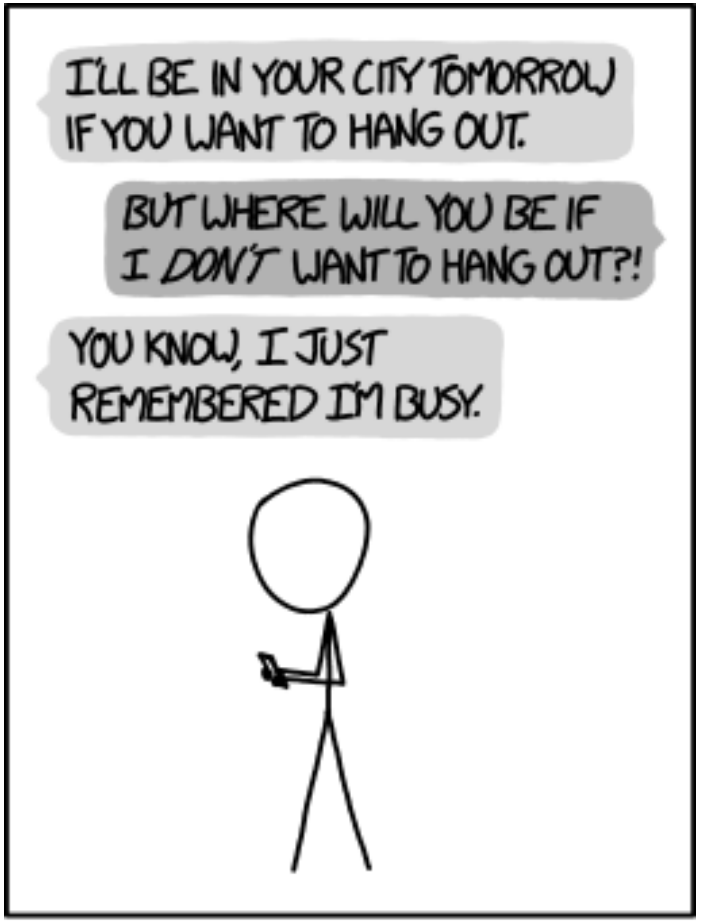
x86-64 Programming III

CSE 351 Summer 2022

Instructor:
Kyrie Dowling

Teaching Assistants:
Aakash Srazali
Allie Pfleger
Ellis Haker



WHY I TRY NOT TO BE PEDANTIC ABOUT CONDITIONALS.

