x86-64 Programming III

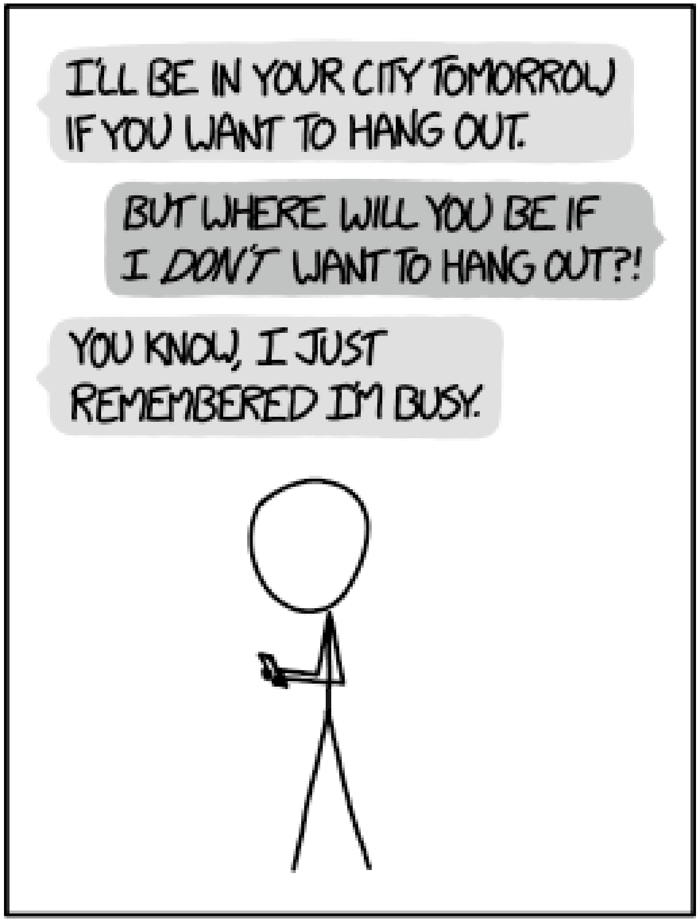
CSE 351 Spring 2019

Instructor:

Ruth Anderson

Teaching Assistants:

Gavin Cai
Jack Eggleston
John Feltrup
Britt Henderson
Richard Jiang
Jack Skalitzky
Sophie Tian
Connie Wang
Sam Wolfson
Casey Xing
Chin Yeoh



WHY I TRY NOT TO BE PEDANTIC ABOUT CONDITIONALS.

http://xkcd.com/1652/

Administrivia

- Lab 1b due TONIGHT Monday (4/22)
  - Submit `bits.c` and `lab1Breflect.txt`
- Homework 2 due Wednesday (4/24)
  - On Integers, Floating Point, and x86-64
- Lab 2 (x86-64), due Wednesday (5/01)
  - Ideally want to finish well before the midterm

- **Midterm** (Fri 5/03, 4:30-5:30pm in KNE 130)

GDB Demo

- See files on course schedule:
    - `mov.s` – assembly file
    - `mov_demo.txt` – commands to for use with gdb
    - `mov_tui_demo.txt` – commands for gdb using TUI
- The `movz` and `movs` examples on a real machine!
- You will need to use GDB to get through Lab 2
- 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

Conditionals are how we implement if statements, while/for loops, switches, etc in assembly

- Come in instruction *pairs*

Given the previous operation (e.g. addq 5, (p)) we compare to zero

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

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

		(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`
`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

subtraction; e.g.saying a == b is the same as saying a - b == 0. Hence the cmpq command matches the "comparison to zero" framework from last slide.

compare

- 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`	`b&a > 0U`
`jb`	"Below" (unsigned <)	`b <  a`	`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`

this last example shows how the test function can behave like a mask

Choosing instructions for conditionals

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

		① **`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`
② **`jge`**	"Greater or equal"	$\overset{x}{b}$ >= $\overset{3}{a}$	`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`

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

do this if x ≥ 3

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

x

x-3≥0

x≥3

(if)

(else)

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;
}
```

Vote at http://pollev.com/rea

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

x-y

$x-y \le 0$

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

$x \le y$

~~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"	② $\underline{\overset{x}{b} == \overset{y}{a}}$	③ $\underline{b\&a == 0}$
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"	① $\underline{\overset{x}{b} < \overset{3}{a}}$	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

```
     %al       %bl
if (x < 3 && x == y) {
  return 1;
} else {
  return 2;  ← do this if either %al or %bl are False
}
```

cmp_ behaves like subtraction and setl implements the < check; so we have x - 3 < 0

