x86-64 Programming III

CSE 351 Winter 2019

Instructors:

Max Willsey

Luis Ceze

Teaching Assistants:

Britt Henderson

Lukas Joswiak

Josie Lee

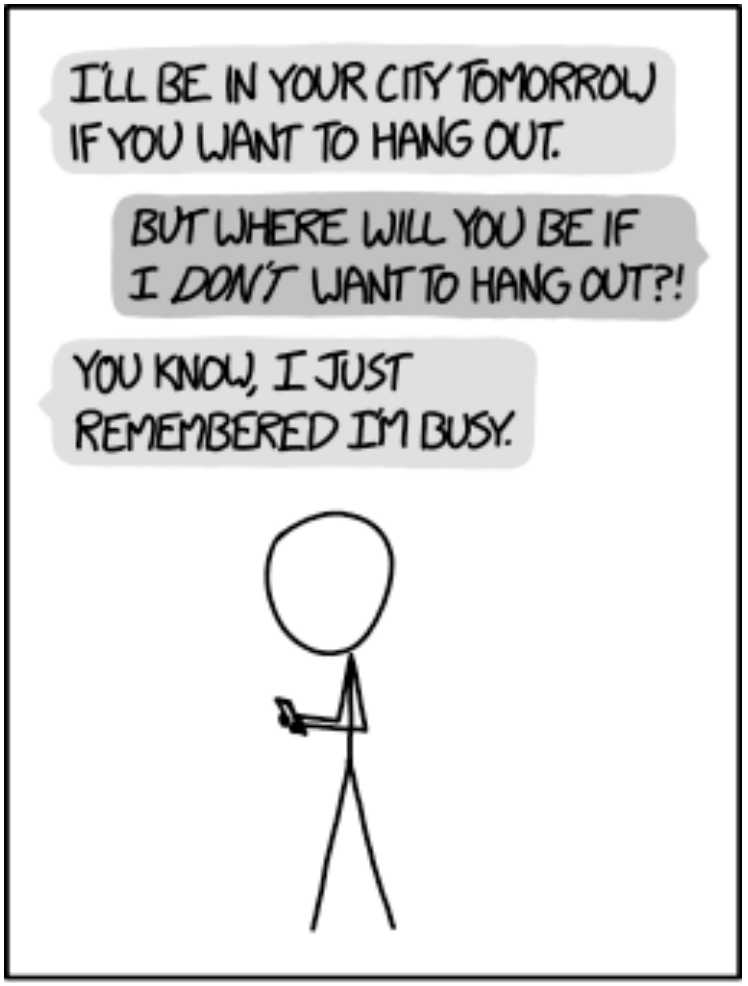
Wei Lin

Daniel Snitkovsky

Luis Vega

Kory Watson

Ivy Yu



WHY I TRY NOT TO BE PEDANTIC ABOUT CONDITIONALS.

http://xkcd.com/1652/

Administrivia

- Homework 2 due Friday (10/19)
- Lab 2 due next Friday (10/26)

- Section tomorrow on Assembly and GDB
  - Bring your laptops!

- Midterm: Feb 13, 8:30am KNE 130 (here ☺)
  - You will be provided a fresh reference sheet
  - You get 1 *handwritten*, double-sided cheat sheet (letter)

Conditionals and Control Flow

- Conditional branch/*jump*
  - Jump to somewhere else if some *condition* is true, otherwise execute next instruction
- Unconditional branch/*jump*
  - *Always* jump when you get to this instruction

- Together, they can implement most control flow constructs in high-level languages:
  - **`if`** (*condition*) **`then`** {…} **`else`** {…}
  - **`while`** (*condition*) {…}
  - **`do`** {…} **`while`** (*condition*)
  - **`for`** (*initialization*; *condition*; *iterative*) {…}
  - **`switch`** {…}

Condition Codes (Implicit Setting)

- *Implicitly* set by **arithmetic** operations
  - (think of it as side effects)
  - Example: **addq** `src, dst` ↔ `r = d+s`

    result = dst + src

  - **CF=1** if carry out from MSB (*unsigned* overflow)
  - **ZF=1** if `r==0`
  - **SF=1** if `r<0` (if MSB is 1)
  - **OF=1** if *signed* overflow

    `(s>0 && d>0 && r<0)||(s<0 && d<0 && r>=0)`

    ↑ signs don't match!
  - *Not* set by `lea` instruction (beware!)

example if %eax holds 0x80 00 00 00:

addl %eax, %eax

0x0 stored in %eax
CF = 1
ZF = 1
SF = 0
OF = 1 (⊖ + ⊖ = ⊕)

CF	*Carry Flag*	ZF	*Zero Flag*	SF	*Sign Flag*	OF	*Overflow Flag*

Condition Codes (Explicit Setting: Compare)

- *Explicitly* set by **Compare** instruction
  - **`cmpq`** `src1, src2`
  - **`cmpq`** `a, b` sets flags based on `b-a`, but doesn't store

  - **`CF=1`** if carry out from MSB (good for *unsigned* comparison)
  - **`ZF=1`** if `a==b`
  - **`SF=1`** if `(b-a)<0` (if MSB is 1)
  - **`OF=1`** if *signed* overflow

```
(a>0  && b<0  && (b-a)>0)  ||
(a<0  && b>0  && (b-a)<0)
```

