

Lecture 5: Arrays, Operations and Jumps in Assembly

CS 105

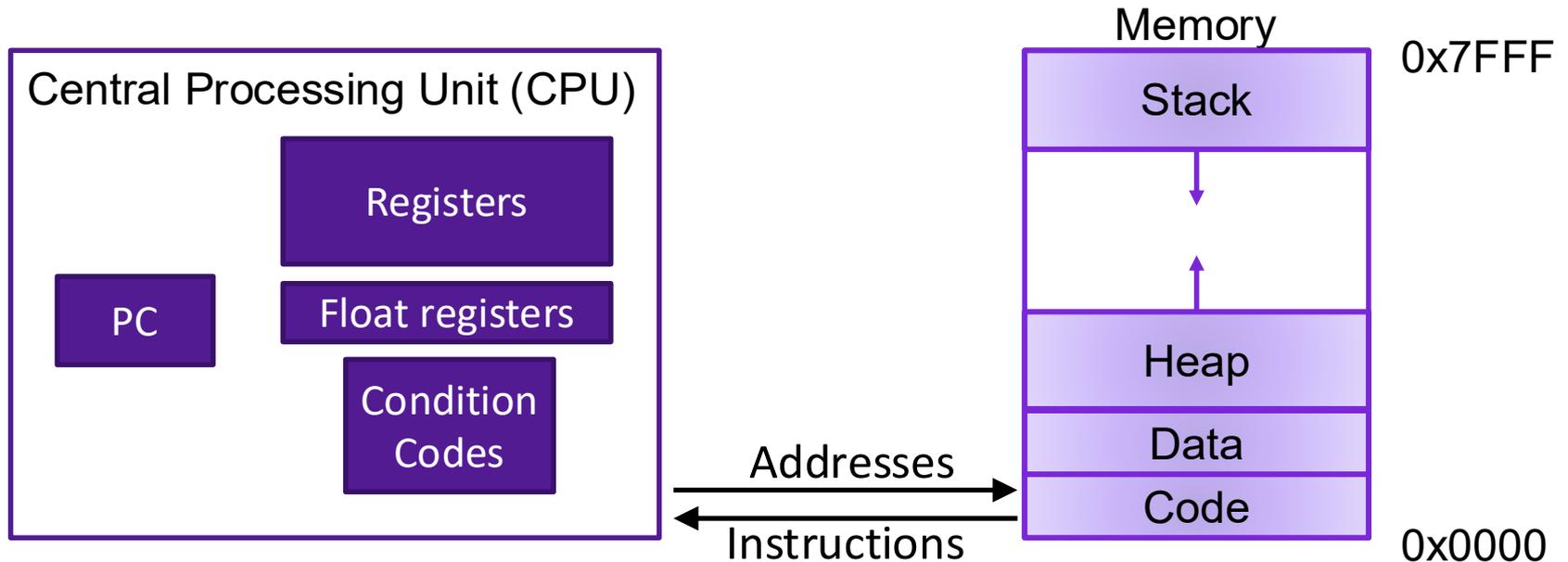
Spring 2026

Logistical Note

- Check In 1 in lab next week (Feb 18 or 19)
- 90 minutes
- 3 questions

- 1 page of written notes

Review: Assembly/Machine Code View



Programmer-Visible State

- ▶ PC: Program counter (%rip)
- ▶ Register file: 16 Registers
- ▶ Float registers
- ▶ Condition codes

Memory

- ▶ Byte addressable array
- ▶ Code and user data
- ▶ Stack to support procedures

Review: X86-64 Integer Registers

<code>%rax</code> (function result)	<code>%r8</code> (fifth argument)
<code>%rbx</code>	<code>%r9</code> (sixth argument)
<code>%rcx</code> (fourth argument)	<code>%r10</code>
<code>%rdx</code> (third argument)	<code>%r11</code>
<code>%rsi</code> (second argument)	<code>%r12</code>
<code>%rdi</code> (first argument)	<code>%r13</code>
<code>%rsp</code> (stack pointer)	<code>%r14</code>
<code>%rbp</code>	<code>%r15</code>

Review: Assembly Operations

- Transfer data between memory and register
 - Load data from memory into register
 - Store register data into memory
- Perform arithmetic function on register or memory data
- Transfer control
 - Conditional branches
 - Unconditional jumps to/from procedures

