x86-64 Programming III

CSE 351 Summer 2024

Instructor:
Ellis Haker

Teaching Assistants:
Naama Amiel
Micah Chang
Shananda Dokka
Nikolas McNamee
Jiawei Huang

Administrivia

- Today
  - HW7 due (11:59pm)
  - **Lab2 out!**
    - Due 7/19
- Wednesday, 7/10
  - RD10 due (1pm)
  - HW8 due (11:59pm)
  - **Lab1b due (11:59pm)**
- Friday, 7/12
  - RD11 due (1pm)
  - HW10 due (11:59pm)
  - <u>**Quiz 1 due (11:59pm)**</u>

Review Question

What should go in the two blank lines?

%rdi	x
%rsi	y
%rax	result

```c
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
```
B)
```
cmpq %rsi, %rdi
jg .L4
```
~~**C)**~~
```
testq %rsi, %rdi
jle .L4
```
~~**D)**~~
```
testq %rsi, %rdi
jg .L4
```

cmp = subtract
test = and (bitwise)
cmp lets us check which is greater

cmpq rsi, rdi
= rdi − rsi
= x − y
jump to else if
x !> y → x <= y, x − y <= 0
↳ jle

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

Lecture Topics

- **Condition Codes**
- **Conditional and Unconditional Branches**
- Loops
- Switches

Recap: Condition Codes

- 4 1-byte registers that store information about the result of a previous instruction
    - **CF**: Carry (unsigned overflow)
    - **ZF**: Zero
    - **SF**: Sign (negative)
    - **OF**: Overflow (signed)
- **Implicitly** set by any arithmetic or logical instruction
- **Explicitly** set by two instructions:
    - `cmp_ src1, src2` sets flags based on the result of `src2-src1`
    - `test_ src1, src2 sets` flags based on the result of `src2&src1`

Recap: Set Instructions

don't need to learn these

- Sets lowest byte to 1 if the condition is met, 0 otherwise

takes on 1B register or memory address

comparison is against zero!

ex: set*e* = set if equal to 0

Instruction	Condition	Description
sete dst	ZF	Equal (to zero)
setne dst	~ZF	Not Equal (to 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 than (signed)
setle dst	(SF^OF)|ZF	Less or Equal (signed)
seta dst	~CF&~ZF	Above (unsigned ">")
setb dst	CF	Below (unsigned "<")

Recap: Reading Condition Codes

%rdi	x
%rsi	y
%rax	Return

- set* Instructions
  - Set a byte to 0 or 1 based on condition codes
  - Operand is byte register (e.g., %al) or a byte in memory
    - Does not alter remaining bytes in register
    - Typically use movzb_ (zero-extended mov) to fill in the rest of the register

```c
int gt(long x, long y)
{
    return x > y;
}
```

set flags based on rdi - rsi = x - y

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

set al (lowest B of rax) to 1 if x-y>0 → x>y

extend to 4B int

Recap: Jump Instructions

- Sets `%rip` to `target` if the condition is met
  - `jmp` is **unconditional** - always jumps
- Used to create if statements, loops, etc.

Reminder! rip = instruction pointer
stores address of
next instruction

Instruction	Condition	Description
`jmp` `target`	`1`	Unconditional
`je` `target`	`ZF`	Equal (to zero)
`jne` `target`	`~ZF`	Not Equal (to 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 than (signed)
`jle` `target`	`(SF^OF)|ZF`	Less or Equal (signed)
`ja` `target`	`~CF&~ZF`	Above (unsigned “>”)
`jb` `target`	`CF`	Below (unsigned “<”)

Choosing Instructions for Conditionals

- All arithmetic instructions set condition flags based on result of operation (op)
  - Jump and set are comparisons against 0
- Come in *pairs*
  - First instruction sets condition codes
  - Second instruction uses them

```
addq a, b
je:        b+a == 0
jne:       b+a != 0
jg:        b+a >  0
jl:        b+a <  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`	(nonnegative)	`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 < -U`

Choosing Instructions for Conditionals (pt 2)

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

		(op) s, d
je	"Equal"	d (op) s == 0
jne	"Not Equal"	d (op) s != 0
js	"Sign" (negative)	d (op) s < 0
jns	(nonnegative)	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 < -U

```
cmpq a, b
je:      b-a == 0
jne:     b-a != 0
jg:      b-a >  0
jl:      b-a <  0
```

```
testq a, b
je:      b&a == 0
jne:     b&a != 0
jg:      b&a >  0
jl:      b&a <  0
```

a&a = a

Use **test, a, a** to set the flags based on a single value!