CF	*Carry Flag*	ZF	*Zero Flag*	SF	*Sign Flag*	OF	*Overflow Flag*

Condition Codes (Explicit Setting: Test)

- *Explicitly* set by **Test** instruction
  - **`testq`** `src2, src1`
  - **`testq`** `a, b` sets flags based on `a&b`, but doesn't store
    - Useful to have one of the operands be a ***mask***
  - Can't have carry out (**CF**) or overflow (**OF**)
  - **ZF=1** if `a&b==0`
  - **SF=1** if `a&b<0` (signed)

CF	*Carry Flag*	ZF	*Zero Flag*	SF	*Sign Flag*	OF	*Overflow Flag*

Using Condition Codes: Jumping

- `j*` Instructions
  - Jumps to ***target*** (an address) based on condition codes

Instruction	Condition	Description
`jmp` *target*	`1`	Unconditional
`je` *target*	`ZF`	Equal / Zero
`jne` *target*	`~ZF`	Not Equal / Not Zero
`js` *target*	`SF`	Negative
`jns` *target*	`~SF`	Nonnegative
`jg` *target*	`~(SF^OF)&~ZF`	Greater (Signed)
`jge` *target*	`~(SF^OF)`	Greater or Equal (Signed)
`jl` *target*	`(SF^OF)`	Less (Signed)
`jle` *target*	`(SF^OF)|ZF`	Less or Equal (Signed)
`ja` *target*	`~CF&~ZF`	Above (unsigned “>”)
`jb` *target*	`CF`	Below (unsigned “<“)

Using Condition Codes: Setting

- `set*` Instructions
  - Set low-order byte of `dst` to 0 or 1 based on condition codes
  - Does not alter remaining 7 bytes

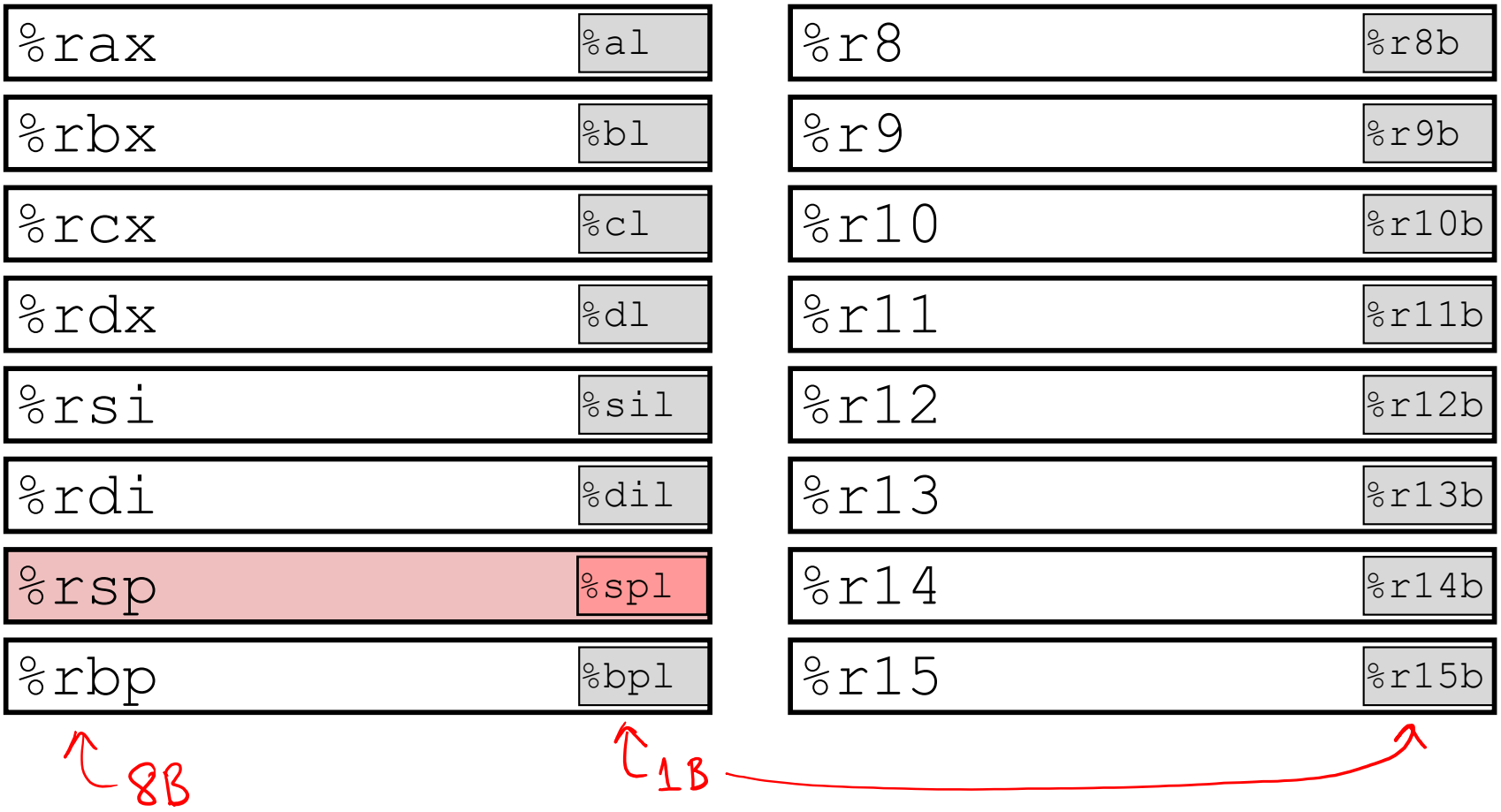
False → 0b 0000 0000 = 0x 00
True → 0b 0000 0001 = 0x 01

same instruction suffixes as j* instructions!