```
①  cmpq $3, %rdi  } %al = (x<3)
   setl %al       }

②  cmpq %rsi, %rdi  } %bl=(x==y)
   sete %bl         }

③  testb %al, %bl
   je T2  ← jump to T2 if (%al & %bl) == 0
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!!! Do not write this in your code!**

Compiling Loops

C/Java code:

cond
test

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

Assembly code:

! cond
! ~~test~~

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

e.g. just use another jump statement within the loop to jump to the top of the loop (after checking any desired conditionals, like the condition on a while loop to keep iterating)

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

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

"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:

Test

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

x86-64:

sum == 0

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

~Test

Do-while Loop:

C:

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

Test

x86-64:

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

Test

While Loop (ver. 2):

C:

Test

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

x86-64:

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

~Test

do-while loop

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*

e.g. since we update last in our while loop implementation, a continue statement (which just skips the rest of the code in a given iteration of a for loop) would get stuck in an infinite loop as it would skip the update! Instead, we will just jump straight to the 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*

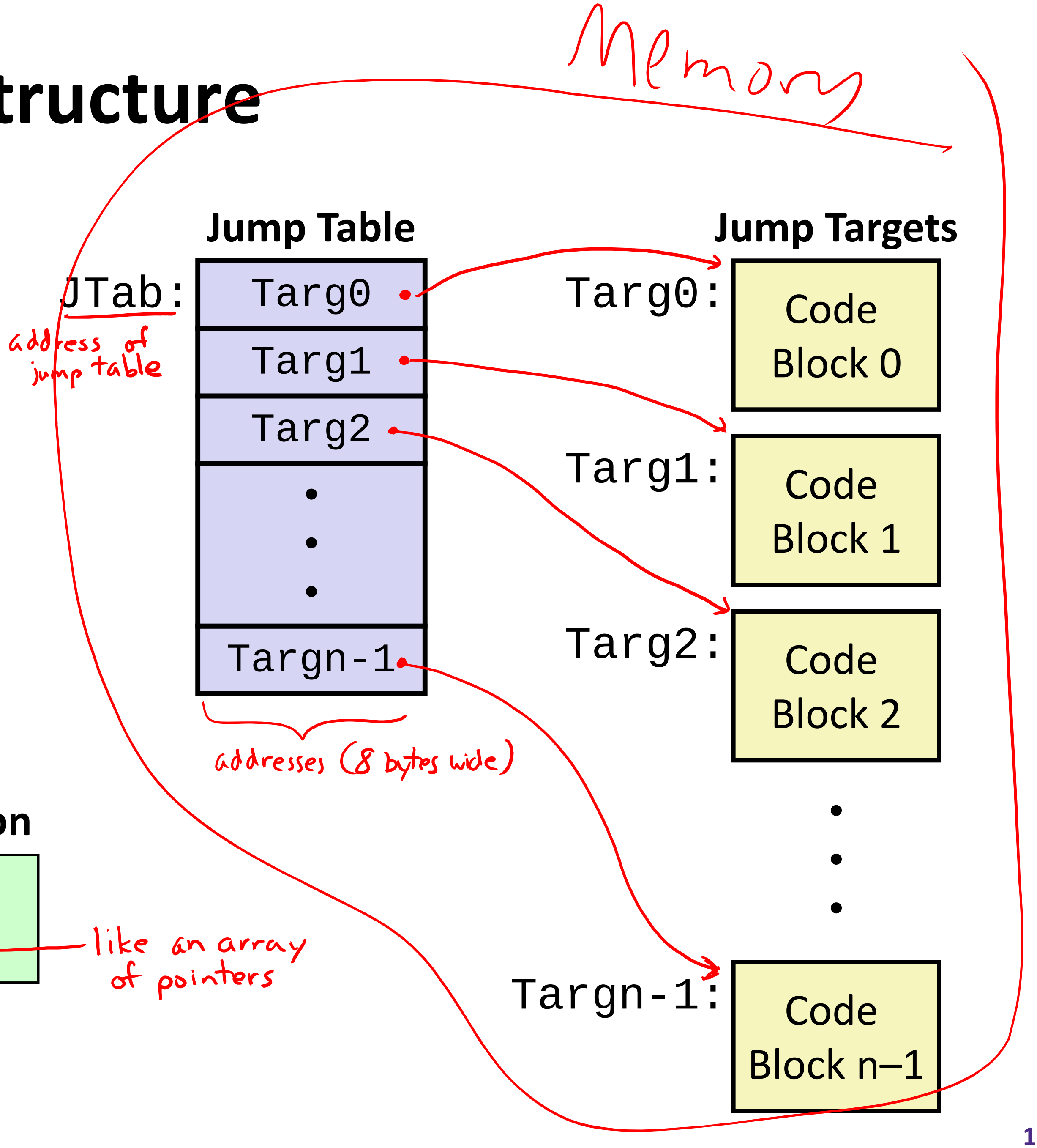
Jump Table Structure

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;
```



address of jump table

addresses (8 bytes wide)

like an array of pointers

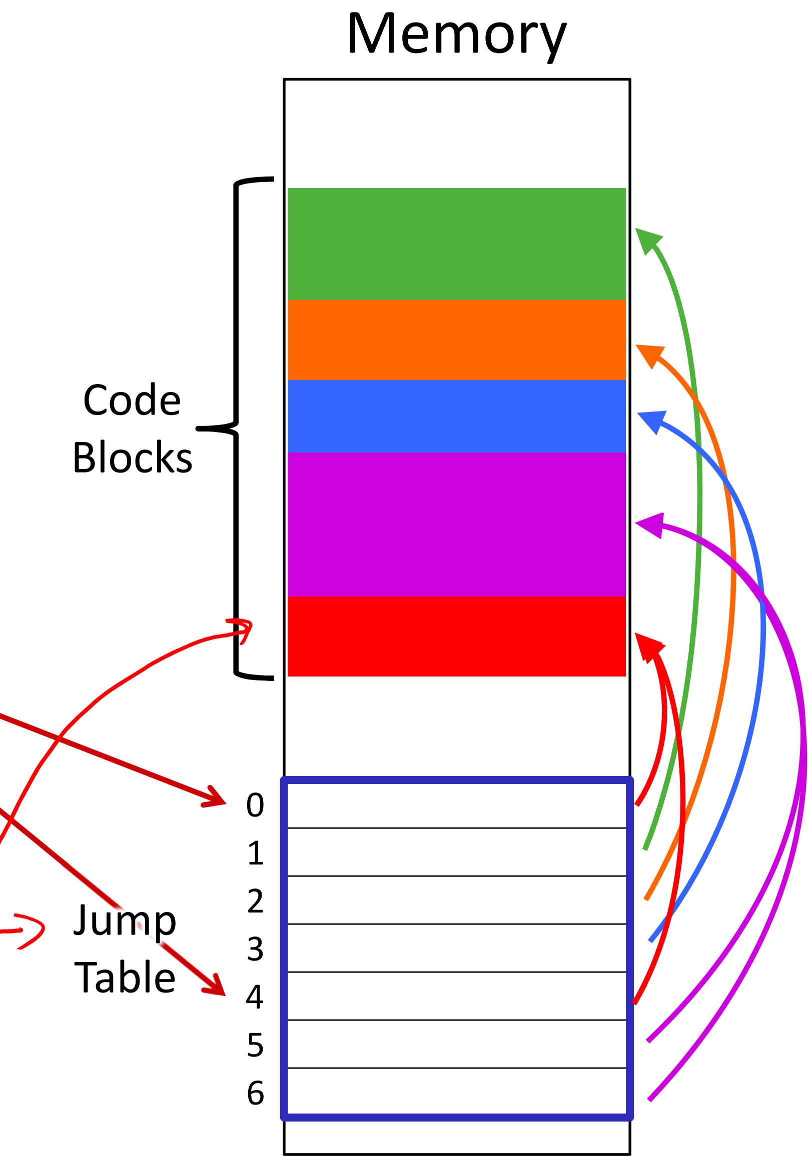
Jump Table Structure

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

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

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

where?

Note compiler chose to not initialize w

```
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)

Take a look!

https://godbolt.org/z/dOWSFR

L4 points to the start of the jump table, and the addresses take up 8 byte each; so the (, %rdi, 8) behaves like L4[%rdi] = L4[x] … e.g. it is indexing the jump table which contains the addresses of the different code blocks for each case in switch

jump above – unsigned > catches negative default cases

-1 > 6U → jump to default case

e.g. due to modular arithmetic, negative numbers when treated as unsigned wrap back around and are much larger than 6

Switch Statement Example

```
long switch_ex(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
```

address

following data is a "quad word" = 8 bytes

namely its our address for the given code block

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

Indirect jump

$D + R_i * S$

addr of jump table

x

sizeof (void*)

e.g. L4 references some point in memory

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`

%rip

- **Indirect jump:** `jmp *.L4(,%rdi,8)`

Mem[D+ Reg[Ri]*S]

