The Hardware/Software Interface

x86-64 Programming IV

Instructors:
Justin Hsia, Amber Hu

Teaching Assistants:
Anthony Mangus
Grace Zhou
Jiuyang Lyu
Kurt Gu
Mendel Carroll
Naama Amiel
Rose Maresh
Violet Monserate
Divya Ramu
Jessie Sun
Kanishka Singh
Liander Rainbolt
Ming Yan
Pollux Chen
Soham Bhosale

Futurama, “I, Roommate” (Season 1, Episode 3)

Relevant Course Information

- Lab 1a regrade requests open on Gradescope
- Lab 1b submissions close tonight
- Lab 2 due next Friday (10/24)
  - Section 4 will prep you for Lab 2 – bring midterm reference sheet & your laptop!
  - Recommended GDB Tutorial in Ed Lessons

- Midterm: 10/27, 5:30 pm split across ARC 147, CSE2 G20, and SIG 134
  - You will be provided a fresh midterm reference sheet
  - You get 1 *handwritten*, double-sided cheat sheet (letter-size)
  - Form study groups and look at past exams!

Lecture Outline (1/4)

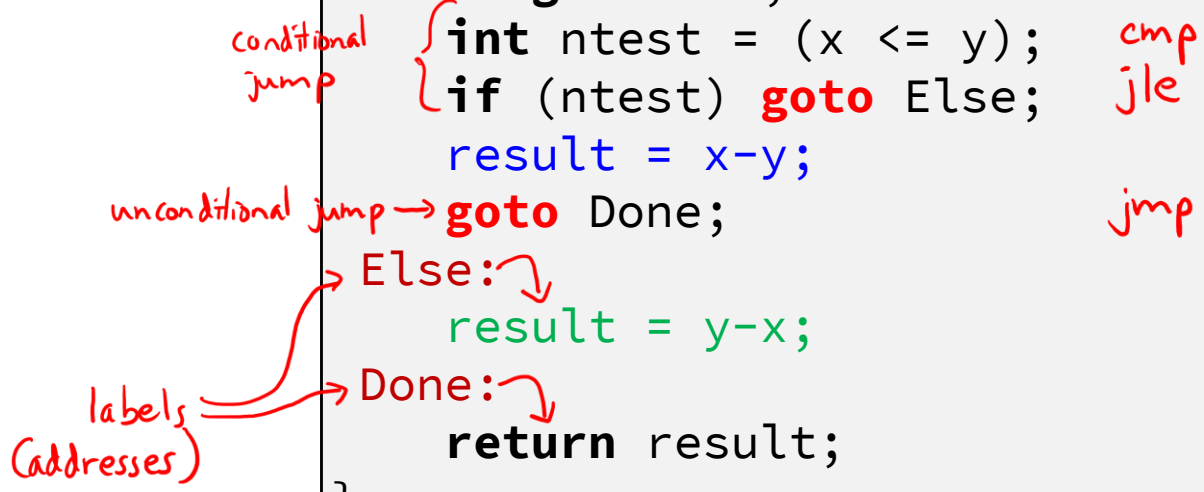
- **If-Else Statements**
- Loops
- Switch Statements
- Mainstream ISAs, Revisited

Aside: Labels & Jumps in C – `goto` (1/2)