Instruction	Condition	Description
sete *dst*	ZF	Equal / Zero
setne *dst*	~ZF	Not Equal / Not Zero
sets *dst*	SF	Negative
setns *dst*	~SF	Nonnegative
setg *dst*	~(SF^OF)&~ZF	Greater (Signed)
setge *dst*	~(SF^OF)	Greater or Equal (Signed)
setl *dst*	(SF^OF)	Less (Signed)
setle *dst*	(SF^OF)|ZF	Less or Equal (Signed)
seta *dst*	~CF&~ZF	Above (unsigned “>”)
setb *dst*	CF	Below (unsigned “<”)

Reminder: x86-64 Integer Registers

- Accessing the low-order byte:



Reading Condition Codes

Register	Use(s)
%rdi	1st argument (x)
%rsi	2nd argument (y)
%rax	return value

- set* Instructions
  - Set a low-order byte to 0 or 1 based on condition codes
  - Operand is byte register (e.g. al, dl) or a byte in memory
  - Do not alter remaining bytes in register
    - Typically use movzbl (zero-extended mov) to finish job

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

```
cmpq    %rsi, %rdi    #
setg    %al           #
movzbl  %al, %eax     #
ret
```

Reading Condition Codes

Register	Use(s)
%rdi	1st argument (x)
%rsi	2nd argument (y)
%rax	return value

- `set*` Instructions (e, ne, g, l, ...)
  - Set a low-order byte to 0 or 1 based on condition codes
  - Operand is byte register (e.g. `al`, `dl`) or a byte in memory
  - Do not alter remaining bytes in register
    - Typically use `movzbl` (zero-extended `mov`) to finish job

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

// x-y > 0

```
cmpq    %rsi, %rdi      # set flags based on x-y
setg    %al             # %al = (x>y)
movzbl  %al, %eax       # %rax = (x>y)
ret
```

a(y), b(x)

zero-extend

lowest byte

whole register

Aside: `movz` and `movs`

`movz__` src, *reg*Dest # Move with zero extension

`movs__` src, *reg*Dest # Move with sign extension

- Copy from a *smaller* source value to a *larger* destination
- Source can be memory or register; Destination *must* be a register
- Fill remaining bits of dest with **zero** (`movz`) or **sign bit** (`movs`)

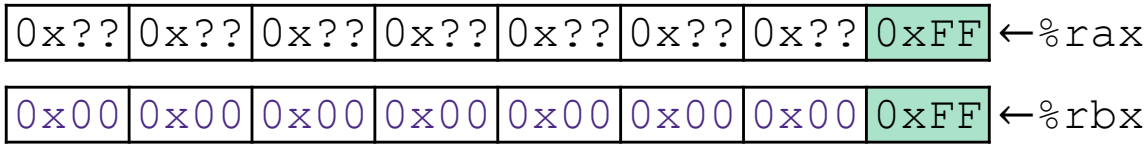
`movz`*SD* **/** **`movs`*SD*:**

S – size of source (**`b`** = 1 byte, **`w`** = 2)

D – size of dest (**`w`** = 2 bytes, **`l`** = 4, **`q`** = 8)

Example:

```
movzbq %al, %rbx
```



Aside: `movz` and `movs`

`movz__` src, *reg*Dest # Move with zero extension

`movs__` src, *reg*Dest # Move with sign extension

- Copy from a *smaller* source value to a *larger* destination
- Source can be memory or register; Destination *must* be a register
- Fill remaining bits of dest with **zero** (`movz`) or **sign bit** (`movs`)

`movz`*SD* / **`movs`*SD***:

S – size of source (**`b`** = 1 byte, **`w`** = 2)

D – size of dest (**`w`** = 2 bytes, **`l`** = 4, **`q`** = 8)