  - 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

e.g. index x in the jump table

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
```

```
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;
}
```

this data is 64-bits wide

8 bytes. Remember the suffixes for the x86 commands:
b = byte
w = word = 2 bytes
l = double word = 4 bytes
q = quad word = 8 bytes

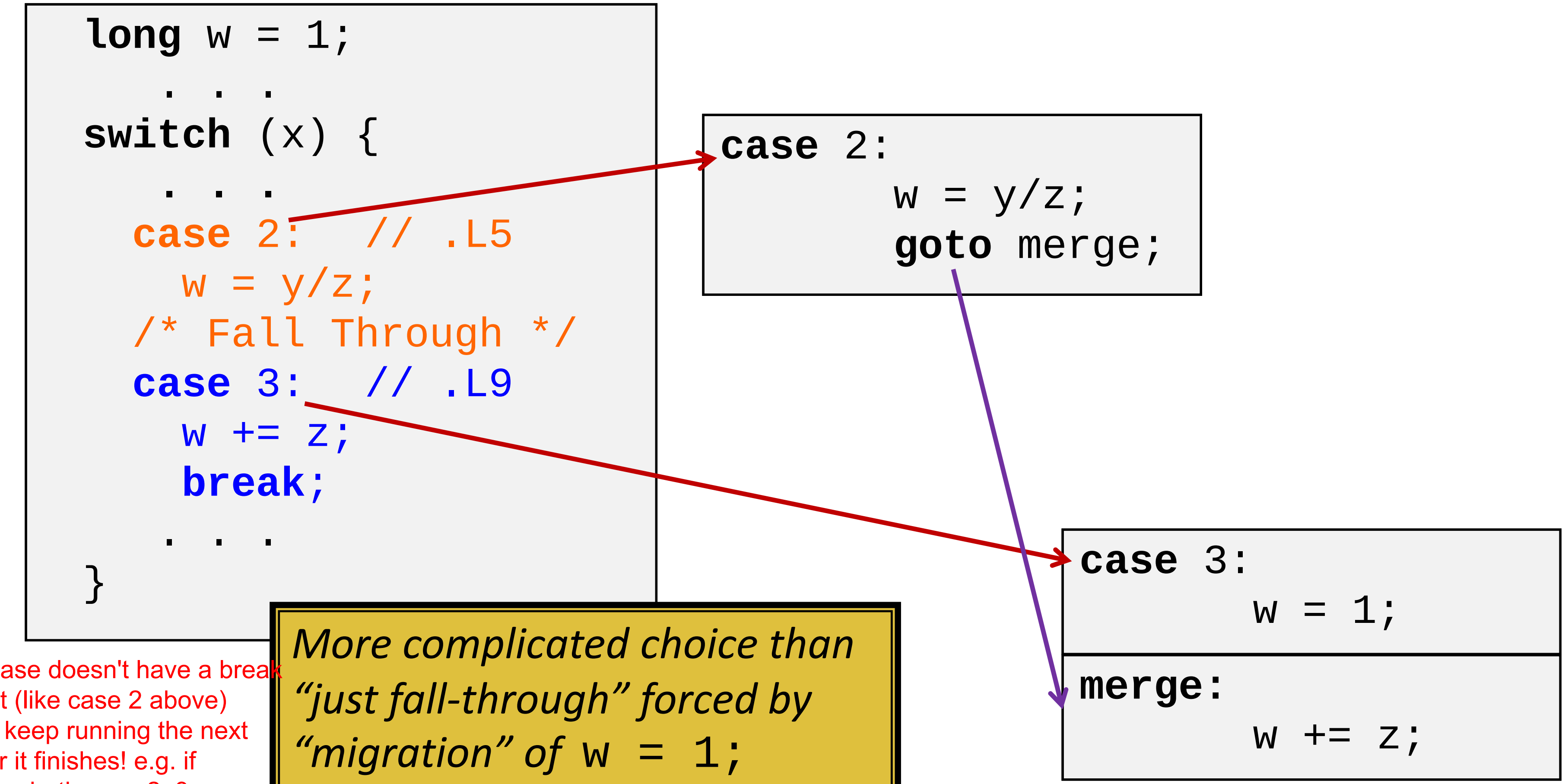
Code Blocks (x == 1)

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

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

```
.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;
```

e.g. If a case doesn't have a break statement (like case 2 above) it will just keep running the next case after it finishes! e.g. if x == 2 then both case 2, 3 are run.

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
```

sign extend %rax to %rdx:%rax into a 128 bit int

should this be %rdx = z?

see textbook section 3.5.5 for the formatting of the idivq command. Essentially, it pulls its numerator from %rax (which is why we move y there first)

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
```