

x86-64 Programming III

CSE 351 Spring 2021

Instructor:

Ruth Anderson

Teaching Assistants:

Allen Aby

Joy Dang

Alena Dickmann

Catherine Guevara

Corinne Herzog

Ian Hsiao

Diya Joy

Jim Limprasert

Armin Magness

Aman Mohammed

Monty Nitschke

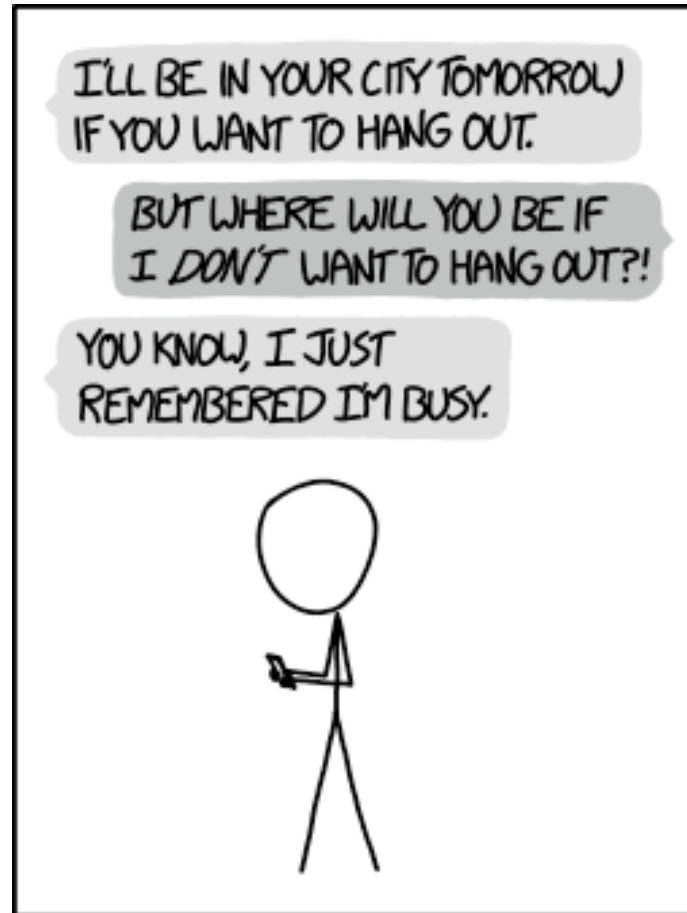
Allie Pflieger

Neil Ryan

Alex Saveau

Sanjana Sridhar

Amy Xu



<http://xkcd.com/1652/>

Administrivia

- ❖ hw8 due TONIGHT (4/19) @ 11:59 pm
- ❖ Lab 1b, due TONIGHT, Monday 4/19 @ 11:59 pm
 - Submit `aisle_manager.c`, `store_client.c`, and `lab1Breflect.txt`
 - Submissions that fail the autograder get a **ZERO**
 - No excuses – make full use of tools & Gradescope’s interface
- ❖ Unit Summary #1 due this Friday (4/23) on Gradescope
- ❖ Lab 2 (x86-64) due next Friday (4/30)
 - Learn to read x86-64 assembly and use GDB
- ❖ **Questions Docs:** Use @uw google account to access!!
 - <https://tinyurl.com/CSE351-21sp-Questions>

Reading Review

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

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*

```

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

```

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

Choosing instructions for conditionals

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

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

```

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:   aLSB == 0
jne:  aLSB == 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;

```

```

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

```

Practice Question 1

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

- 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...

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

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/Tfrv33>

```

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 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 (jump)
 - Closer to assembly programming style
 - Generally considered bad coding style

Compiling Loops

C/Java code:

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

Assembly code:

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

- ❖ 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

While Loop:

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

x86-64:

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

loopDone:
```

Do-while Loop:

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

x86-64:

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

loopDone:
```

While Loop (ver. 2):

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

x86-64:

```

            testq %rax, %rax
            je     loopDone

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

loopDone:
```

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)
 - $i \rightarrow \%eax$, $x \rightarrow \%rdi$, $y \rightarrow \%esi$

