

x86-64 Programming IV

CSE 351 Autumn 2024

Instructor:

Ruth Anderson

Teaching Assistants:

Alexandra Michael

Connie Chen

Chloe Fong

Chendur Jayavelu

Joshua Tan

Nikolas McNamee

Nahush Shrivatsa

Naama Amiel

Neela Kausik

Renee Ruan

Rubee Zhao

Samantha Dreussi

Sean Siddens

Waleed Yagoub



Relevant Course

- ❖ HW8 due TONIGHT, Wednesday (10/16) @ 11:59 pm
- ❖ HW9 due Friday (10/18) @ 11:59 pm
- ❖ Lab 2 (x86-64) due next Friday (10/25)
 - GDB [Demo Video on Ed!](#)
 - Learn to read x86-64 assembly and use GDB
 - See GDB Tutorial and Phase 1 walkthrough in Section 4!

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 {...}`

Conditional Example using set

		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;
}

```

Handwritten notes: %rdi points to x, %rsi points to y

```

cmpq $3, %rdi
setl %al  ← set if x < 3

cmpq %rsi, %rdi
sete %bl  ← set x == y

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

Labels & Jumps in C?

C

```
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

Do not use `goto` in your C code!

- ❖ 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 `gotos` 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:

x86 Control Flow

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

SUM is in %rax

Compiling Loops

C/Java code: LoopTest

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

Assembly code:

sum

```
loopTop:    testq  %rax, %rax    ! loopTest if sum == 0  
            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 *jump if rax==0*
 <loop body code>
 jmp loopTop *Sum--*
loopDone:

Do-while Loop:

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

x86-64:

loopTop:
 <loop body code>
 testq %rax, %rax
 jne loopTop *jump if rax!=0*
loopDone:

While Loop (ver. 2):

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

x86-64:

testq %rax, %rax *jump if rax==0*
 je loopDone
loopTop: <loop body code> *Sum--*
 testq %rax, %rax
 jne loopTop *jump if rax!=0*
loopDone:

sum = 2

times thru loop

sum is in %rax

Times thru loop

Jumps Taken

jmp & je

times thru loop

*jump taken
← jne*

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*

Polling Question

- ❖ The following is assembly code for a for-loop; identify the corresponding parts (Init, Test, Update)

■ i → %eax, x → %rdi, y → %esi

Line	
1	movl <u>0</u> , %eax
2	.L2: cmpl %esi, %eax
3	jge .L4
4	movslq %eax, %rdx
5	leaq (%rdi, %rdx, 4), %rcx
6	movl (%rcx), %edx
7	addl \$1, %edx
8	movl %edx, (%rcx)
9	addl \$1, %eax
10	jmp .L2
11	.L4:

line 1 line 2+3 line 9
 for (int i = 0 ; i < y ; i++)

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*

C code Jump Table Structure

Memory

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

JTab
address of
jump table

Targ0
Targ1
Targ2
•
•
•
Targn-1

addresses (8 bytes wide)

Jump Targets

Targ0:

Code
Block 0

Targ1:

Code
Block 1

Targ2:

Code
Block 2

•
•
•

Targn-1:

Code
Block n-1

Jump Table Structure

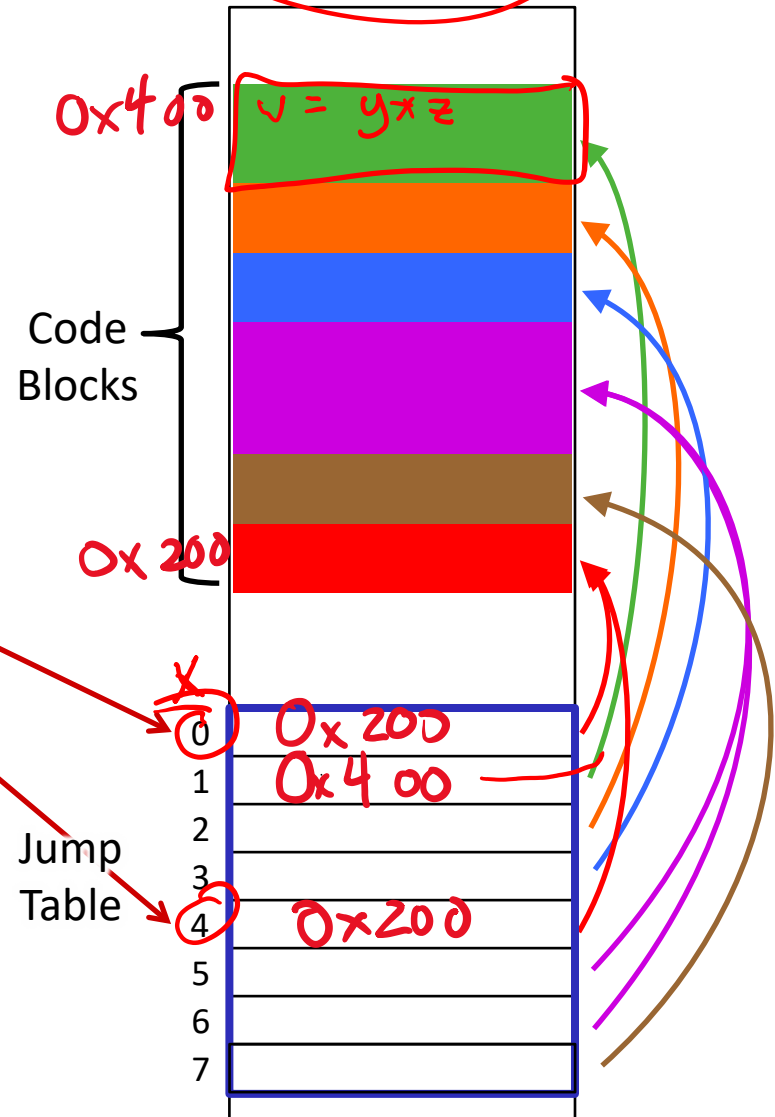
C code:

```
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 $x \leq 7$:

```
if (x <= 7)
    target = JTab[x];
    goto target;
else
    goto default;
```

Memory



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

Take a look!

<https://godbolt.org/z/Y9Kerb>

jump above – unsigned > catches negative default cases

$x > 7u$

If x is neg then when
interpreted as unsigned it is
large

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

$D + R_i * S$

addr of jump table

x

sizeof (word*)

following data is
a "quad word" = 8 bytes

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

Assembly Setup Explanation

❖ Table Structure

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

❖ **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 $0 \leq x \leq 7$

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
```

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