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

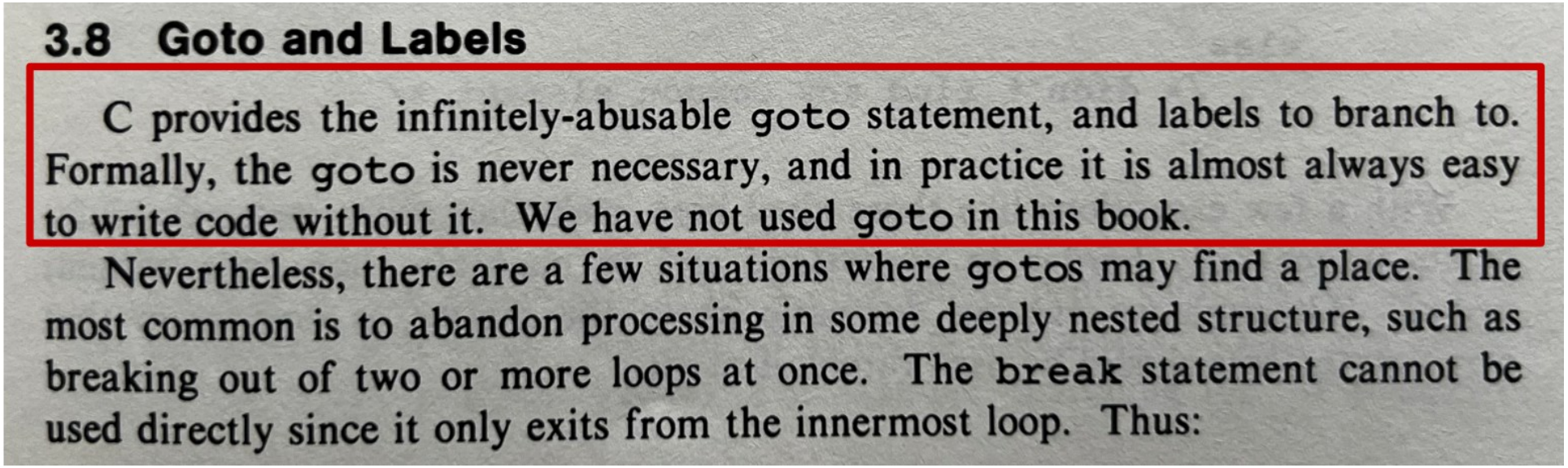
```
long absdiff(long x, long y) {
    long result;
    if (x > y)
        result = x-y;
    else
        result = y-x;
    return result;
}
```

```
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;
}
```



Aside: Labels & Jumps in C – `goto` (2/2)

- C allows `goto` as means of transferring control (`jump`)
  - Closer to assembly programming style
  - Generally considered bad coding style… listen to Kernighan & Ritchie:



3.8 Goto and Labels

C provides the infinitely-abusable `goto` statement, and labels to branch to. Formally, the `goto` is never necessary, and in practice it is almost always easy to write code without it. We have not used `goto` in this book.

Nevertheless, there are a few situations where `goto`s may find a place. The most common is to abandon processing in some deeply nested structure, such as breaking out of two or more loops at once. The `break` statement cannot be used directly since it only exits from the innermost loop. Thus:

If-Else Implementation (Review)

- **Goal:** Ensure that only one case/branch gets executed
  - Jumps in if-else statements almost always go later/forward, though not always
  - Fewer jumps and labels when using `ret`

- Examples: From last lecture

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

then
conditional (x > y)
else

```
abs:
    movl   %edi, %eax
    testl  %edi, %edi
    js     .L3
.L2: ret
.L3: negl  %eax
    jmp    .L2
```

else
conditional (x<0)
then
could be ret

If-Else Implementation Variations

- **Variation:** Which case/branch do you want to execute when the conditional is True versus False

- Example:

```
if (x < 3)   // x in %rdi
  y += 5;    // y in %rsi
else
  y = 10;
```

```
    cmpq $3, %rdi    # x-3
    jl   .L2         # x-3<0 → x<3
    movq $10, %rsi   #else
    jmp  .L3
.L2:                 # then
    addq $5, %rsi
.L3:
    # other code
```

```
    cmpq $2, %rdi    # x-2
    jg   .L2         # x>2 → x≥3 for integers
    addq $5, %rsi    # then
    jmp  .L3
.L2:                 # else
    movq $10, %rsi
.L3:
    # other code
```

Lecture Outline (2/4)

- If-Else Statements
- **Loops**
- Switch Statements
- Mainstream ISAs, Revisited