```
.L2:  movl    $0, %eax
      cmpl   %esi, %eax
      jge   .L4
      movslq %eax, %rdx
      leaq  (%rdi,%rdx,4), %rcx
      movl  (%rcx), %edx
      addl  $1, %edx
      movl  %edx, (%rcx)
      addl  $1, %eax
      jmp   .L2
.L4:
```

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

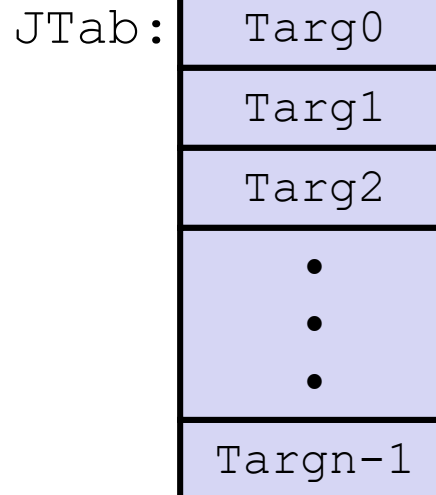
Switch Form

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

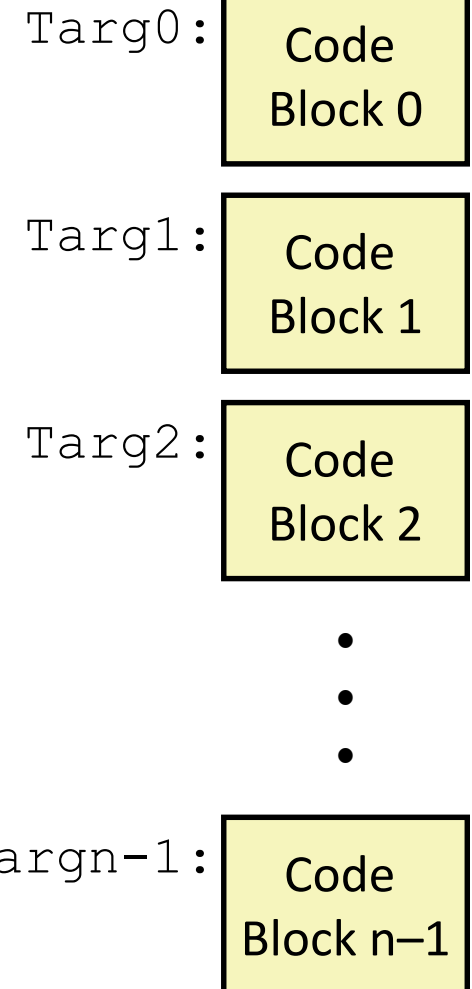
Approximate Translation

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

Jump Table



Jump Targets



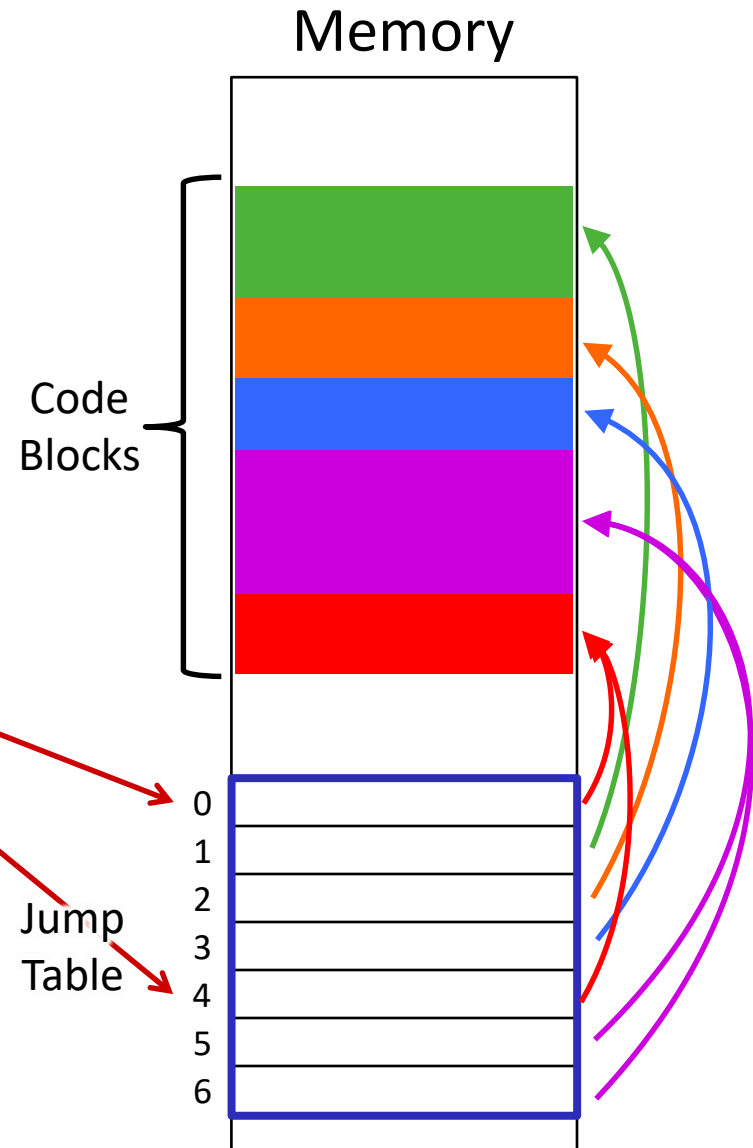
Jump Table Structure

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 $x \leq 6$:

```
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;
}

```

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

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

```

Take a look!

<https://godbolt.org/z/aY24el>

jump above – unsigned > catches negative default cases

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**



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

```

Assembly Setup Explanation

❖ Table Structure

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

❖ **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 $0 \leq x \leq 6$

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
```

BONUS SLIDES

Slides that expand on the simple switch code in assembly. **These slides expand on material covered today, so while you don't need to read these, the information is "fair game."**

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	1 st argument (x)
%rsi	2 nd argument (y)
%rdx	3 rd 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)

```

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

```

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

Code Blocks (rest)

```
switch (x) {  
    . . .  
    case 5: // .L7  
    case 6: // .L7  
        w -= z;  
        break;  
    default: // .L8  
        w = 2;  
}
```

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

```
.L7: # Case 5,6:  
    movl    $1, %eax # w = 1  
    subq   %rdx, %rax # w -= z  
    ret  
.L8: # Default:  
    movl    $2, %eax # 2  
    ret
```