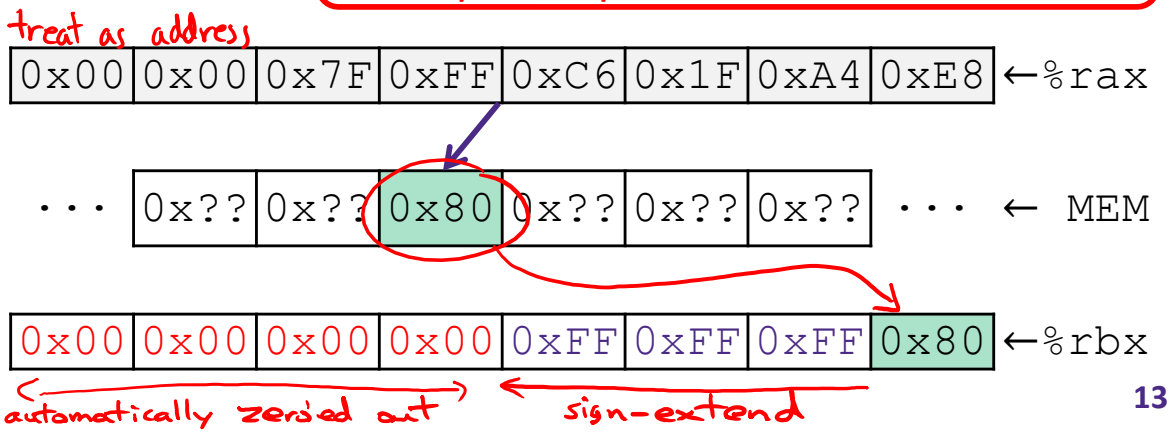
Note: In x86-64, *any instruction* that generates a 32-bit (long word) value for a register also sets the high-order portion of the register to 0. Good example on p. 184 in the textbook.

Example:

```
movsbl (%rax), %ebx
```

sign-extend; 1 byte; 4 bytes

Copy 1 byte from memory into 8-byte register & sign extend it



GDB Demo

- The `movz` and `movs` examples on a real machine!
  - `movzbq %al, %rbx`
  - `movsbl (%rax), %ebx`

- You will need to use GDB to get through Lab 2
  - Useful debugger in this class and beyond!

- Pay attention to:
  - Setting breakpoints (`break`)
  - Stepping through code (`step`/`next` and `stepi`/`nexti`)
  - Printing out expressions (`print` – works with regs & vars)
  - Examining memory (`x`)

Choosing instructions for conditionals

- All arithmetic instructions set condition flags based on result of operation (`op`)
  - Conditionals are comparisons against 0
- Come in instruction *pairs*

```
      (4) addq 5, (p)
  je:     *p+5 == 0
(2) jne:  *p+5 != 0
  jg:     *p+5 >  0
  jl:     *p+5 <  0
```

```
      (1) orq a, b
  je:     b|a == 0
  jne:    b|a != 0
(2) jg:   b|a >  0
  jl:     b|a <  0
```

		(1) (op) s, d
je	"Equal"	d (op) s == 0
jne	"Not equal"	d (op) s != 0
js	"Sign" (negative)	d (op) s < 0
jns	(non-negative)	d (op) s >= 0
jg	"Greater"	d (op) s > 0
jge	"Greater or equal"	d (op) s >= 0
(2) **jl**	"Less"	d (op) s < 0
jle	"Less or equal"	d (op) s <= 0
ja	"Above" (unsigned >)	d (op) s > 0U
jb	"Below" (unsigned <)	d (op) s < 0U

Choosing instructions for conditionals

- Reminder: `cmp` is like `sub`, `test` is like `and`
  - Result is not stored anywhere

		`cmp a,b`	`test a,b`
je	"Equal"	`b == a`	`b&a == 0`
jne	"Not equal"	`b != a`	`b&a != 0`
js	"Sign" (negative)	`b-a < 0`	`b&a < 0`
jns	(non-negative)	`b-a >=0`	`b&a >= 0`
jg	"Greater"	`b > a`	`b&a > 0`
jge	"Greater or equal"	`b >= a`	`b&a >= 0`
jl	"Less"	`b < a`	`b&a < 0`
jle	"Less or equal"	`b <= a`	`b&a <= 0`
ja	"Above" (unsigned >)	`b-a > 0U`	`b&a > 0U`
jb	"Below" (unsigned <)	`b-a < 0U`	`b&a < 0U`

```
    cmpq 5, (p)
je:  *p == 5
jne: *p != 5
jg:  *p >  5
jl:  *p <  5
```

```
    testq a, a
je:   a == 0
jne:  a != 0
jg:   a >  0
jl:   a <  0
```

```
    testb a, 0x1
je:   a_LSB == 0
jne:  a_LSB == 1
```

Choosing instructions for conditionals

		cmp a,b	test a,b
je	"Equal"	b == a	b&a == 0
jne	"Not equal"	b != a	b&a != 0
js	"Sign" (negative)	b-a < 0	b&a < 0
jns	(non-negative)	b-a >=0	b&a >= 0
jg	"Greater"	b > a	b&a > 0
jge	"Greater or equal"	b >= a	b&a >= 0
jl	"Less"	b < a	b&a < 0
jle	"Less or equal"	b <= a	b&a <= 0
ja	"Above" (unsigned >)	b > a	b&a > 0U
jb	"Below" (unsigned <)	b < a	b&a < 0U