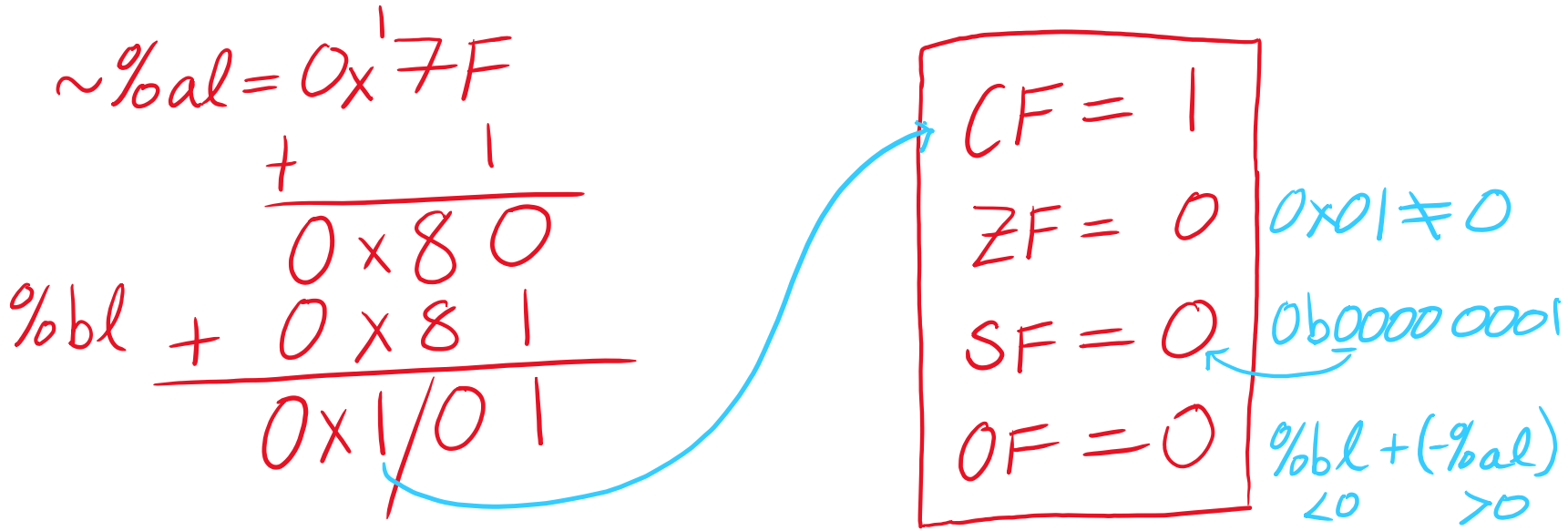
http://xkcd.com/1652/

Relevant Course Information

- hw7 due tonight, hw8 due Fri
- Lab 2 due next Friday (7/22)
  - Can start in earnest after today's lecture!
  - See GDB Tutorial and Phase 1 walkthrough in Section 4 Lesson on Ed
- Unit Portfolio 1 due Friday
  - Ideally your video should be no more than five minutes
  - Work on explaining your thought process for solving the problem *at a high level*

Example Condition Code Setting

- Assuming that `%al` = `0x80` and `%bl` = `0x81`, which flags (CF, ZF, SF, OF) are set when we execute **`cmpb %al, %bl`**? → computes %bl − %al = %bl + ~%al + 1



Using Condition Codes: Jumping

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

don't worry about the details

Instruction	Condition	Description → always compared to zero
`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 → 0b0000 0000 = 0x00
True → 0b0000 0001 = 0x01

same instruction suffixes as j*

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 “<”)

Reading Condition Codes

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

- `set*` Instructions (e, eg, l, ...)
  - Set a low-order byte to 0 or 1 based on condition codes
  - Operand is byte register (*e.g.*, `%al`) 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
```

y, x

zero-extend → movzbl

setg ← lowest byte

movzbl ← whole reg

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

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*

```
    (1) 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
(2) jne:    b|a != 0
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 >)	$b >_U a$	`b&a > 0U`
`jb`	"Below" (unsigned <)	$b <_U a$	`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
```

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 >)	$b >_U a$	b&a > 0U
jb	"Below" (unsigned <)	$b <_U a$	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 $x \geq 3$

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

labels

Practice Question 1

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
```
x-y

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

C.
```
testq %rsi, %rdi
jle   .L4
```
x & y

D.
```
testq %rsi, %rdi
jg    .L4
```
x & y

E. We're lost…

```
absdiff:
    _____________________
    _____________________
                          # x > y:
    movq    %rdi, %rax
    subq    %rsi, %rax
    ret
.L4:                      # x <= y:
    movq    %rsi, %rax    x-y <= 0
    subq    %rdi, %rax
    ret
```

less than or equal to (le)

