

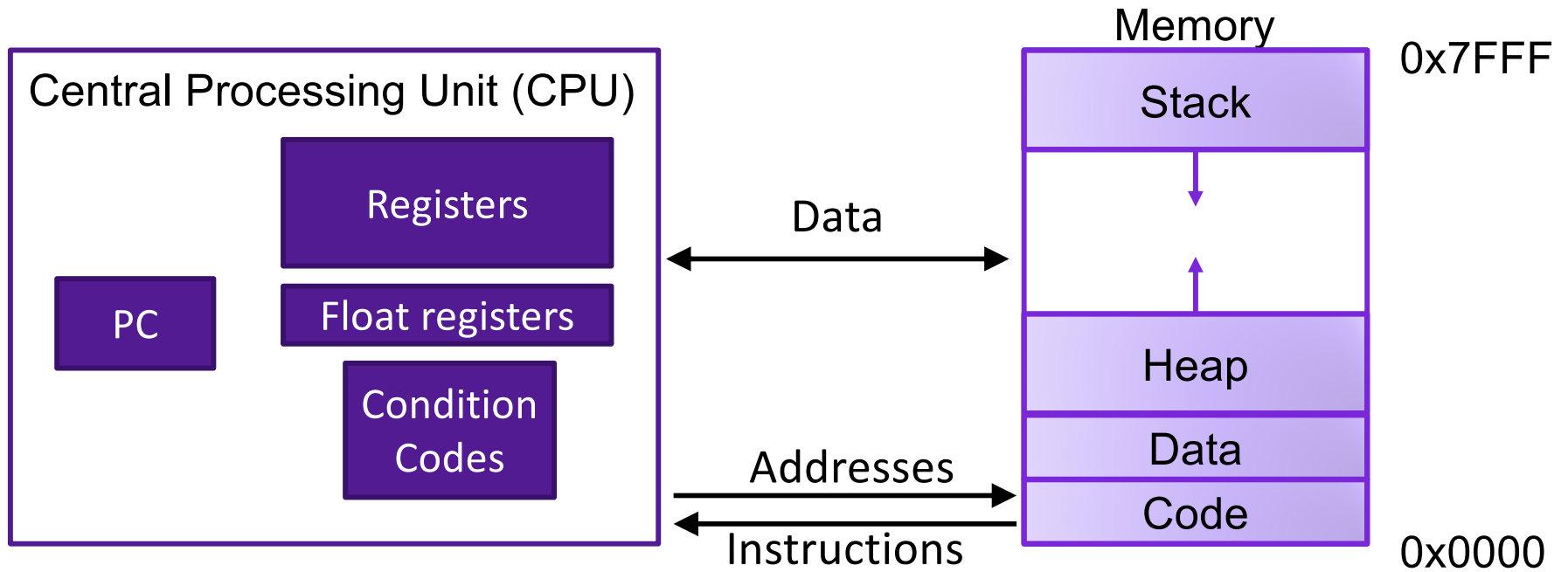


Lecture 7: Control Flow in Assembly

CS 105

Spring 2024

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: Conditional Jumps

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

jX	Description
jmp	Unconditional
je	Equal / Zero
jne	Not Equal / Not Zero
jl	Less (Signed)
jle	Less or Equal (Signed)
jg	Greater (Signed)
jge	Greater or Equal (Signed)

- Whether or not we jump depends on how the output of the last operation compares to zero
- Operation includes arithmetic, cmp, test
- Not set by **lea** instruction

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

Review: 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

Review: Implementing Conditional Jumps

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

jX	Condition	Description
jmp	1	Unconditional
je	ZF	Equal / Zero
jne	\sim ZF	Not Equal / Not Zero
jl	$(SF \wedge OF)$	Less (Signed)
jle	$(SF \wedge OF) \vee ZF$	Less or Equal (Signed)
jg	$\sim(SF \wedge OF) \ \& \ \sim ZF$	Greater (Signed)
jge	$\sim(SF \wedge OF)$	Greater or Equal (Signed)

Conditional Branching

```
long absdiff(long x, long y) {
    long result;

    if (x > y) {
        result = x-y;
    } else {
        result = y-x;
    }

    return result;
}
```

```
absdiff:
    cmpq    %rsi, %rdi
    jle    .L4
    movq    %rdi, %rax
    subq    %rsi, %rax
    ret

.L4:
    # x-y <= 0
    movq    %rsi, %rax
    subq    %rdi, %rax
    ret
```