Register	Use(s)
%rdi	argument x
%rsi	argument y
%rax	return value

```
if (x < 3) {
  return 1;
}
return 2;
```

do this if x ≥ 3

```
  cmpq $3, %rdi
  jge T2
T1: # x < 3:  (if)
  movq $1, %rax
  ret
T2: # !(x < 3): (else)
  movq $2, %rax
  ret
```

labels

Question

Register	Use(s)
%rdi	1st argument (x)
%rsi	2nd argument (y)
%rax	return value

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

A.
```
cmpq  %rsi, %rdi
jle   .L4
```

B.
```
cmpq  %rsi, %rdi
jg    .L4
```

C.
```
testq %rsi, %rdi
jle   .L4
```

D.
```
testq %rsi, %rdi
jg    .L4
```

E. We're lost…

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

Choosing instructions for conditionals

		cmp a,b	test a,b
je	"Equal"	b == a	b&a == 0
jne	"Not equal"	b != a	b&a != 0
js	"Sign" (negative)	b-a < 0	b&a < 0
jns	(non-negative)	b-a >=0	b&a >= 0
jg	"Greater"	b > a	b&a > 0
jge	"Greater or equal"	b >= a	b&a >= 0
jl	"Less"	b < a	b&a < 0
jle	"Less or equal"	b <= a	b&a <= 0
ja	"Above" (unsigned >)	b > a	b&a > 0U
jb	"Below" (unsigned <)	b < a	b&a < 0U

- https://godbolt.org/z/j72AEn

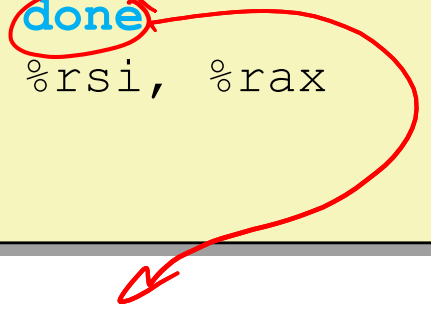
```
if (x < 3 && x == y) {
  return 1;
} else {
  return 2;
}
```

```
  cmpq $3, %rdi
  setl %al
  cmpq %rsi, %rdi
  sete %bl
  testb %al, %bl
  je T2
T1: # x < 3 && x == y:
  movq $1, %rax
  ret
T2: # else
  movq $2, %rax
  ret
```

Labels

```
swap:
    movq   (%rdi), %rax
    movq   (%rsi), %rdx
    movq   %rdx, (%rdi)
    movq   %rax, (%rsi)
    ret
```

```
max:
    movq %rdi, %rax
    cmpq %rsi, %rdi
    jg   done
    movq %rsi, %rax
done:
    ret
```



- A jump changes the program counter (`%rip`)
  - `%rip` tells the CPU the *address* of the next instr to execute
- **Labels** give us a way to refer to a specific instruction in our assembly/machine code
  - Associated with the *next* instruction found in the assembly code (ignores whitespace)
  - Each ***use*** of the label will eventually be replaced with something that indicates the final address of the instruction that it is associated with

x86 Control Flow

- Condition codes
- Conditional and unconditional branches
- **Loops**
- Switches

Expressing with Goto Code

```
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;
}
```

conditional jump → cmp, jle

unconditional jump → jmp

labels (addresses)

- C allows `goto` as means of transferring control (`jump`)
  - Closer to assembly programming style
  - Generally considered bad coding style

Compiling Loops

C/Java code:

```
while ( sum != 0 ) {
    <loop body>
}
```

Test

Assembly code:

```
loopTop:    testq %rax, %rax
            je    loopDone
            <loop body code>
            jmp   loopTop
loopDone:
```

!Test

- Other loops compiled similarly
  - Will show variations and complications in coming slides, but may skip a few examples in the interest of time
- Most important to consider:
  - When should conditionals be evaluated? (*while* vs. *do-while*)
  - How much jumping is involved?

Compiling Loops

C/Java code:

```
while ( Test ) {
    Body
}
```

Goto version

```
Loop: if ( !Test )  goto Exit;
      Body
      goto Loop;
Exit:
```

- What are the Goto versions of the following?
  - Do…while: Test and Body

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

  - For loop: Init, Test, Update, and Body

"i=0" "i<n" "i++"

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

Do…while

```
Loop: Body
      if (Test) goto Loop;
```

For loop

```
      Init
Loop: if (!Test) goto Exit;
      Body
      Update
      goto Loop;
Exit:
```

Compiling Loops

all jump instructions update the program counter (%rip)

While Loop:

C:
```
while ( sum != 0 ) {
    <loop body>
}
```
(Test)

x86-64:
```
loopTop:    testq %rax, %rax
            je    loopDone
            <loop body code>
            jmp   loopTop
loopDone:
```
(sum == 0; testq/je = ~Test)

Do-while Loop:

C:
```
do {
    <loop body>
} while ( sum != 0 )
```
(Test)

x86-64:
```
loopTop:
            <loop body code>
            testq  %rax, %rax
            jne    loopTop
loopDone:
```
(testq/jne = Test)

While Loop (ver. 2):

C:
```
while ( sum != 0 ) {
    <loop body>
}
```
(Test)

x86-64:
```
            testq %rax, %rax
            je    loopDone
loopTop:
            <loop body code>
            testq %rax, %rax
            jne   loopTop
loopDone:
```
(testq/je = ~Test; do-while loop; testq/jne = Test)

For-Loop → While-Loop

For-Loop:

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

While-Loop Version:

