x86-64 Programming III

CSE 351 Summer 2020

Instructor:

Porter Jones

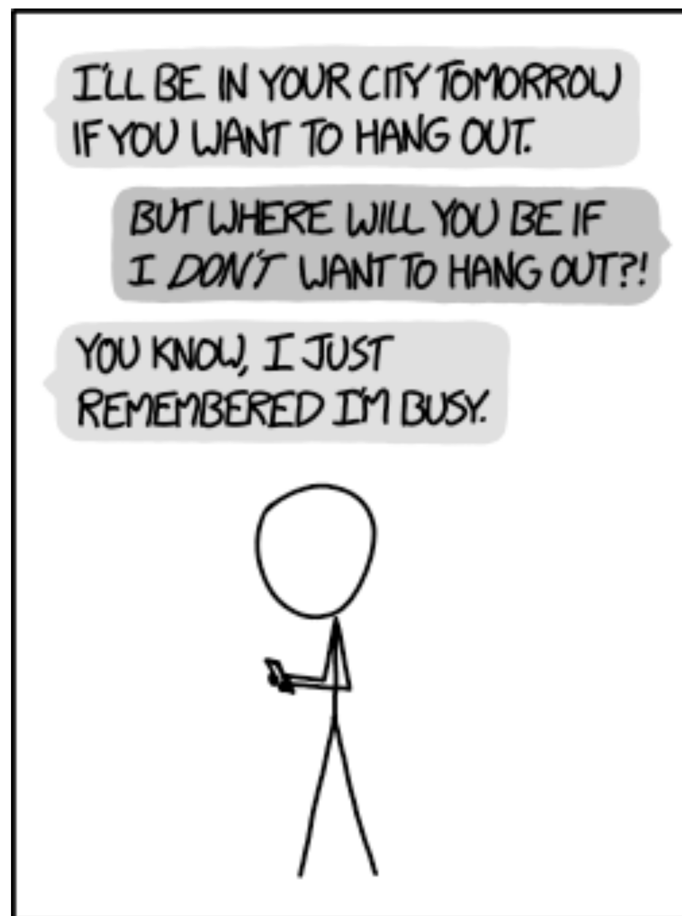
Teaching Assistants:

Amy Xu

Callum Walker

Sam Wolfson

Tim Mandzyuk



WHY I TRY NOT TO BE PEDANTIC ABOUT CONDITIONALS.

http://xkcd.com/1652/

Administrivia

- Questions doc: https://tinyurl.com/CSE351-7-13
- Lab 1b due ***tonight*** at 11:59pm (7/13)
  - Submit `aisle_manager.c`, `store_client.c`, and `lab1Breflect.txt`
  - Can still use late days until 7/15
- Unit Summary 1 due Friday (7/17) – 11:59pm
  - Can still use late days until 7/20
- Mid-quarter Survey due Friday (7/17) – 11:59pm
  - Submit via Canvas!
- hw8, hw9, hw10 now due Monday (7/20) – 10:30am
  - hw11 also due Monday (7/20)
  - See course schedule for original/suggested deadlines

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
```

First instruction → `(op) s, d`

Second instruction options ↓

		`(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`
`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`

When jump occurs

Choosing instructions for conditionals

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

first instruction options

		`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`

second instruction options

when jump occurs

```
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

We want to jump when x >= 3

cmpq 3, %rdi

jge T2 ← where to jump

		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      if branch
  ret
T2: # !(x < 3):      else branch
  movq $2, %rax
  ret
```

Polling Question [Asm III - a]

we want to jump when x <= y

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

cmpq + jle!

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

Vote at http://pollev.com/pbjones

A. `cmpq %rsi, %rdi` (y, x)
   `jle .L4`

B. `cmpq %rsi, %rdi`
   `jg .L4`

C. `testq %rsi, %rdi`
   `jle .L4`

D. `testq %rsi, %rdi`
   `jg .L4`

E. We're lost...