Register	Use
%rdi	x
%rsi	y
%rax	result

Exercise 4: Conditionals

```
test:
    leaq (%rdi, %rsi), %rax
    addq %rdx, %rax
    cmpq $47, %rax
    jne .L2
    movq %rdi, %rax
    jmp .L4
.L2:
    cmpq $47, %rax
    jle .L3
    movq %rsi, %rax
    jmp .L4
.L3:
    movq %rdx, %rax
.L4:
    rep; ret
```

```
long test(long x, long y, long z){

    long val = _____;

    if (_____);

        _____;

} else if (_____);

        _____;

} else {

        _____;

}

    return val;

}
```

Reg	Use
%rdi	x
%rsi	y
%rdx	z
%rax	result

Loops

- All use conditions and jumps
 - do-while
 - while
 - for

Register	Use(s)
%rdi	Argument x
%rax	result

Do-while Loops

```
long bitcount(unsigned long x) {
    long result = 0;
    do {
        result += x & 0x1;
        x >>= 1;
    } while (x != 0);
    return result;
}
```

```
long bitcount(unsigned long x) {
    long result = 0;
loop:
    result += x & 0x1;
    x >>= 1;
    if(x != 0) goto loop;
    return result;
}
```

```
    movq    $0, %rax    # result = 0
.L2:      # loop:
    movq    %rdi, %rdx
    andq    $1, %rdx    # t = x & 0x1
    addq    %rdx, %rax  # result += t
    shrq    %rdi, $1    # x >>= 1
    jne     .L2         # if (x) goto loop
    rep; ret
```

Register	Use(s)
%rdi	Argument x
%rax	result

While Loops

```
long bitcount(unsigned long x) {  
    long result = 0;  
    while (x != 0) {  
        result += x & 0x1;  
        x >>= 1;  
    }  
    return result;  
}
```

```
movq    $0, %rax  
jmp     .L2  
  
.L3:  
movq    %rdi, %rdx  
andq    $1, %rdx  
addq    %rdx, %rax  
shrq    %rdi, $1  
  
.L2:  
testq   %rdi, %rdi  
jne     .L3  
rep; ret
```

```
movq    $0, %rax  
  
.L1:  
test    %rdi, %rdi  
je      .L2  
movq    %rdi, %rdx  
andq    $1, %rdx  
addq    %rdx, %rax  
shrq    %rdi, $1  
jmp     .L1  
  
.L2:  
rep; ret
```

Reg	Use(s)
%rdi	Argument <code>val</code>
%rdx	Local <code>i</code>
%rax	Local <code>ret</code>

Exercise: Loops

```
loop:
    movq $0, %rax
    movq $0, %rdx
    jmp L1
L0:
    addq %rdx, %rax
    incq %rdx
L1:
    cmp %rdi, %rdx
    jl L0
    ret
```

```
long loop(long val) {
    long ret = _____;
    long i   = _____;

    while(_____) {

        ret = _____;
        i   = _____;

    }

    return ret;
}
```

Register	Use(s)
%rdi	Argument x
%rax	result

For loops

```
for (Init; Cond; Incr) {  
    Body  
}
```



```
Init;  
while (Cond) {  
    Body;  
    Incr;  
}
```

Initial test can often be optimized away:

```
for (j = 0; j < 99; j++)
```

```
long bitcount(unsigned long x) {  
    long result;  
    for (result = 0; x != 0; x >>= 1)  
        result += x & 0x1;  
    return result;  
}
```



```
.L1:    movq    $0, %rax  
  
        test   %rdi, %rdi  
        je     .L2  
        movq   %rdi, %rdx  
        andq   $1, %rdx  
        addq   %rdx, %rax  
        shrq   %rdi, $1  
        testq  %rdi, %rdi  
        jmp    .L1  
  
.L2:  
        rep   ret
```

Variable	Register
z	%rdi
sum	%rax
i	%rsi

Exercise : Array Loop

```
array_loop:
    movl    $0, %esi
    xorl    %eax, %eax
    jmp     L2
L1:
    addl    (%rdi,%rsi,4), %eax
    incq    %rsi
L2:
    cmpq    $5, %rsi
    jl     L1
    retq
```

```
int array_loop(int * z) {
    int sum = _____;

    for(int i = ____; i < ____; ____ ) {
        sum = _____;
    }
    return _____;
}
```