While-Loop Implementation (Review)

- Constructed with same building blocks as if-else statements
  - One of the jump targets *must* go backward to create a loop
  - The test condition must be contained in the loop body
- Loop variants based on:
  - When are conditionals evaluated: top or bottom of loop
  - How much jumping is involved: execution efficiency
- In the following pseudocode examples:
  - Example test condition (*Test*) is `c != 0` → `testq %rsi, %rsi`
  - `<loop body>` represents the loop body
  - Our labels will be called `loopTop` and `loopDone`

c & c = c

While-Loop Variants

	C/Java	x86-64 Assembly
Do-while:	`do { <loop body> } while ( c != 0 )` (Test)	`loopTop: <loop body code>` `testq %rax, %rax` `jne loopTop` (Test) `loopDone:`
While:	(Test) `while ( c != 0 ) { <loop body> }`	`loopTop: testq %rax, %rax` `je loopDone` (~Test) `<loop body code>` `jmp loopTop` `loopDone:` (c == 0)
While Alt:		`testq %rax, %rax` `je loopDone` (~Test) `loopTop: <loop body code>` (do-while loop) `testq %rax, %rax` `jne loopTop` (Test) `loopDone:`

While-Loop Example

- Fill in the mystery function, which contains a loop
  - %rdi → ar (**int** array), %esi → c (**int**), %eax → s (**int**)

```
mystery:
    movl    $0, %eax        #s = 0
    jmp     .L2             # does cond first
                            #  ↳ while loop
.L3:
    movslq  %esi, %rdx      # c in %rdx
    addl    (%rdi,%rdx,4), %eax
    subl    $1, %esi        # c = c-1
.L2:
    testl   %esi, %esi      # c
    jne     .L3             # c != 0
    ret
```

Test because # loops when taken

```
int mystery(int ar[], int c) {
    int s = 0;
    while (c != 0) {
        s = s + ar[c];
        c = c - 1;
    }
    return s;
}
```

For-Loop Implementation (Review)

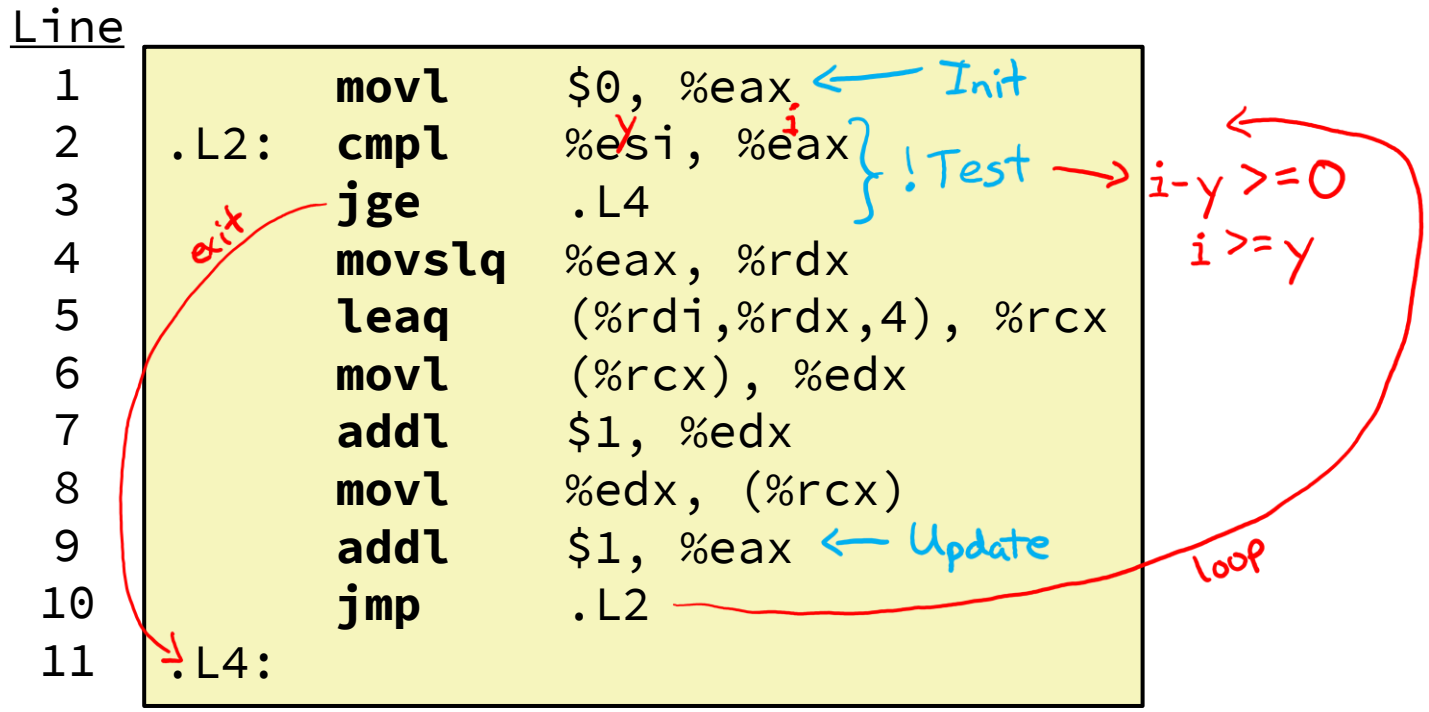
- A for-loop is just a more structured form of a while-loop!
  - Can use whichever implementation of while-loop as you'd like to

```
for (<init>; <test>; <update>) {
    <loop body>
}
```

```
<init>
while (<test>) {
    <loop body>
    <update>
}
```