Reading Review

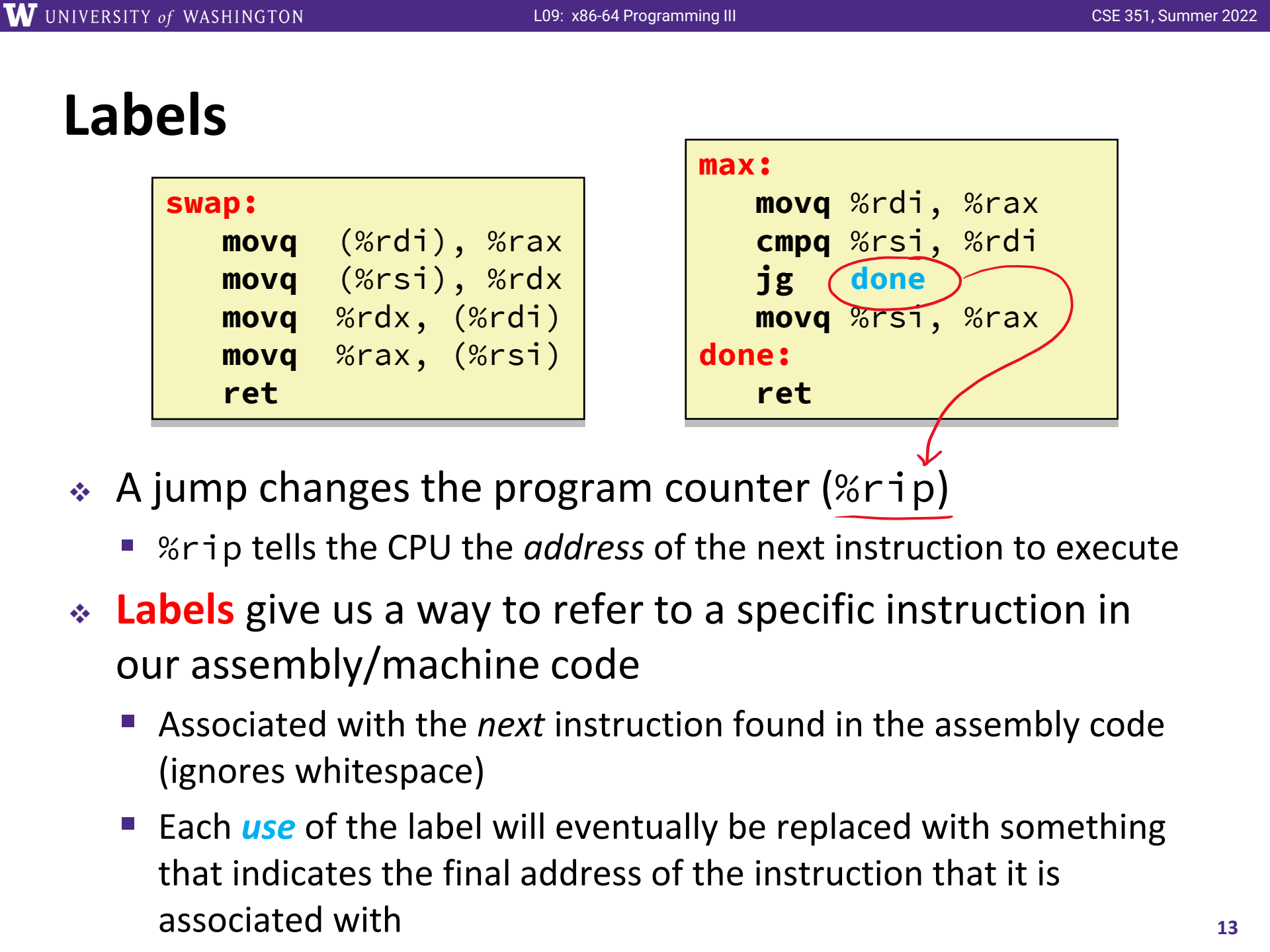
- Terminology:
  - Label, jump target
  - Program counter
  - Jump table, indirect jump

- Questions from the Reading?