Review: Operand Forms

- Immediate:

- Syntax: \$c Ex: \$47 Val: c C Equiv: 47

- Register:

- Syntax: r Ex: %rbp Val: Reg[r] C Equiv: x

- Memory (Absolute):

- Syntax: addr Ex: 0x4050 Val: Mem[addr] C Equiv:
*0x60201a

- Memory (Indirect):

- Syntax: (r) Ex: (%rsp) Val: Mem[Reg[r]] C Equiv: *x

Review: Data Movement Instructions

- MOV source, dest

- movb

- movw

- movl

- movq

Move data source->dest

Move 1 byte

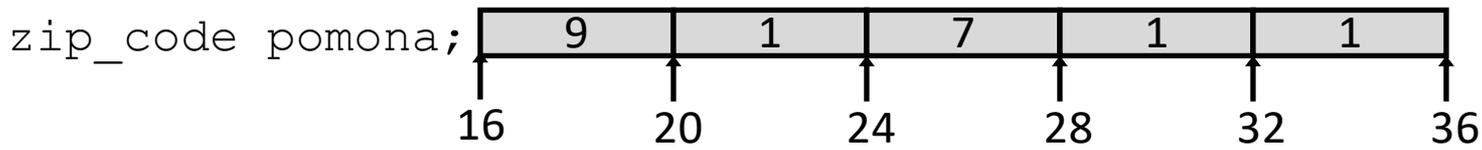
Move 2 bytes

Move 4 bytes

Move 8 bytes

Register	Use(s)
%rdi	zipcode
%rax	return val

Example 1: Array Accessing



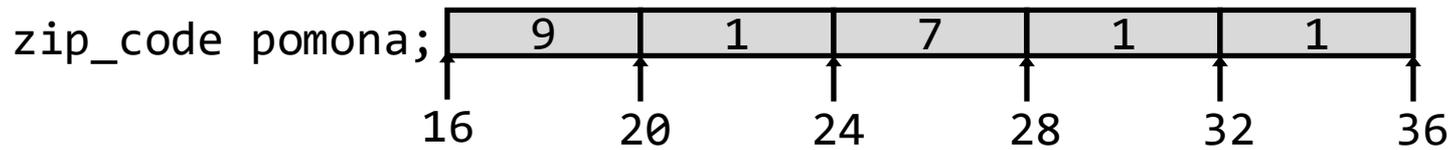
```
int get_first_digit(int* zipcode){
    return zipcode[0];
}
```

```
movl (%rdi), %eax # ret = z[0]
```

- Register %rdi contains starting address of array zipcode
- Desired digit at %rdi
- Use indirect memory reference (%rdi) or raw address for absolute memory reference 0x14

Register	Use(s)
%rdi	zipcode
%rsi	digit
%rax	return val

Example 2: Array Accessing



```
int get_digit(int* zipcode, int digit){  
    return zipcode[digit];  
}
```

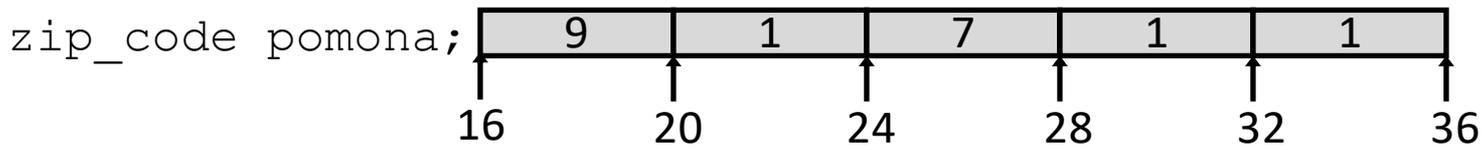
???