Polling Question

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



for(int i = 0 ; i < y ; i++)
Init Test Update

Lecture Outline (3/4)

- If-Else Statements
- Loops
- **Switch Statements**
- Mainstream ISAs, Revisited

Switch Statement Example

- Implemented with:
  - *Jump table*
  - *Indirect jump instruction*
- Special cases:
  - Multiple case labels (5 & 6)
  - Fall through cases (2)
  - Missing cases (4) ???

Take a look!
https://godbolt.org/z/fsK1KEboW

```
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;
}
```

General Jump Table Structure

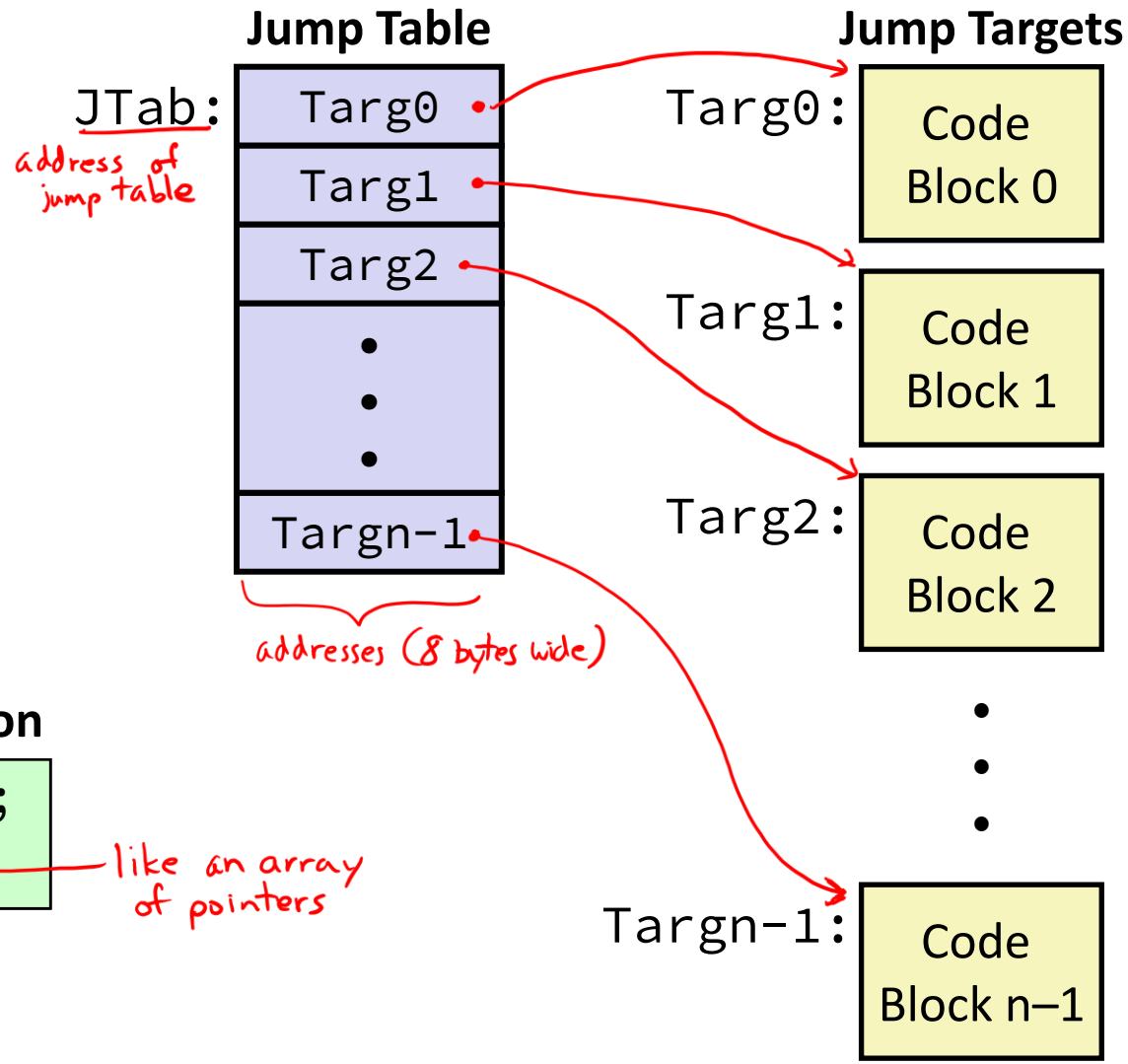
Switch Form

```
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



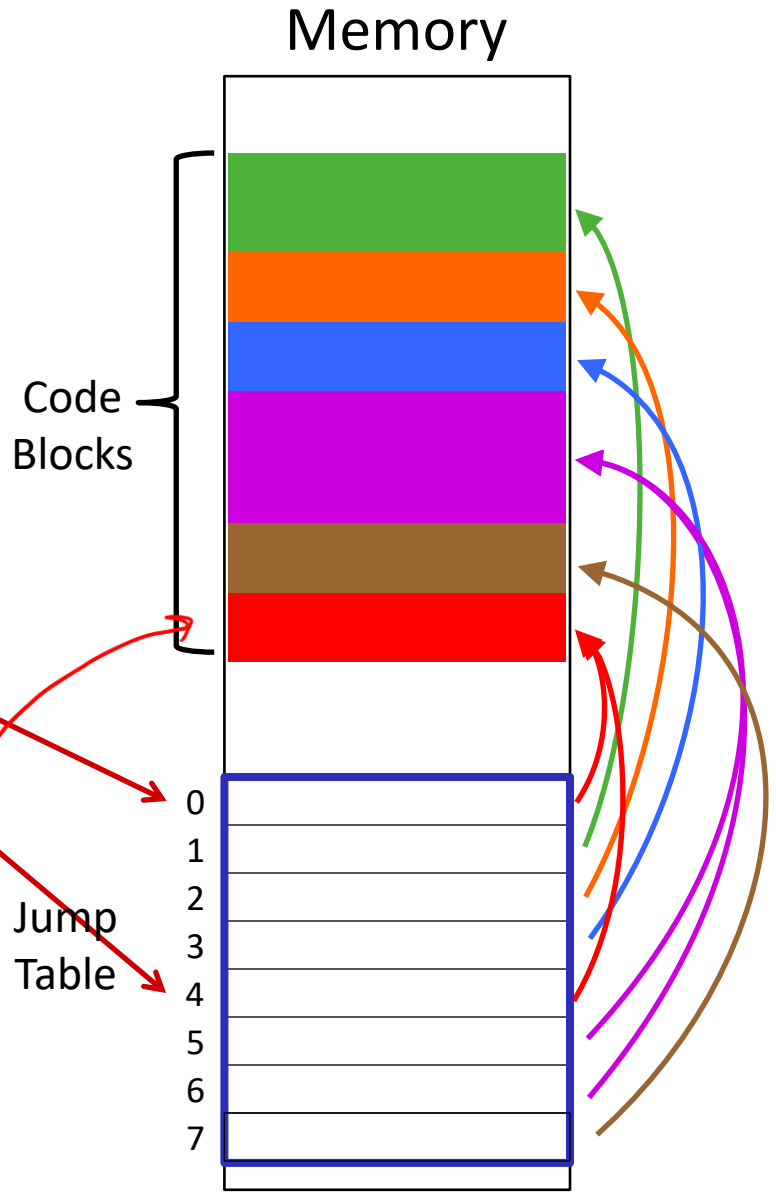
Example Jump Table Structure (Review)

C/Java 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;
```