```
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

```
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
```

Handwritten annotations: (1) %al = (x < 3); (2) %bl = (x == y); (3) jump if (%al & %bl) == 0

❖ https://godbolt.org/z/GNxpqv

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:
```

(handwritten annotations: "exit", "test again")

- 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

C/Java code:

```
while ( Test ) {
    Body
}
```

Goto version:

```
Loop: if ( !Test )  goto Exit;
      Body
      goto Loop;
Exit:
```

- What are the Goto versions of the following?
  - Do…while: Test and Body
  - For loop: Init, Test, Update, and Body

do{
body
}while(test);

for(init; test; update){
body
}

Do while

Loop: Body
if(test) goto Loop;

For Loop

Init
Loop: if(!test) goto exit;
Body
update
goto Loop;
Exit:

Compiling Loops

all jmp instructions update program counter (%rip)

While Loop:

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

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

exit; test again

Do-while Loop:

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

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

loop again; fall through/exit

While Loop (ver. 2):

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

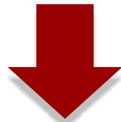
x86-64:
```
            testq  %rax, %rax
            je     loopDone
loopTop:
            <loop body code>
            testq  %rax, %rax
            jne    loopTop
loopDone:
```

initial test; loop again; fall through/exit

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*

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;
}
```

if x == 2

exit switch

Switch Statement Example

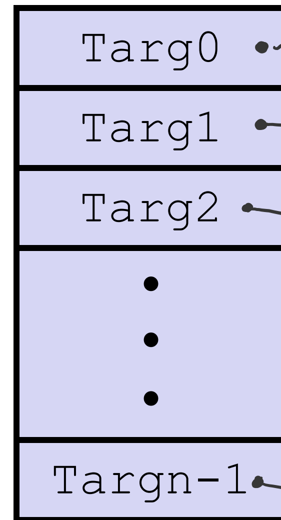
- Multiple case labels
  - Here: 5 & 6
- Fall through cases
  - Here: 2
- Missing cases
  - Here: 4
- Implemented with:
  - *Jump table*
  - *Indirect jump instruction*