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

Caveat: C and Java have `break` and `continue`

- Conversion works fine for break
  - Jump to same label as loop exit condition
- But not continue: would skip doing *Update*, which it should do with for-loops
  - Introduce new label at *Update*

x86 Control Flow

- Condition codes
- Conditional and unconditional branches
- Loops
- **Switches**

```
long switch_ex
    (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

- Implemented with:
  - *Jump table*
  - *Indirect jump instruction*

Jump Table Structure

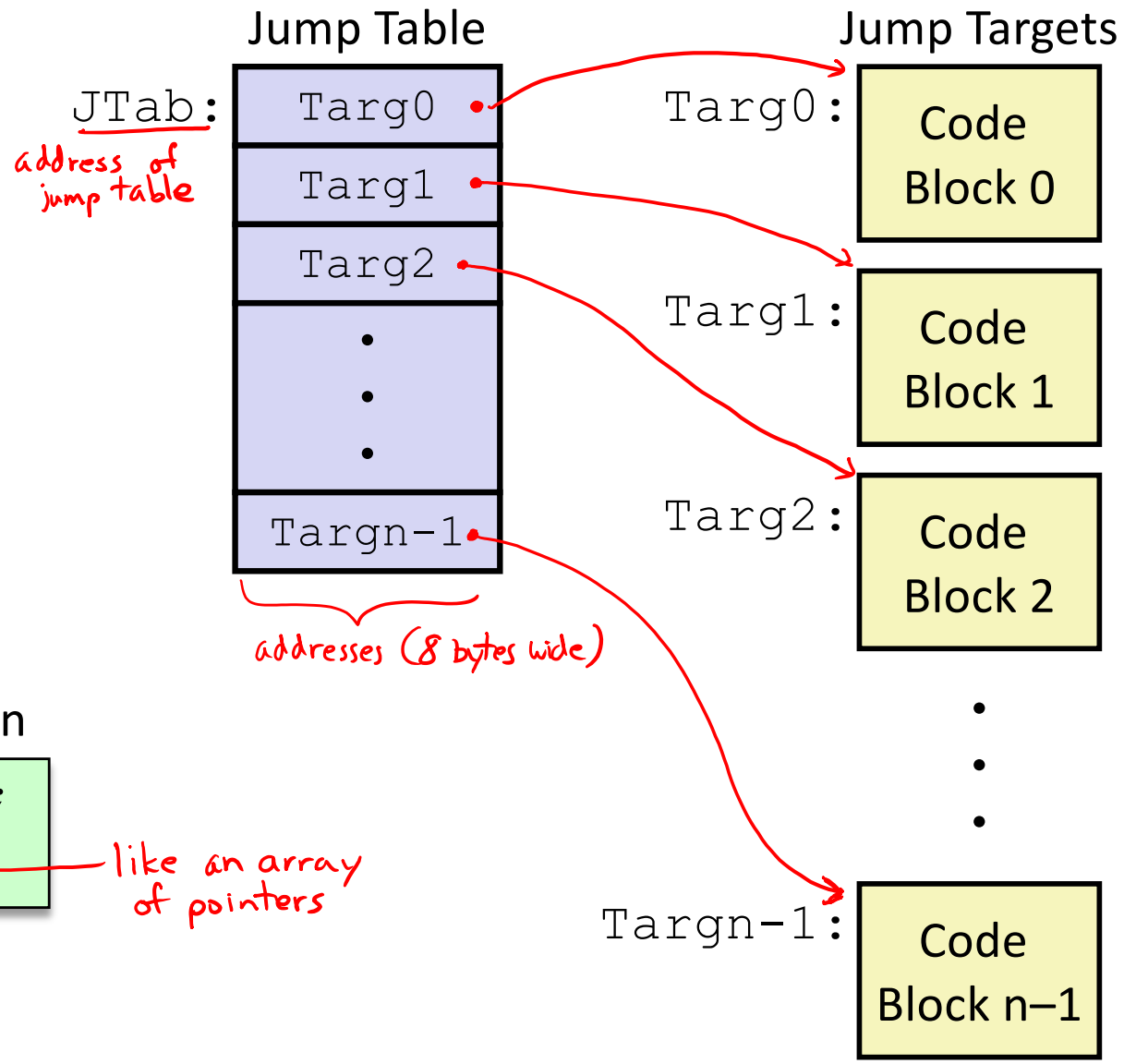
Switch Form

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

Approximate Translation

