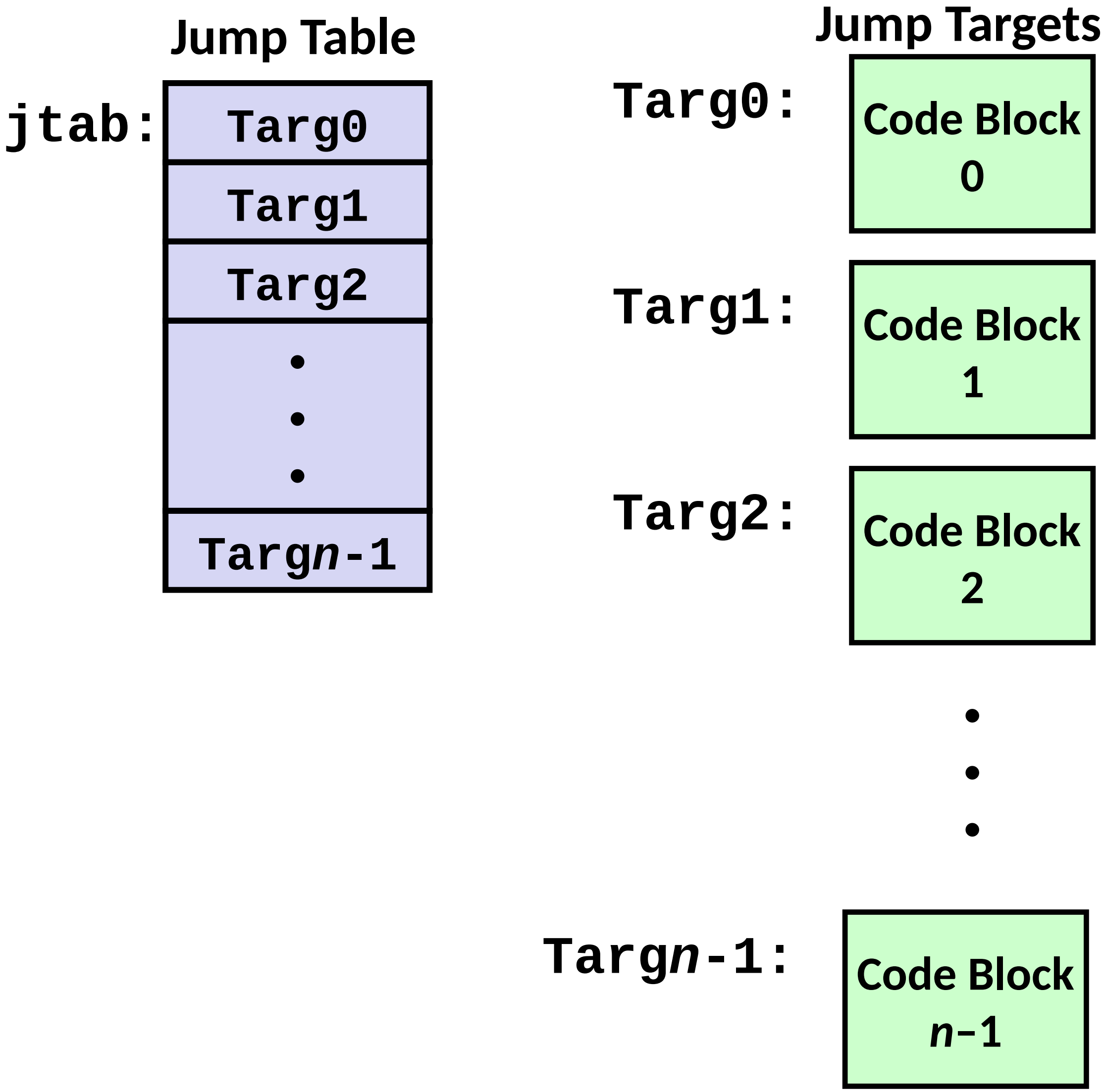
Switch Statement Example

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

Jump Table Structure

Switch Form

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



Jump Table

```
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)
```

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

Jump table

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

Setup:

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

Indirect jump

Assembly Setup Explanation

- **Table Structure**
  - Each target requires 8 bytes
  - Base address at **`.L4`**