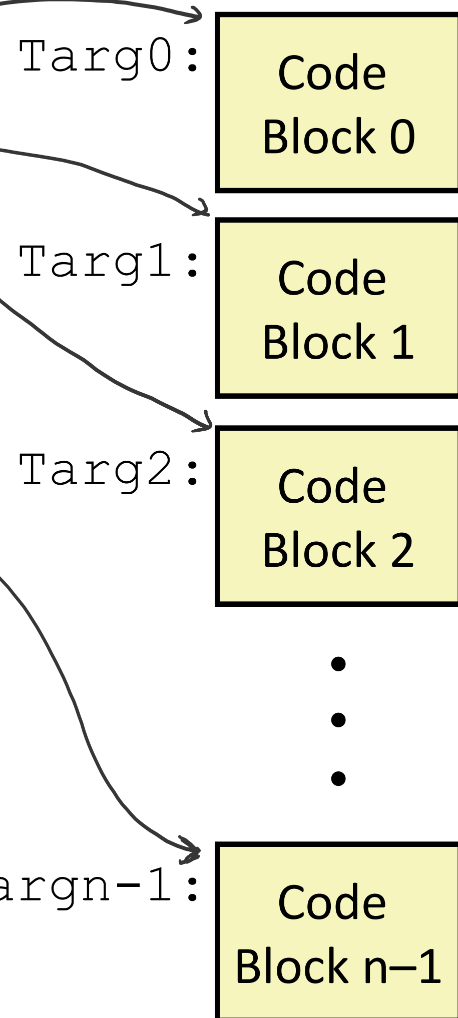
Jump Table Structure

in 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

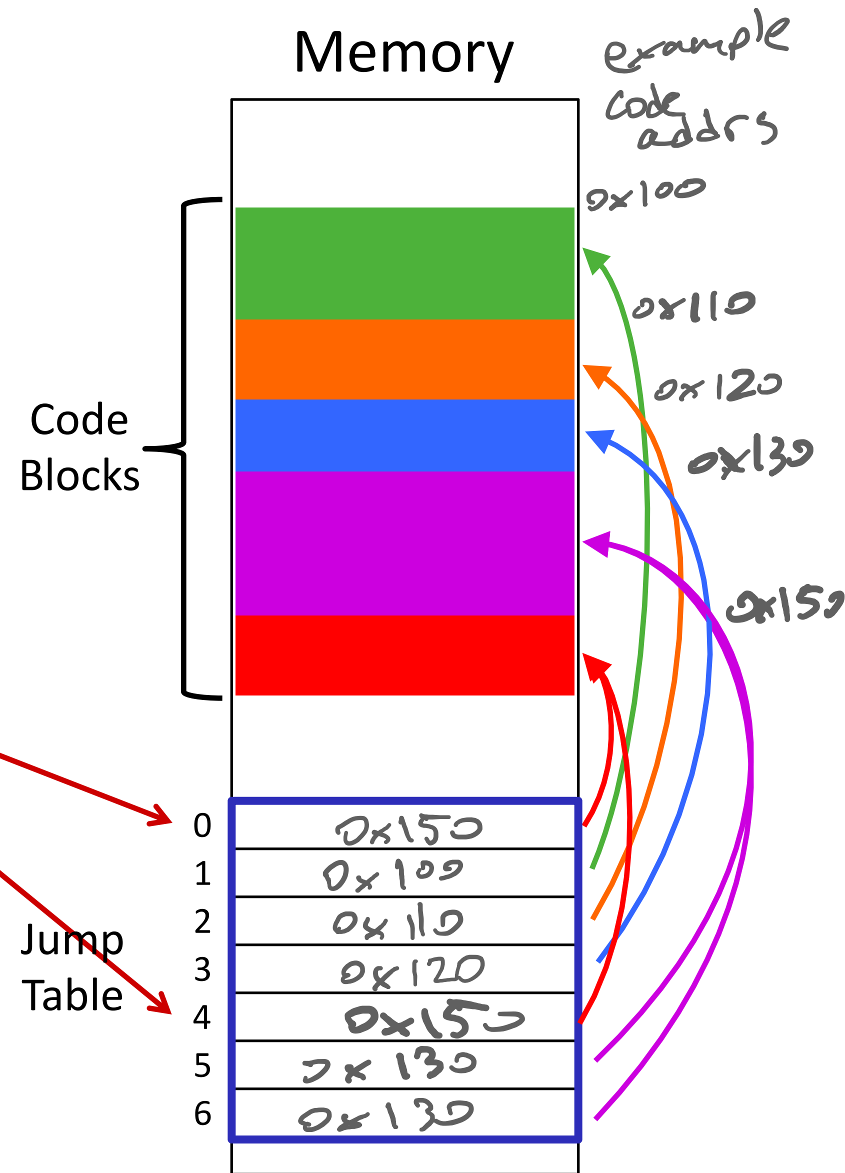
Jump Table Structure

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 ≤ 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;
}
```

rdi rsi rdx

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

```
switch_eg:
  movq    %rdx, %rcx
  cmpq    $6, %rdi          # x:6
  ja      .L8               # default
  jmp     *.L4(,%rdi,8)     # jump table
```

unrelated to jump

jump to default if x>6

Note compiler chose to not initialize w

Take a look!

https://godbolt.org/z/aY24el

jump above – unsigned > catches negative default cases

negative numbers are large unsigned values

Switch Statement Example

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

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
```

```
switch_eg:
    movq    %rdx, %rcx
    cmpq    $6, %rdi          # x:6
    ja      .L8               # default
    jmp     *.L4(,%rdi,8)     # jump table
```

Indirect jump

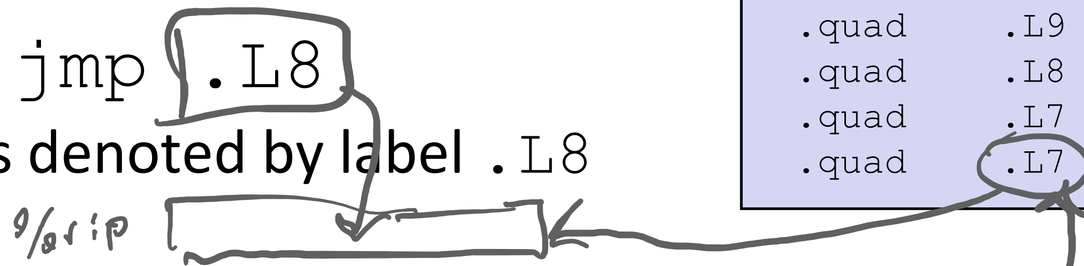
Assembly Setup Explanation

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

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
```

- Direct jump: `jmp .L8`
  - Jump target is denoted by label `.L8`



- Indirect jump: `jmp *.L4(,%rdi,8)`

  Mem[D+ Reg[Ri]*S]

  - 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 $0 \le x \le 6$

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)

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

```
switch(x) {
  case 1:      // .L3
    w = y*z;
    break;
   . . .
}
```

```
.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)

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

```
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
```

Code Blocks (rest)

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

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

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

GDB Demo

- The `movz` and `movs` examples on a real machine!
  - `movzbq %al, %rbx`
  - `movsbl (%rax), %ebx`

- You will need to use GDB to get through Lab 2
  - Useful debugger in this class and beyond!

- Pay attention to:
  - Setting breakpoints (`break`)
  - Stepping through code (`step`/`next` and `stepi`/`nexti`)
  - Printing out expressions (`print` – works with regs & vars)
  - Examining **memory** (`x`)