```
target = JTab[x];
goto target;
```

like an array of pointers



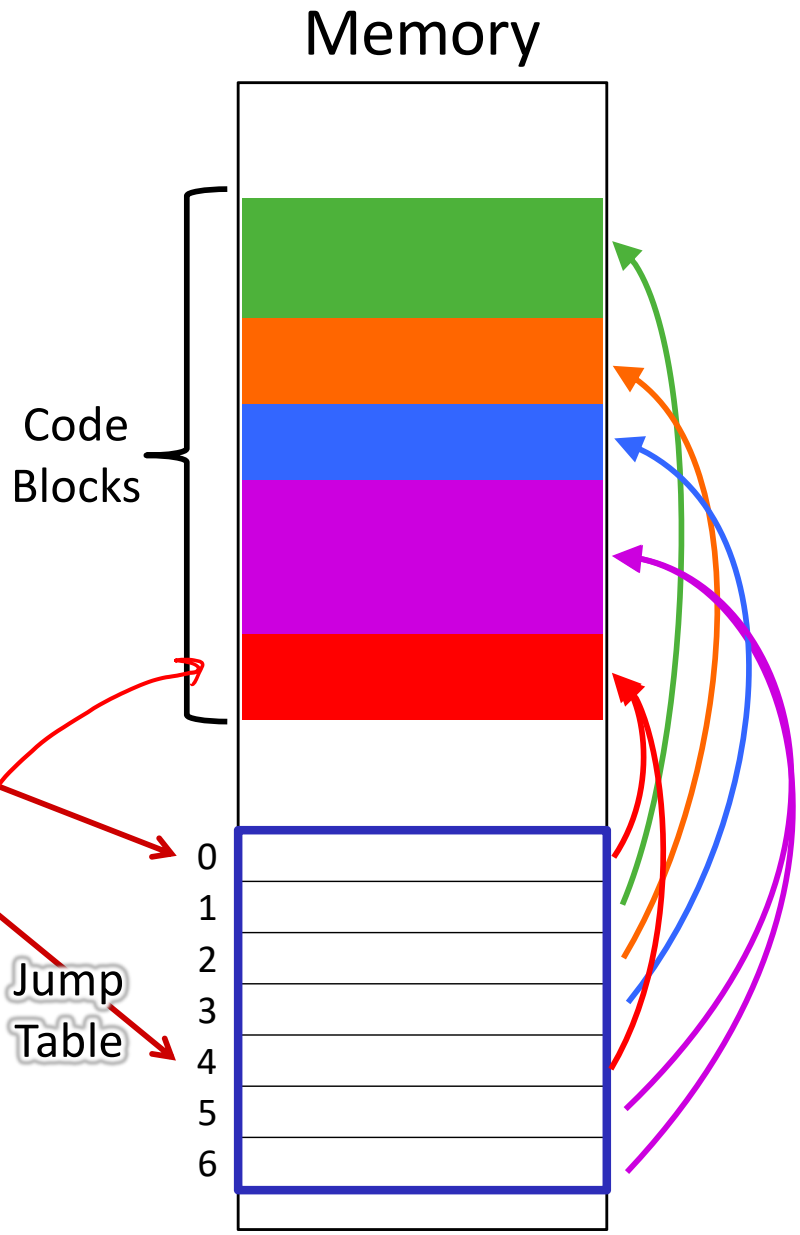
Jump Table Structure

C code:

```
switch (x) {
  case 1: <some code>
          break;
  case 2: <some code>
  case 3: <some code>
          break;
  case 5:
  case 6: <some code>
          break;
  default: <some code>
}
```

Use the jump table when x ≤ 6:

```
if (x <= 6)
  target = JTab[x];
  goto target;
else
  goto default;
```



Switch Statement Example

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

where?

Register	Use(s)
%rdi	1st argument (x)
%rsi	2nd argument (y)
%rdx	3rd argument (z)
%rax	return value

Note compiler chose to not initialize w

Take a look!

https://godbolt.org/z/dOWSFR

```
switch_eg:
    movq    %rdx, %rcx
    cmpq    $6, %rdi        # x:6
    ja      .L8             # default
    jmp     *.L4(,%rdi,8)   # jump table
```

a b

jump to default case if x>6 (unsigned)

jump above – unsigned > catches negative default cases

-1 > 6U → jump to default case

Switch Statement Example

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

```
switch_eg:
    movq    %rdx, %rcx
    cmpq    $6, %rdi        # x:6
    ja      .L8             # default
    jmp     *.L4(,%rdi,8)   # jump table
```

Indirect jump

D + Ri * S

addr of jump table

x

sizeof (void*)

Jump table

following data is a "quad word" = 8 bytes

address

```
.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
```

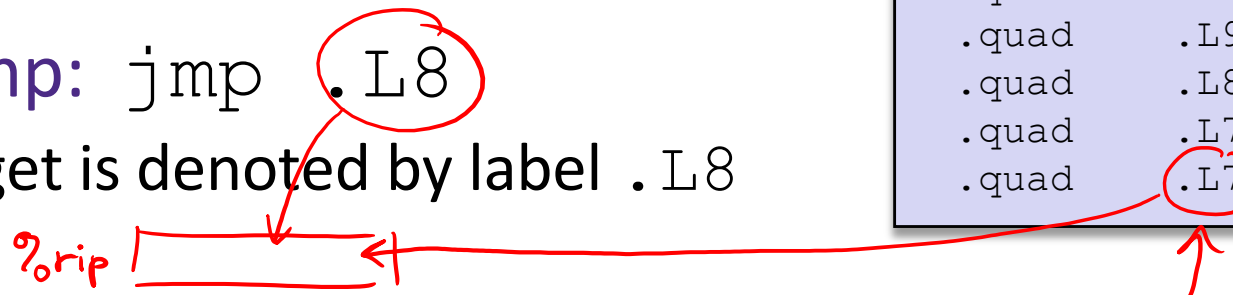
Assembly Setup Explanation

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

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
```