```
testq a, a
je:      a == 0
jne:     a != 0
jg:      a >  0
jl:      a <  0
```

Labels

note: compiler-generated labels usually in the form .L##

- 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 assembly code
    - Associated with the next instruction found in the assembly code
    - Eventually, each ***use*** of the label will be replaced with something that indicates the address of the instruction that it is associated with

```
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
```

Putting it all Together (pt 1)

%rdi	x
%rsi	y
%rax	Return value

reminder: need opposite condition to jump to else case

```
long max(long x, long y)
{
    long max;
    if (x > y) {
        max = x;
    } else {
        max = y;
    }
    return max;
}
```

```
max:
    if x<y, jump to else
    movq %rdi, %rax
    jump to done
else:
    movq %rsi, %rax
done:
    ret
```

unconditional jump; otherwise we would execute both if and else code

Putting it all Together (pt 2)

%rdi	x
%rsi	y
%rax	Return value

y-x >= 0 → y >= x

```
long max(long x, long y)
{
    long max;
    if (x > y) {
        max = x;
    } else {
        max = y;
    }
    return max;
}
```

rsi - rdi = y-x

```
max:
        cmpq %rdi, %rsi    # jump if
        jge else           # y >= x
        movq %rdi, %rax
        jmp done
else:
        movq %rsi, %rax
done:
        ret
```

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 a means of transferring control
  - Closer to assembly programming style
  - Don't do this!!! Bad!!!

exists for historical reasons, mimics assembly

Expressing with Goto Code (pt 2)

The `continue` statement is often used when the part of the loop that follows is complicated, so that reversing a test and indenting another level would nest the program too deeply.

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:

Lecture Topics

- Condition Codes
- Conditional and Unconditional Branches
- **Loops**
- Switches

Compiling Loops

Example: while loop

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

%rax = sum

```
loopTop:
    testq %rax, %rax    # exit loop if
    je loopDone         # sum == 0
    <loop body>
    jmp loopTop         # restart loop
loopDone:
```

- Other loops compiled similarly
  - Examples on the next slide
- Most important to consider:
  - When should conditionals be evaluated (ex: *while* vs *do-while*)
  - How much jumping is involved?
    - Compiler may optimize to reduce jumping (as in the above example)

More While Loop Examples

if sum == 0 at the beginning, skip the loop

closer to C code, but less efficient

While Loop (other version)

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

```
   testq %rax, %rax   # skip loop if
   je loopDone        # sum == 0
loopTop:
   <loop body>
   testq %rax, %rax   # restart loop
   jne loopTop        # if sum != 0
loopDone:
```

this condition is the same as the C code

Do-While Loop

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

execute loop body at least once

```
loopTop:
     <loop body>         # execute first
     testq %rax, %rax    # restart loop
     jne loopTop         # if sum != 0
loopDone:
```

For Loops

```
for (init; test; update)
{
        body
}
```

==

```
init
while (test)
{
        body
        update
}
```

- Just turn it into a while loop!
- **Caveats**: `break` and `continue`
  - `break` is easy, just add another jump outside of the loop
  - For `continue`, need to introduce a new label for Update, and jump to that

```
<init>
loopTop:
    <body>
    <update>
    <test>
    <conditional jump to loopTop>
loopDone:
```

if loop has continue stmts, jump here

Lecture Topics

- Condition Codes
- Conditional and Unconditional Branches
- Loops
- **Switches**

Switch Statement Example

- Specify *cases* depending on the result of an expression
  - Multiple case labels
    - ex: 5 & 6
  - Fall through cases
    - ex: 2
  - Missing cases
    - ex: 4
  - Default case
    - Executed if the switch expression doesn't match any other case

could also use an if/else ladder, but this looks cleaner

```
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;
}
```

if x==1

if x==2

2 case will also execute case 3 code

args not listed

Jump Table Structure

- **Jump table** = array of addresses to the start of each case

Switch Form

```
switch (x) {
    case val_0:
        <lock 0>
    case val_1:
        <block 1>
    • • •
    case val_n-1:
        <block n-1>
}
```

Approximate goto form

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

Jump Table

Targ0
Targ1
• • •
Targn-1

case = index in jump table

Jump Targets