Jump Table Usage (1/2)

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

Variable	Register
x	%rdi
y	%rsi
z	%rdx

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

a b

jump to default case if x>7 (unsigned)

jump above – unsigned > catches negative default cases

-1 > 7U → jump to default case

Jump Table Usage (2/2)

```
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
    ...
```

D + R[i] * S

D: addr of jump table

R[i]: x

S: sizeof(void*)

Indirect jump

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
```

address

following data is a "quad word" = 8 bytes

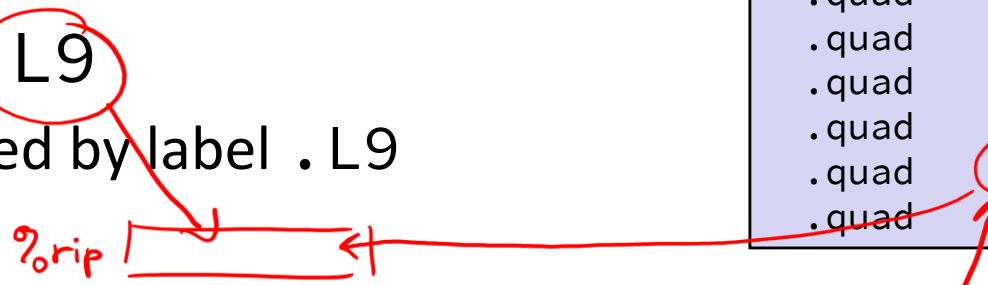
Direct vs. Indirect Jump

- 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:** `ja .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 $0 \le x \le 7$

Lecture Outline (4/4)

- If-Else Statements
- Loops
- Switch Statements
- **Mainstream ISAs, Revisited**

Mainstream ISAs, Revisited

x86

Designer	Intel, AMD
Bits	16-bit, 32-bit and 64-bit
Introduced	1978 (16-bit), 1985 (32-bit), 2003 (64-bit)
Design	CISC
Type	Register–memory
Encoding	Variable (1 to 15 bytes)
Branching	Condition code
Endianness	Little

ARM

Designer	Arm Holdings
Bits	32-bit, 64-bit
Introduced	1985
Design	RISC
Type	Register-Register
Encoding	AArch64/A64 and AArch32/A32 use 32-bit instructions, T32 (Thumb-2) uses mixed 16- and 32-bit instructions; ARMv7 user-space compatibility.[1]

RISC-V

Designer	University of California, Berkeley
Bits	32 · 64 · 128
Introduced	2010
Design	RISC
Type	Load-store
Encoding	Variable