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 instruction 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

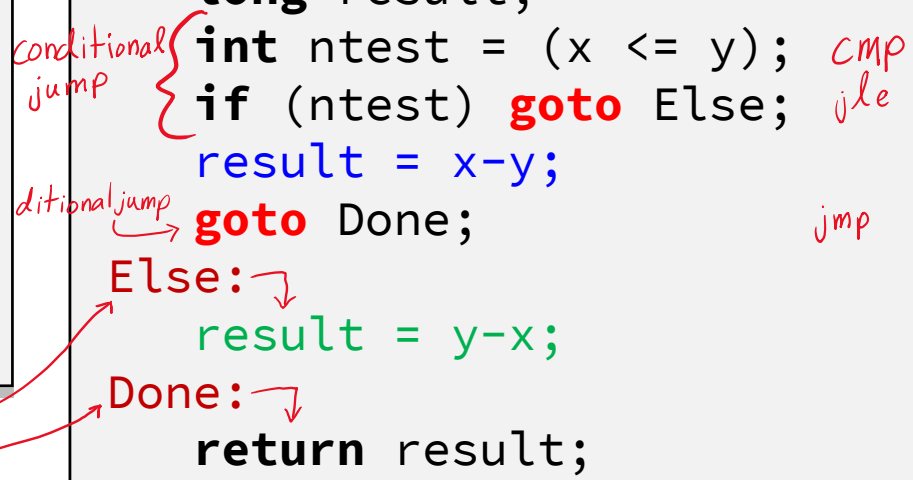
x86 Control Flow

- 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;
}
```



- C allows goto as means of transferring control
  - Closer to assembly programming style
  - Don't do this!! Bad!!!

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

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

x86-64:
```
            testq %rax, %rax
            je    loopDone
loopTop:
            <loop body code>
            testq %rax, %rax
            jne   loopTop
loopDone:
```