- **Jumping**
  - **Direct:** **`jmp .L8`**
  - Jump target is denoted by label **`.L8`**

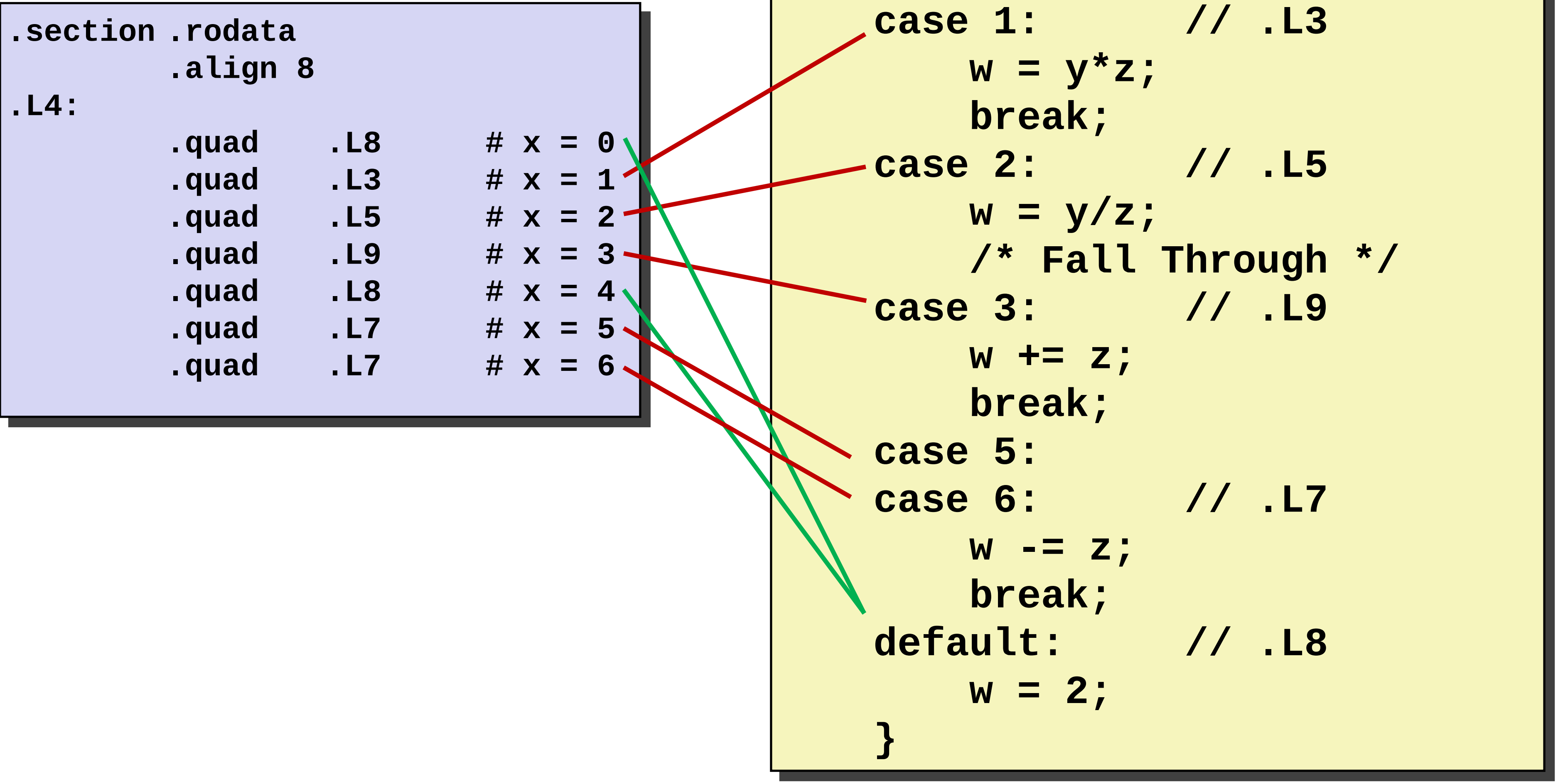
  - **Indirect:** **`jmp *.L4(,%rdi,8)`**
  - Start of jump table: **`.L4`**
  - Must scale by factor of 8 (addresses are 8 bytes)
  - Fetch target from effective Address **`.L4 + x*8`**
    - Only for 0 ≤ **x** ≤ 6