Operand Forms

- Immediate:
 - Syntax: \$c Ex: \$47 Val: c C Equiv: 47
- Register:
 - Syntax: r Ex: %rbp Val: Reg[r] C Equiv: x
- Memory (Absolute):
 - Syntax: addr Ex: 0x4050 Val: Mem[addr] C Equiv: *0x60201a
- Memory (Indirect):
 - Syntax: (r) Ex: (%rsp) Val: Mem[Reg[r]] C Equiv: *x
- Memory (Base+displacement):
 - Syntax: c(r) Ex: 12(%rsp) Val: Mem[Reg[r]+c] C Equiv: *(x+12)
- Memory (Scaled indexed):
 - Syntax: (r1,r2,s) Ex: (%rdx,%rsi,4) Val: Mem[Reg[r1]+Reg[r2]*s] C: r1[r2]
- Memory (Scaled indexed w/ displacement):
 - Syntax: c(r1,r2,s) Ex: 8(%rdx,%rsi,4) Val: Mem[Reg[r1]+Reg[r2]*s+c] C: (r1+8)[r2]

Register	Use(s)
%rdi	z
%rsi	digit
%rax	return val

Example 2: Array Accessing



```
int get_digit(int* zipcode, int digit){
    return z[digit];
}
```

```
movl (%rdi,%rsi,4), %eax # ret = z[digit]
```

- Register %rdi contains starting address of array zipcode
- Register %rsi contains array index digit
- Desired digit at %rdi + 4*%rsi
- Use memory reference (%rdi,%rsi,4)

Exercise 1: Operands

Register	Value
%rax	0x108
%rcx	0x01
%rdx	0x03

Memory Address	Value
0x108	0xFF
0x109	0x47
0x10a	0x13
0x10b	0xE1
0x10c	0xAB
0x10d	0x2F
0x10e	0x07

- What are the values of the following operands (assuming register and memory state shown above)?
 1. (%rax,%rcx,4)
 2. (%rax,%rdx,2)
 3. 1(%rax,%rcx,4)

ARITHMETIC IN ASSEMBLY

Some Arithmetic Operations

- Two Operand Instructions:

Format

`andq` *Src, Dest*

`orq` *Src, Dest*

`xorq` *Src, Dest*

`shlq` *Src, Dest*

`shrq` *Src, Dest*

`sarq` *Src, Dest*

`addq` *Src, Dest*

`subq` *Src, Dest*

`imulq` *Src, Dest*

Computation

$\text{Dest} = \text{Dest} \& \text{Src}$

$\text{Dest} = \text{Dest} | \text{Src}$

$\text{Dest} = \text{Dest} \wedge \text{Src}$

$\text{Dest} = \text{Dest} \ll \text{Src}$

$\text{Dest} = \text{Dest} \gg \text{Src}$

$\text{Dest} = \text{Dest} \gg \text{Src}$

$\text{Dest} = \text{Dest} + \text{Src}$

$\text{Dest} = \text{Dest} - \text{Src}$

$\text{Dest} = \text{Dest} * \text{Src}$

Also called `salq`

Logical

Arithmetic

Suffixes

<code>char</code>	<code>b</code>	1
<code>short</code>	<code>w</code>	2
<code>int</code>	<code>l</code>	4
<code>long</code>	<code>q</code>	8
<code>pointer</code>	<code>q</code>	8

Some Arithmetic Operations

- One Operand Instructions

notq *Dest* $Dest = \sim Dest$

incq *Dest* $Dest = Dest + 1$

decq *Dest* $Dest = Dest - 1$

negq *Dest* $Dest = - Dest$

Suffixes

char	b	1
short	w	2
int	l	4
long	q	8
pointer	q	8

Example: Translating Assembly

```
arith:
  orq    %rsi, %rdi
  sarq   $3, %rdi
  notq   %rdi
  movq   %rdx, %rax
  subq   %rdi, %rax
  ret
```

```
long arith(long x, long y, long z){
  x = x | y;
  x = x >> 3;
  x = ~x;

  long ret = z - x;
  return ret;
}
```

Interesting Instructions

- **sarq**: arithmetic right shift

Register	Use(s)
%rdi	Argument x
%rsi	Argument y
%rdx	Argument z
%rax	return value

lea Instruction

Scaled Memory Operands

```
movq (%rdi,%rsi,8), %rax
```

```
void ex1(long* xp, long yp) {  
    long* p = xp + yp;  
    long ret = *p;  
}
```

```
long m12(long x) {  
    return x*12;  
}
```

leaq *Source, Dest*

```
leaq (%rdi,%rsi,8), %rax
```

```
void ex3(long xp, long yp) {  
    long ret = xp + 8*yp;  
}
```

- pointer arithmetic
 - E.g., $p = x + i$;
- arithmetic
 - expressions $x + k*y$ ($k=1, 2, 4, 8$)