~Test

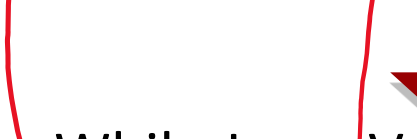
do-while loop

Test

For-Loop → While-Loop

For-Loop:

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



While-Loop Version:

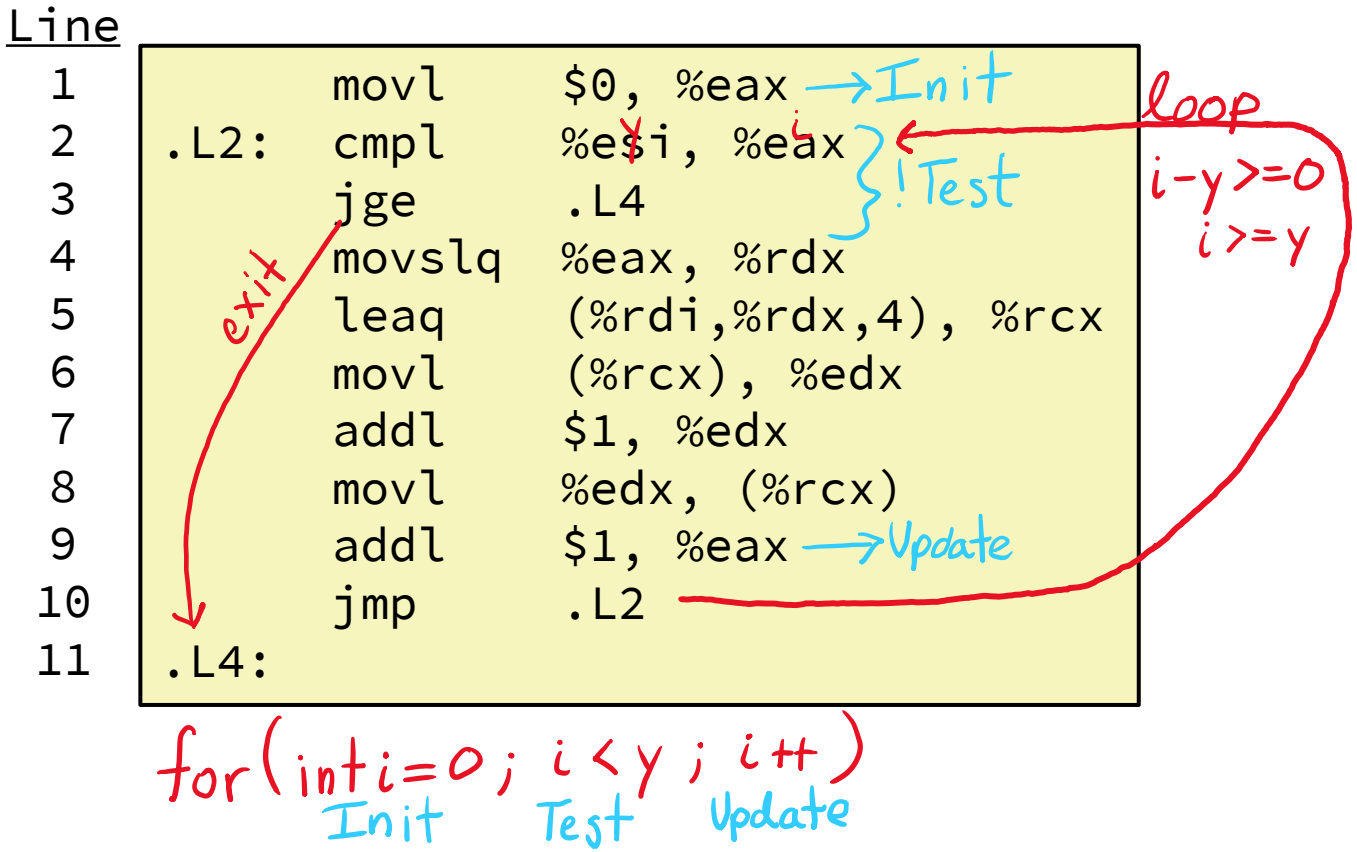
```
Init;
while (Test) {
    Body
    Update;
}
```

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*

Practice Question 2

- The following is assembly code for a for-loop; identify the corresponding parts (Init, Test, Update)
  - i → %eax, x → %rdi, y → %esi



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;
        case 7:
            w = y%z; break;
        default:
            w = 2;
    }
    return w;
}
```