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;
}
```

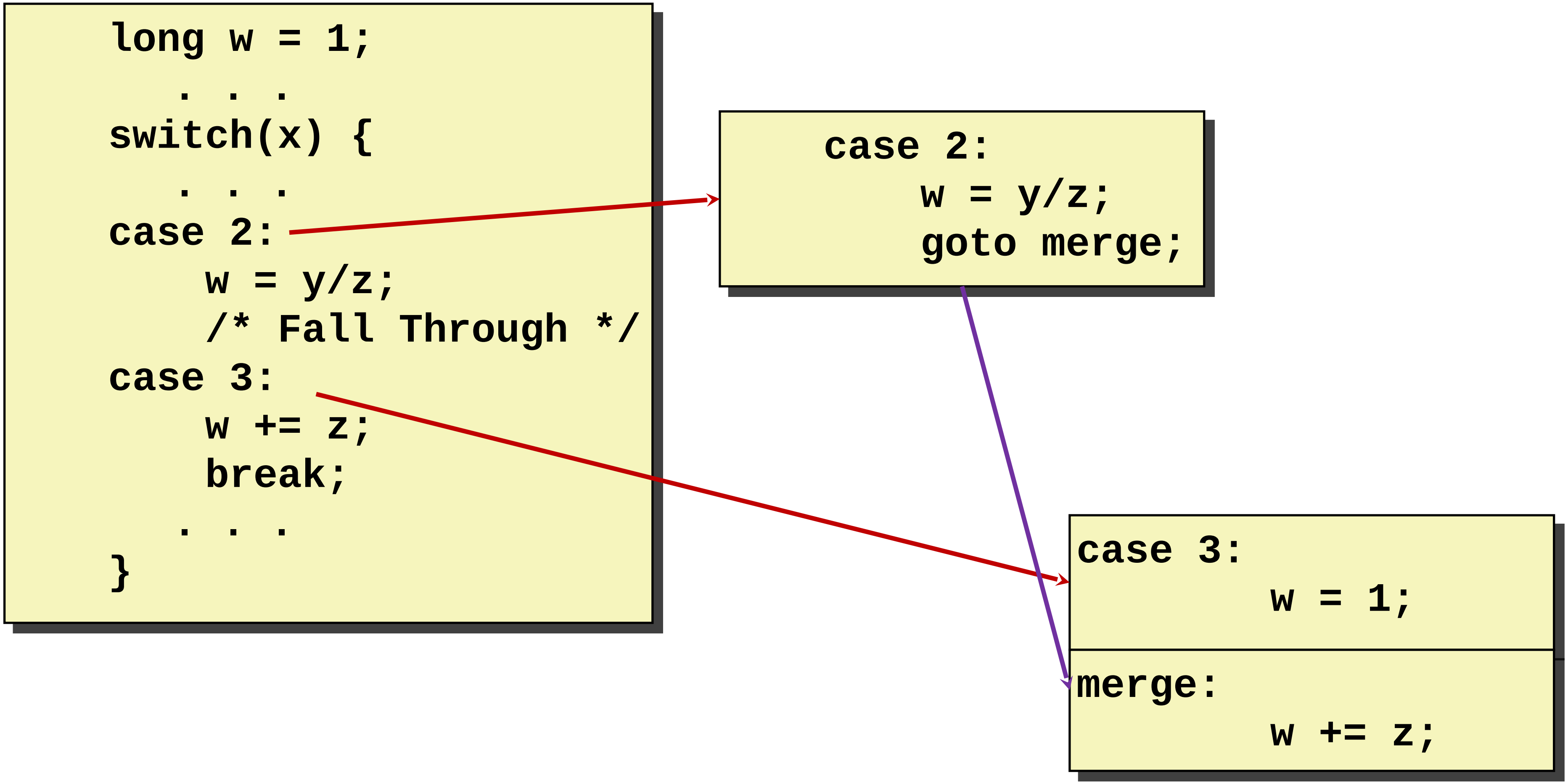
Code Blocks (x == 1)

```
  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
   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