```
Targ0:
  <block 0>
```

```
Targ1:
  <block 1>
```

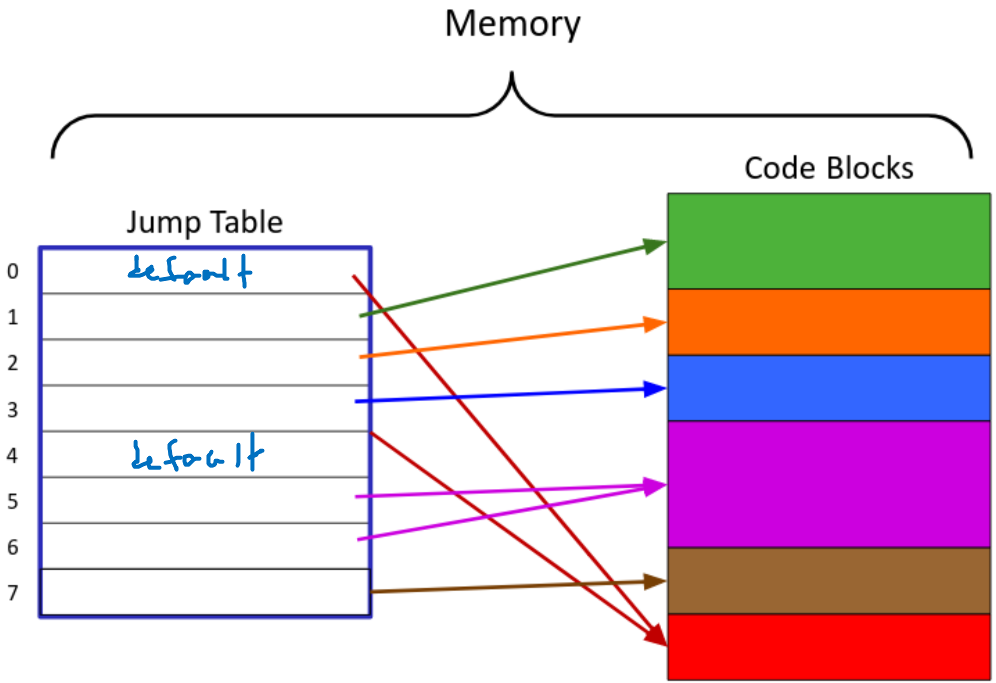
```
Targn-1:
  <block n-1>
```

Jump Table Structure (pt 2)

```
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>
}
```

unsigned, so that negative #s overflow to be >7

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



Switch Statement Example (pt 2)

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

if x(unsigned) >7, go to default

%rdi	x
%rsi	y
%rdx	z

Note: compiler chose not to initialize w

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

go to index "x" in jump table

ja = jump above. Unsigned >, catches negative cases

Switch Statement Example (pt 3)

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

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

Indirect jump

%rdi	x
%rsi	y
%rdx	z

Switch Statement Assembly Explanation

- Table Structure
  - Each element stores an address, 8B each
  - Label `.L4` marks start of table
- **Direct jump**: `jmp .L9`
  - Jump target denoted by label `.L9`
- **Indirect jump**: `jmp *.L4(,%rdi,8)`
  - Uses memory addressing computation
  - `.L4` gets the address that label `.L4` refers to
  - Scaled by 8, since each table element is 8B
  - Set `%rsp` to the label stored at address `L4+%rdi*8`
    - `*` means "get the data at that address" (kind of like a C dereference)

w/out *, it would set %rsp to &table[x]. w/ *, sets it to table[x]

label for code

label for table

```
.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
```

"quad" = 8B in x86_64

Summary

- Control flow in x86 is determined by **condition codes** (CF, ZF, SF, OF)
    - Set flags implicitly with arithmetic/logical instructions, or explicitly with **`cmp`** and **`test`**
    - **`Set`** (`set*`) instructions set a given 1-byte memory location to 1 or 0 depending on condition
    - **Conditional jump** (`j*`) instructions set `%rsp` point to a given instruction if condition is true
        - **Unconditional jump** (`jmp`) always sets `%rsp`
    - Most control flow constructs (e.g., if-else, for-loop, while-loop) can be implemented in assembly using combinations of conditional and unconditional jumps
        - **Indirect jumps** and **jump tables** are used to implement switches