Switch Statement Example

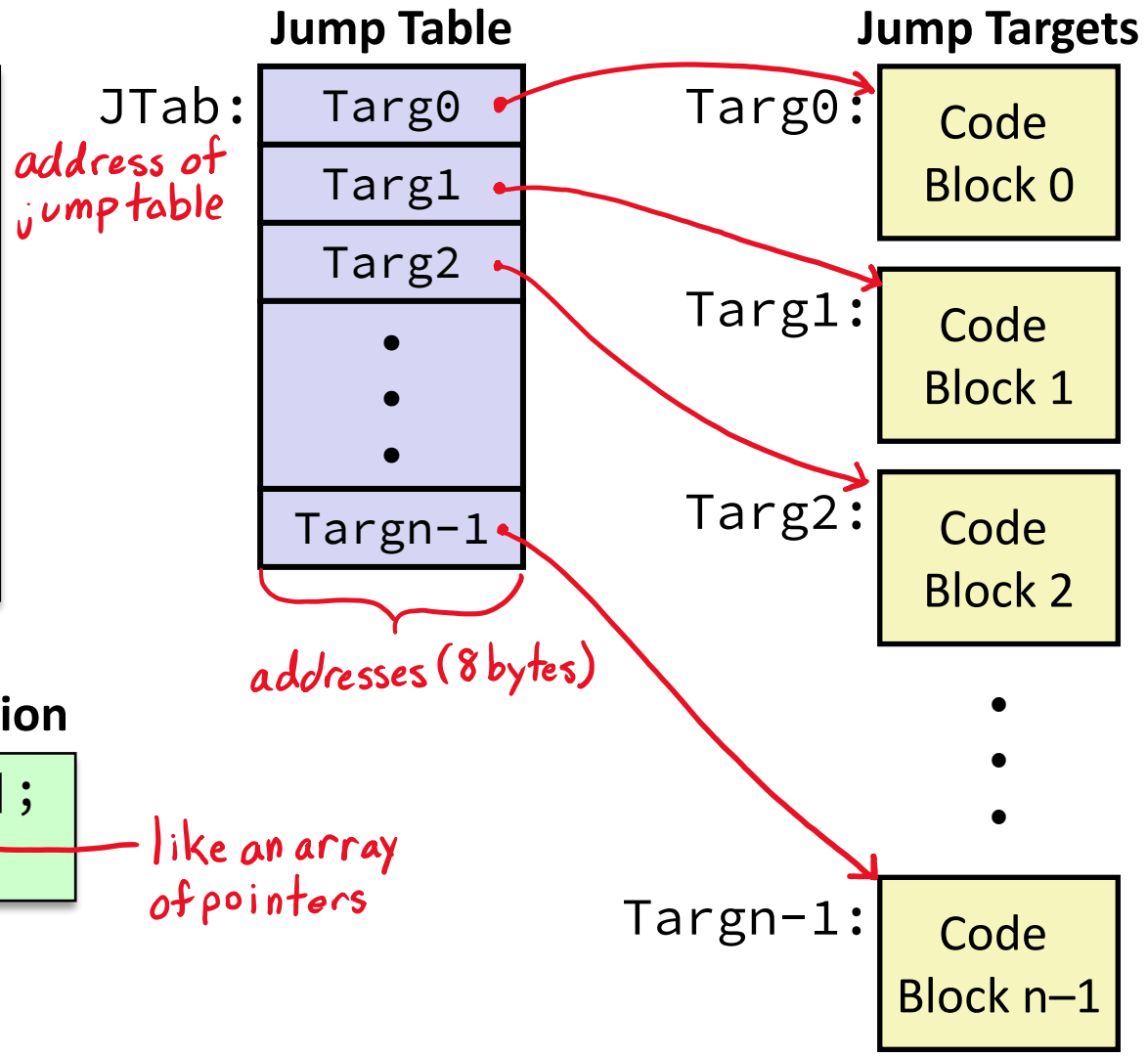
- Multiple case labels
  - Here: 5 & 6
- Fall through cases
  - Here: 2
- Missing cases
  - Here: 4