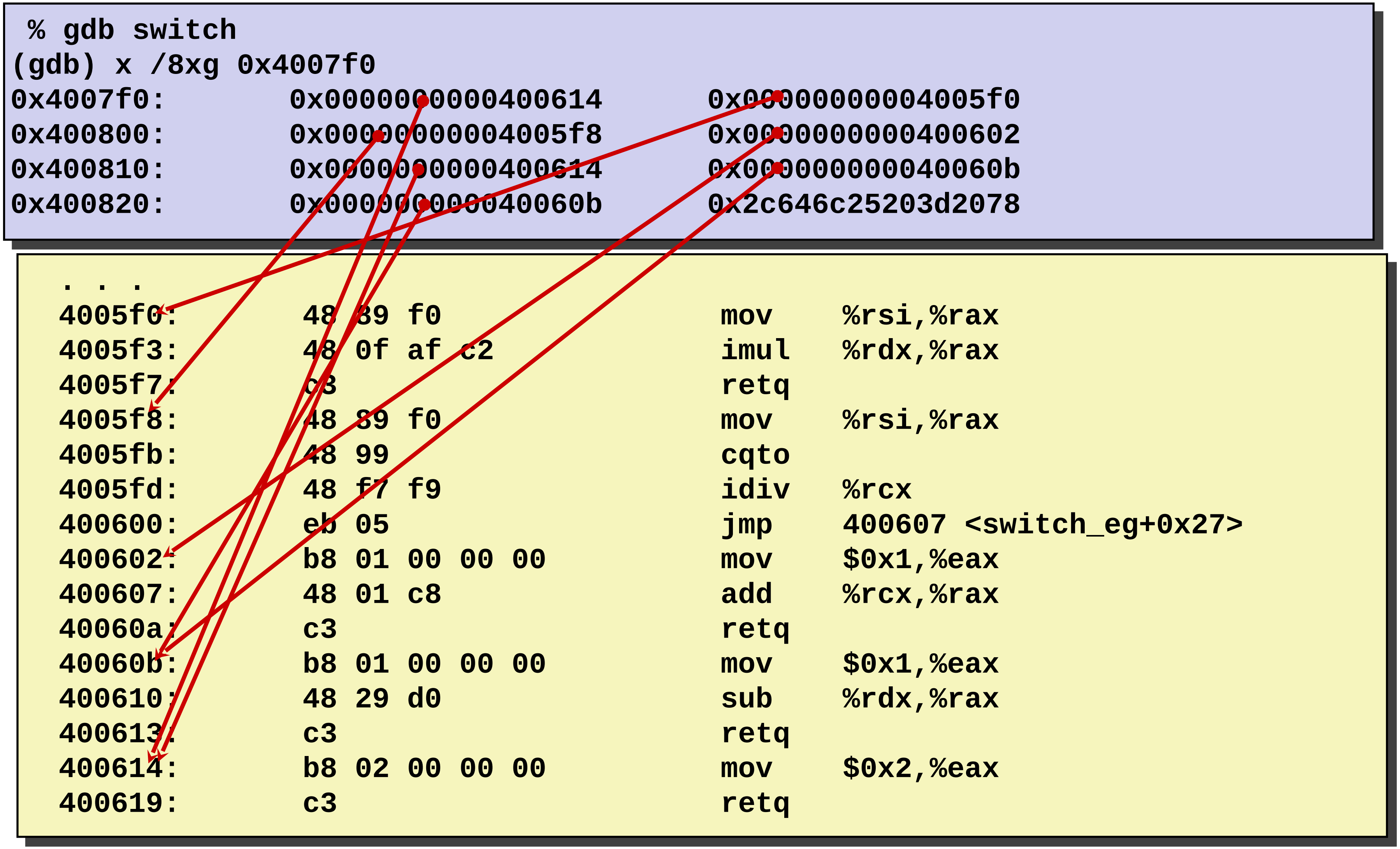
```
 00000000004005e0 <switch_eg>:
4005e0:       48 89 d1                mov    %rdx,%rcx
4005e3:       48 83 ff 06             cmp    $0x6,%rdi
4005e7:       77 2b                   ja     400614 <switch_eg+0x34>
4005e9:       ff 24 fd f0 07 40 00    jmpq   *0x4007f0(,%rdi,8)
4005f0:       48 89 f0                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.)

```
   0000000000004005e0 <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
```