Used by compiler to optimize:

```
leaq (%rdi,%rdi,2), %rax # ret <- x+x*2  
salq $2, %rax           # return ret<<2
```

CONDITIONAL JUMPS

Jumps

- A jump instruction can cause the execution to switch to a completely new position in the program (updates the program counter)
 - `jmp Label`
 - `jmp *Operand`

```
.L0:  
  movq    $0, %rax  
  jmp     .L1  
  movq    (%rax), %rdx  
.L1:  
  movq    %rcx, %rax
```

```
jmp *%rdi
```

Conditional Jumps

- jX instructions
 - Jump to different part of code if condition is true

jX	Description
jmp	Unconditional
je	Equal to Zero
jne	Not Equal to Zero
js	Negative
jns	Nonnegative
jl	Less (Signed)
jle	Less or Equal (Signed)
jg	Greater (Signed)
jge	Greater or Equal (Signed)

What condition are we evaluating?

Conditional Jumps

- Whether or not we jump depends on how the output of the last arithmetic operation compares to zero

```
movq $47, %rax  
subq $13, %rax  
jg .L2
```

jump

```
movq $47, %rax  
subq $13, %rax  
je .L2
```

no jump

- Not set by `leaq` instruction
- Unless there's an explicit conditional evaluation more recently

Condition Evaluations

- `cmp a, b` like computing $b - a$ without setting destination
- `test a, b` like computing $a \& b$ without setting destination

Exercise 2: Conditional Jumps

- Consider each of the following segments of assembly code, and indicate whether or not the jump will occur. In all cases, assume that `%rdi` contains the value 47 and `%rsi` contains the value 13

1. `addq %rdi, %rsi`
`je .L0`
2. `subq %rdi, %rsi`
`jge .L0`
3. `cmpq %rdi, %rsi`
`j1 .L0`
4. `testq %rdi, %rdi`
`jne .L0`

Branches and Jumps

- ▶ Processor state (partial)
 - ▶ Temporary data (`%rax`, ...)
 - ▶ Location of runtime stack (`%rsp`)
 - ▶ Location of next instruction to execute (`%rip`)
 - ▶ Status of recent tests (`CF`, `ZF`, `SF`, `OF`)

Registers

<code>%rax</code> (return val)	<code>%r8</code> (5 th arg)
<code>%rbx</code>	<code>%r9</code> (6 th arg)
<code>%rcx</code> (4 th arg)	<code>%r10</code>
<code>%rdx</code> (3rd arg)	<code>%r11</code>
<code>%rsi</code> (2 nd arg)	<code>%r12</code>
<code>%rdi</code> (1 st arg)	<code>%r13</code>
<code>%rsp</code> (stack ptr)	<code>%r14</code>
<code>%rbp</code>	<code>%r15</code>

`%rip` Program counter

`CF` `ZF` `SF` `OF` Condition codes

Condition Codes

- Single bit registers
 - **ZF** Zero Flag
 - **PF** Parity Flag
 - **SF** Sign Flag (for signed)
 - **OF** Overflow Flag (for signed)
 - **CF** Carry Flag (for unsigned)
- Implicitly set (as a side effect) by arithmetic operations
- Explicitly set by **cmp** and **test**
- Not set by **leaq** instruction

Example Condition Codes: `compare`

- Instruction `cmp` explicitly sets condition codes
- `cmpq a, b` like computing `b-a` without setting destination
 - **ZF set** if $(b-a) == 0$
 - **PF set** if $(b-a) \% 2 == 1$
 - **SF set** if $(b-a) < 0$ (as signed)
 - **OF set** if two's-complement (signed) overflow
 - **CF set** if carry out from most significant bit (used for unsigned comparisons)

Jumping

- jX instructions
 - Jump to different part of code if condition is true

jX	Condition	Description
jmp	1	Unconditional
je	ZF	Equal / Zero
jne	~ZF	Not Equal / Not Zero
js	SF	Negative
jns	~SF	Nonnegative
jl	(SF^OF)	Less (Signed)
jle	(SF^OF) ZF	Less or Equal (Signed)
jg	~(SF^OF) & ~ZF	Greater (Signed)
jge	~(SF^OF)	Greater or Equal (Signed)
jb	CF	Below (unsigned)
jbe	CF ZF	Below or Equal (Signed)
ja	~ZF & ~CF	Above (unsigned)
jae	~CF	Above or Equal (Signed)