- 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

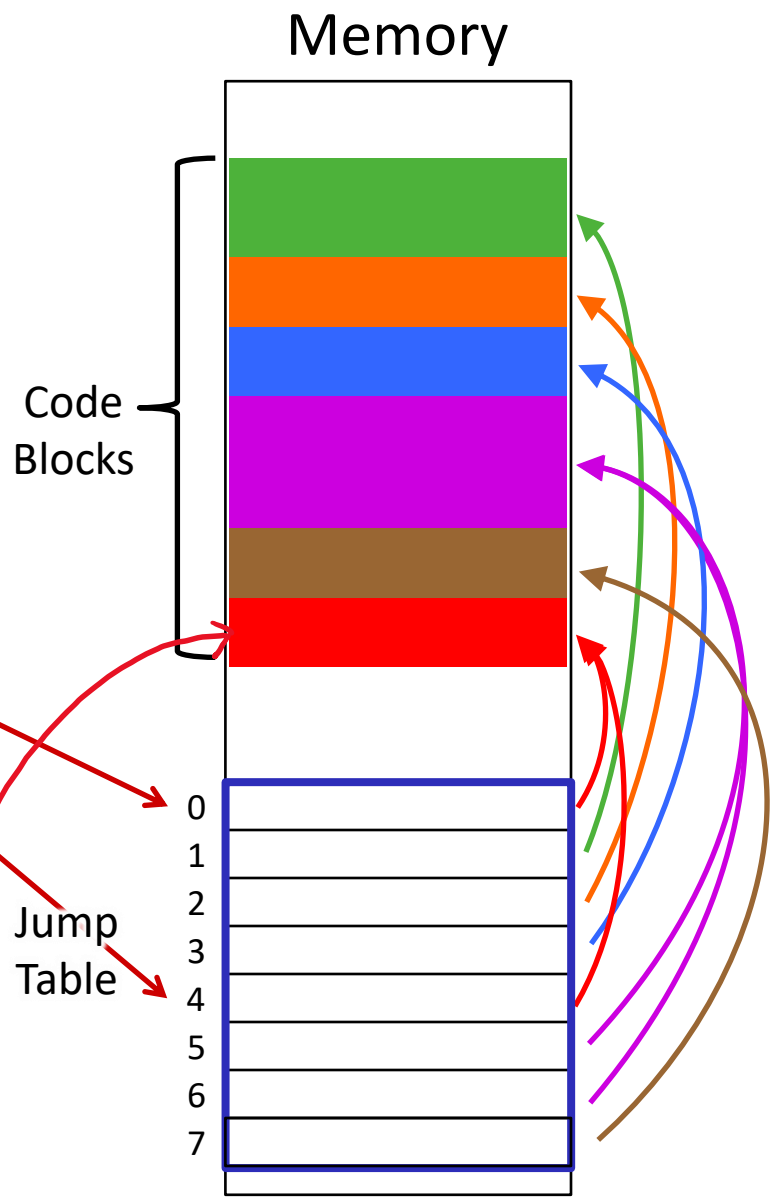
Jump Table Structure

C code:

```
switch (x) {
  case 1: <code> break;
  case 2: <code>
  case 3: <code> break;
  case 5:
  case 6: <code> break;
  case 7: <code> break;
  default: <code>
}
```

Use the jump table when x ≤ 7:

```
if (x <= 7)
  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_ex:
    movq    %rdx, %rcx
    cmpq    $7, %rdi        # x:7
    ja      .L9             # default
    jmp     *.L4(,%rdi,8)   # jump table
```

jump to default case if x>7 (unsigned)

Take a look!
https://godbolt.org/z/Y9Kerb

jump above – unsigned > catches negative default cases

-1 > 7U → jump to default

Switch Statement Example

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

```
switch_ex:
    movq    %rdx, %rcx
    cmpq    $7, %rdi        # x:7
    ja      .L9             # default
    jmp     *.L4(,%rdi,8)   # jump table
```

Indirect jump

D + Ri * S

addr of jtab

x

size of (void *)

Jump table

following data is a "quad word" = 8 bytes

address

```
.section    .rodata
  .align 8
.L4:
  .quad     .L9     # x = 0
  .quad     .L8     # x = 1
  .quad     .L7     # x = 2
  .quad     .L10    # x = 3
  .quad     .L9     # x = 4
  .quad     .L5     # x = 5
  .quad     .L5     # x = 6
  .quad     .L3     # x = 7
```

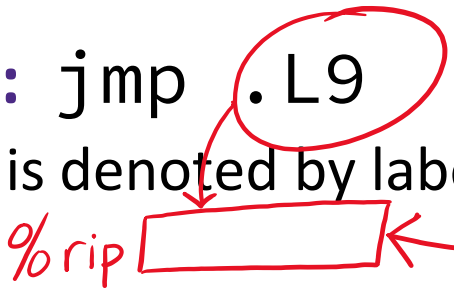
Assembly Setup Explanation

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

Jump table

```
.section    .rodata
  .align 8
.L4:
  .quad    .L9    # x = 0
  .quad    .L8    # x = 1
  .quad    .L7    # x = 2
  .quad    .L10   # x = 3
  .quad    .L9    # x = 4
  .quad    .L5    # x = 5
  .quad    .L5    # x = 6
  .quad    .L3    # x = 7
```

- **Direct jump:** `jmp .L9`
  - Jump target is denoted by label `.L9`



- **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 $0 \le x \le 7$

Mem[D+Reg[Ri]*S]

Summary

- Control flow in x86 determined by Condition Codes
  - Showed **C**arry, **Z**ero, **S**ign, and **O**verflow, though others exist
  - Set flags with arithmetic instructions (implicit) or Compare and Test (explicit)
  - Set instructions read out flag values
  - Jump instructions use flag values to determine next instruction to execute
  - Most control flow constructs (*e.g.*, if-else, for-loop, while-loop) can be implemented in assembly using combinations of conditional and unconditional jumps