Endianness	Little[1][3]

Windows desktop/laptops
(Core i3, i5, i7, Ryzen)
x86-64 Instruction Set

Smartphone-like devices
(iPhone, Android, Raspberry Pi)
Apple products (ca. 2020-)
(Macbook, Mac Mini)
ARM Instruction Set

Mostly research
(some traction in embedded)
RISC-V Instruction Set

Discussion Question

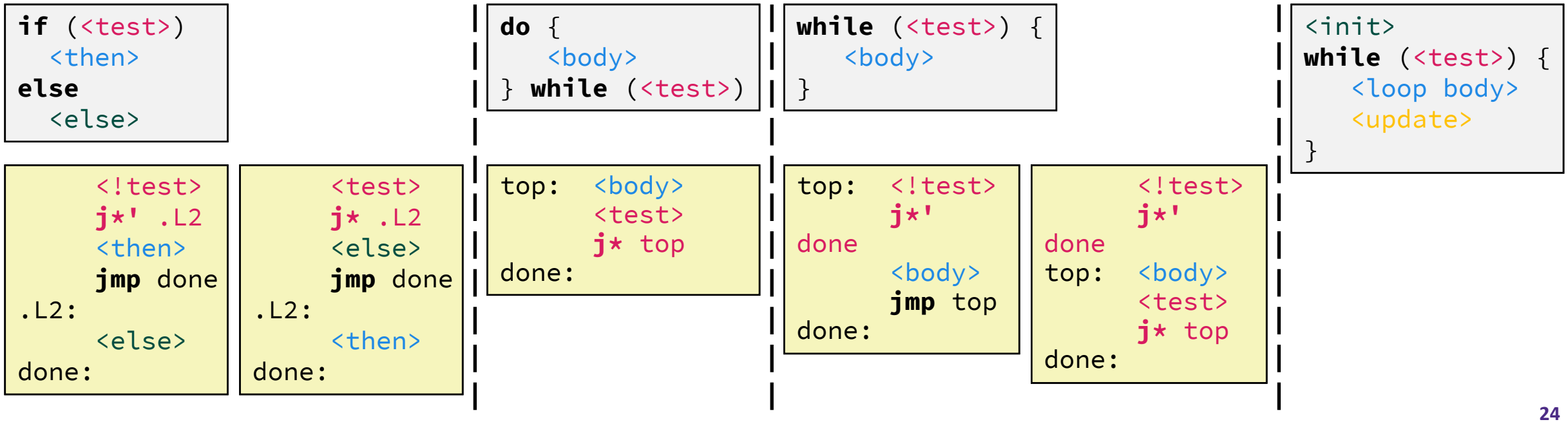
- Discuss the following question(s) in groups of 3-4 students
  - I will call on a few groups afterwards so please be prepared to share out
  - Be respectful of others' opinions and experiences

- We taught you assembly using x86-64; you didn't have a choice
  - What are some of the advantages and drawbacks of this choice?

  - What are some possible assumptions we are making about our students or values we are forcing on our students with this choice?

Summary (1/2)

- Most control flow constructs (*e.g.*, if-else, for-loop, while-loop) can be implemented in assembly using combinations of conditional and unconditional jumps
  - Differences come from placement of jumps and whether they jump forward or backwards in code



Summary (2/2)

- Switch statements can be implemented using *jump tables* and *indirect jump instructions*
  - Jump tables are arrays of pointers to code blocks
  - Indirect jump jumps to address stored somewhere in memory instead of target specified in instruction

```
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>
}
```

```
cmpq $7, %rdi
ja   .L9  # default
jmp  *.L4(,%rdi,8)
```

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
```