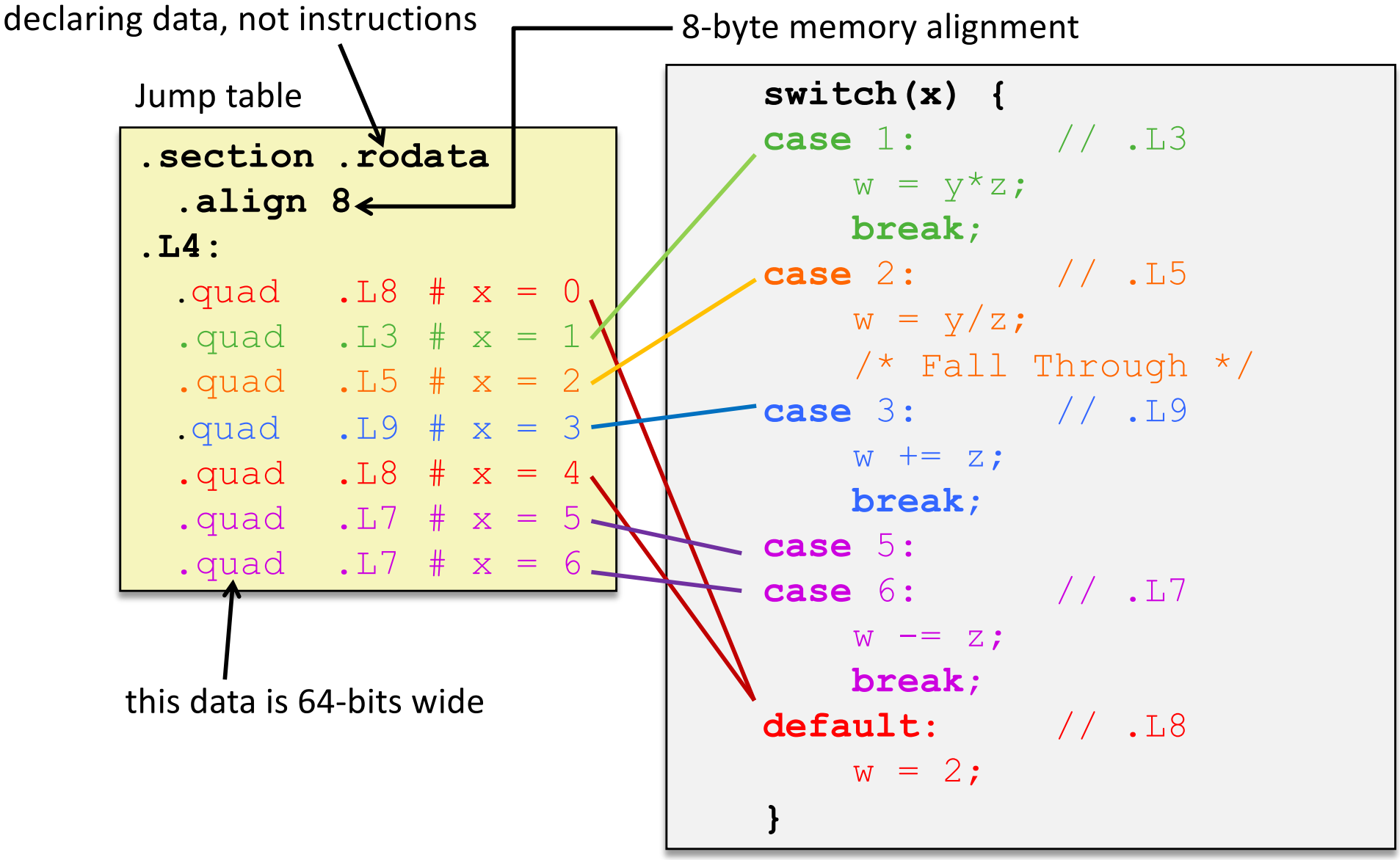
- Direct jump: `jmp .L8`
  - Jump target is denoted by label `.L8`



- Indirect jump: `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 $0 \le x \le 6$

Mem[D+Reg[Ri]*S]

Jump Table


declaring data, not instructions
8-byte memory alignment
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
this data is 64-bits wide
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;
  . . .
}
```

Register	Use(s)
%rdi	1st argument (x)
%rsi	2nd argument (y)
%rdx	3rd argument (z)
%rax	Return value

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

Handling Fall-Through

```
long w = 1;
    . . .
switch (x) {
    . . .
  case 2:   // .L5
    w = y/z;
  /* Fall Through */
  case 3:   // .L9
    w += z;
    break;
    . . .
}
```

```
case 2:
        w = y/z;
        goto merge;
```

```
case 3:
        w = 1;
merge:
        w += z;
```

More complicated choice than "just fall-through" forced by "migration" of `w = 1;`

- *Example compilation trade-off*

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

Register	Use(s)
%rdi	1st argument (x)
%rsi	2nd argument (y)
%rdx	3rd argument (z)
%rax	Return value

```
long w = 1;
    . . .
switch (x) {
    . . .
  case 2:   // .L5
    w = y/z;
  /* Fall Through */
  case 3:   // .L9
    w += z;
    break;
    . . .
}
```

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

Code Blocks (rest)

Register	Use(s)
%rdi	1st argument (x)
%rsi	2nd argument (y)
%rdx	3rd argument (z)
%rax	Return value

```
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
```