CS107 Lecture 13

Assembly: Control Flow

reading:

B&O 3.6

Slides by Jerry Cain and Lisa Yan, who leveraged prior work by Nick Troccoli, Julie Zelenski, Marty Stepp, Cynthia Lee, Chris Gregg, and others.

Reflections + Warm-up

Reflections: Share your CS107 midterm experience with your neighbor:

- What is one concept/skill you are comfortable with?
- What is one concept/skill you can work on for the final exam?
- Examples: logical operations, pointer arithmetic, generics, understanding the spec, studying an assignment, gdb, time management, …

Warm-up: What’s the difference between **`lea`** and **`mov`**?

```
movq -0x8(%rdi,%rsi,8), %rax     leaq -0x8(%rdi,%rsi,8), %rax
```

🤔

Warm-up

What's the difference between **lea** and **mov**?

%rdi 0x7fe850

%rsi 3

0x7fe860	0x1234
0x7fe858	0xCDEF
0x7fe850	0xABCD

```
movq -0x8(%rdi,%rsi,8), %rax
```

%rax 0x1234

```
long last_elt(long arr[], long nelems)
{
    return arr[nelems - 1];
}
```

```
leaq -0x8(%rdi,%rsi,8), %rax
```

%rax 0x7fe860

```
long *last_ptr(long arr[], long nelems)
{
    return arr + (nelems - 1);
}
```

lea makes use of indirect addressing to perform arithmetic with fewer instructions.

Register Responsibilities

Review

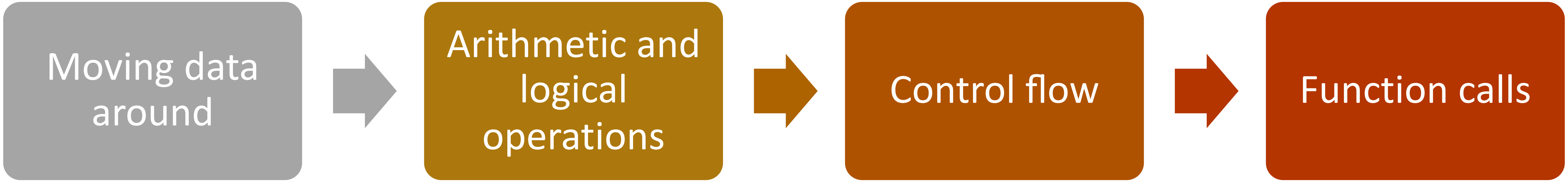
Some registers take on special responsibilities during program execution.

- **%rax** stores the return value
- **%rdi** stores the first parameter to a function
- **%rsi** stores the second parameter to a function
- **%rdx** stores the third parameter to a function
- **%rip** stores the address of the next instruction to execute
- **%rsp** stores the address of the current top of the stack

See the x86-64 Guide and Reference Sheet on the Resources webpage for more!

https://web.stanford.edu/class/cs107/resources/x86-64-reference.pdf

Learning Assembly



Moving data around

Arithmetic and logical operations

Control flow

Function calls

2/10

2/17

Today

(get excited:
learn how instructions
execute in order)

2/24

Register Responsibilities

Some registers take on special responsibilities during program execution.

- **%rax** stores the return value
- **%rdi** stores the first parameter to a function
- **%rsi** stores the second parameter to a function
- **%rdx** stores the third parameter to a function
- **%rip** stores the address of the next instruction to execute
- **%rsp** stores the address of the current top of the stack

See the x86-64 Guide and Reference Sheet on the Resources webpage for more!

Plan For Today

- Control Flow
  - Condition Codes
  - Assembly Instructions
- Conditional branches: If statements
- Announcements
- Loops
  - While loops
  - For loops

Plan For Today

- Control Flow
  - Condition Codes
  - Assembly Instructions
- Conditional branches: If statements
- Announcements
- Loops
  - While loops
  - For loops
- Optimizations

Control Flow

What does it mean for a program to execute?

Executing instructions

So far:

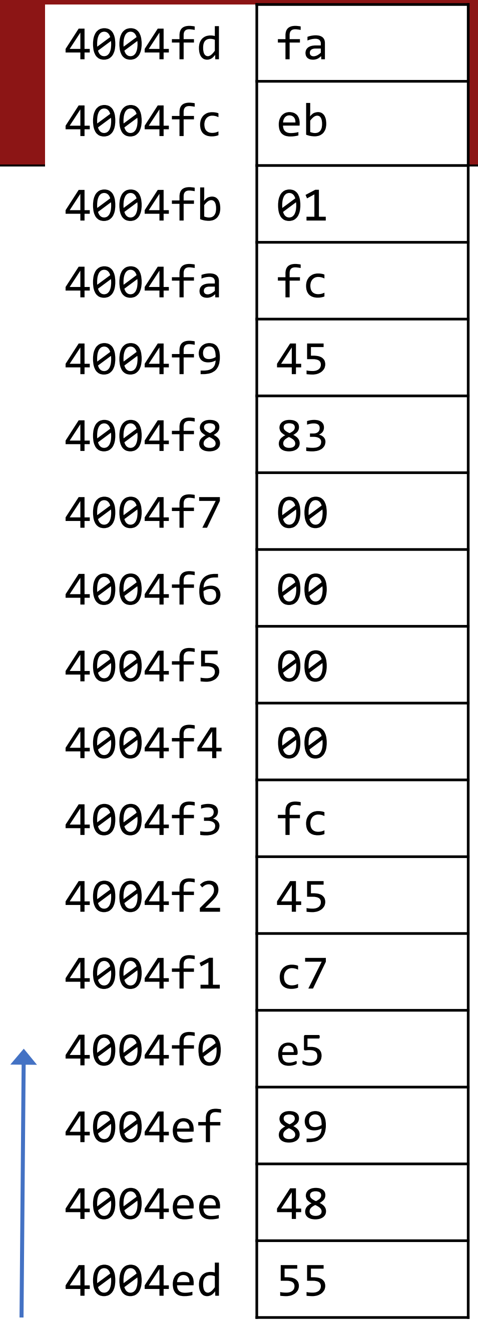
- Program values can be stored in memory or registers.
- Assembly instructions read/write values back and forth between registers (on the CPU) and memory.
- Assembly instructions are also stored in memory.

Today:

- **Who controls the instructions**? How do we know what to do now or next?

Answer:

- The **program counter** (PC), %rip.



4004fd	fa
4004fc	eb
4004fb	01
4004fa	fc
4004f9	45
4004f8	83
4004f7	00
4004f6	00
4004f5	00
4004f4	00
4004f3	fc
4004f2	45
4004f1	c7
4004f0	e5
4004ef	89
4004ee	48
4004ed	55

The program counter %rip

```
00000000004004ed <loop>:
  4004ed:    55                      push   %rbp
  4004ee:    48 89 e5                mov    %rsp,%rbp
  4004f1:    c7 45 fc 00 00 00 00    movl   $0x0,-0x4(%rbp)
  4004f8:    83 45 fc 01             addl   $0x1,-0x4(%rbp)
  4004fc:    eb fa                   jmp    4004f8 <loop+0xb>
```

The **program counter** (PC), known as %rip in x86-64, stores the address in memory of the ***next instruction*** to be executed.

%rip 0x4004ed

The program counter %rip

```
00000000004004ed <loop>:
  4004ed:    55                        push
  4004ee:    48 89 e5                  mov
  4004f1:    c7 45 fc 00 00 00 00      movl
  4004f8:    83 45 fc 01               addl
  4004fc:    eb fa                     jmp
```

4004fd	fa
4004fc	eb
4004fb	01
4004fa	fc
4004f9	45
4004f8	83
4004f7	00
4004f6	00
4004f5	00
4004f4	00
4004f3	fc
4004f2	45
4004f1	c7
4004f0	e5
4004ef	89
4004ee	48
4004ed	55

Special hardware sets the program counter to the next instruction:

%rip += size of bytes of current instruction

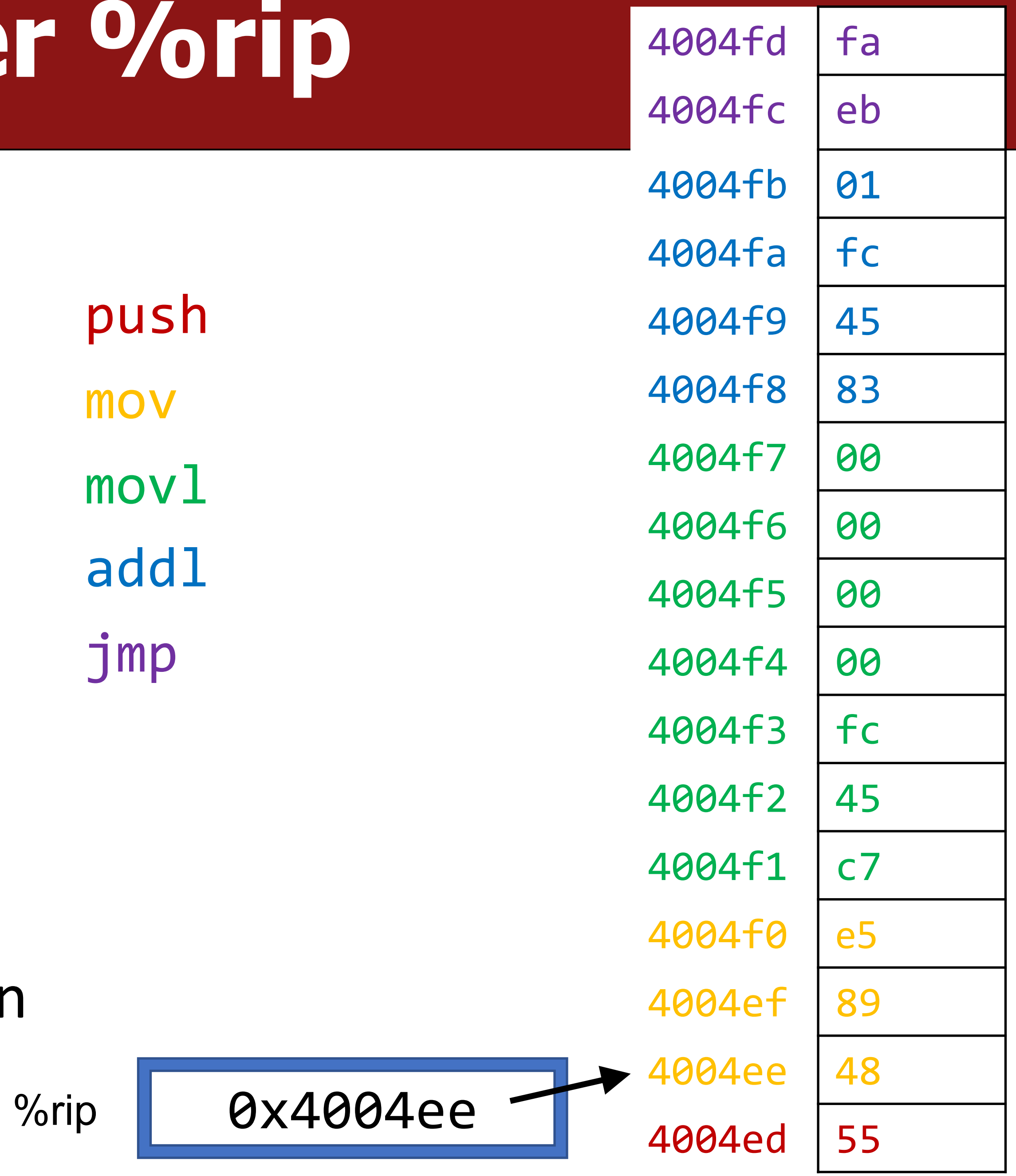
%rip 0x4004ed

The program counter %rip

```
00000000004004ed <loop>:
  4004ed:   55                     push
  4004ee:   48 89 e5               mov
  4004f1:   c7 45 fc 00 00 00 00   movl
  4004f8:   83 45 fc 01            addl
  4004fc:   eb fa                  jmp
```

Special hardware sets the program counter to the next instruction:

%rip += size of bytes of current instruction



4004fd	fa
4004fc	eb
4004fb	01
4004fa	fc
4004f9	45
4004f8	83
4004f7	00
4004f6	00
4004f5	00
4004f4	00
4004f3	fc
4004f2	45
4004f1	c7
4004f0	e5
4004ef	89
4004ee	48
4004ed	55

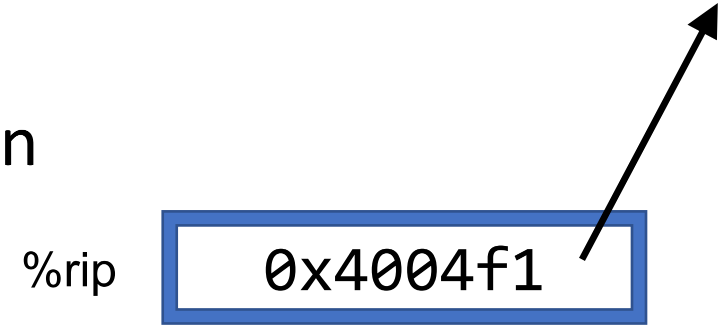
The program counter %rip

```
0000000004004ed <loop>:
  4004ed:    55                     push
  4004ee:    48 89 e5               mov
  4004f1:    c7 45 fc 00 00 00 00   movl
  4004f8:    83 45 fc 01            addl
  4004fc:    eb fa                  jmp
```

4004fd	fa
4004fc	eb
4004fb	01
4004fa	fc
4004f9	45
4004f8	83
4004f7	00
4004f6	00
4004f5	00
4004f4	00
4004f3	fc
4004f2	45
4004f1	c7
4004f0	e5
4004ef	89
4004ee	48
4004ed	55

Special hardware sets the program counter to the next instruction:

%rip += size of bytes of current instruction

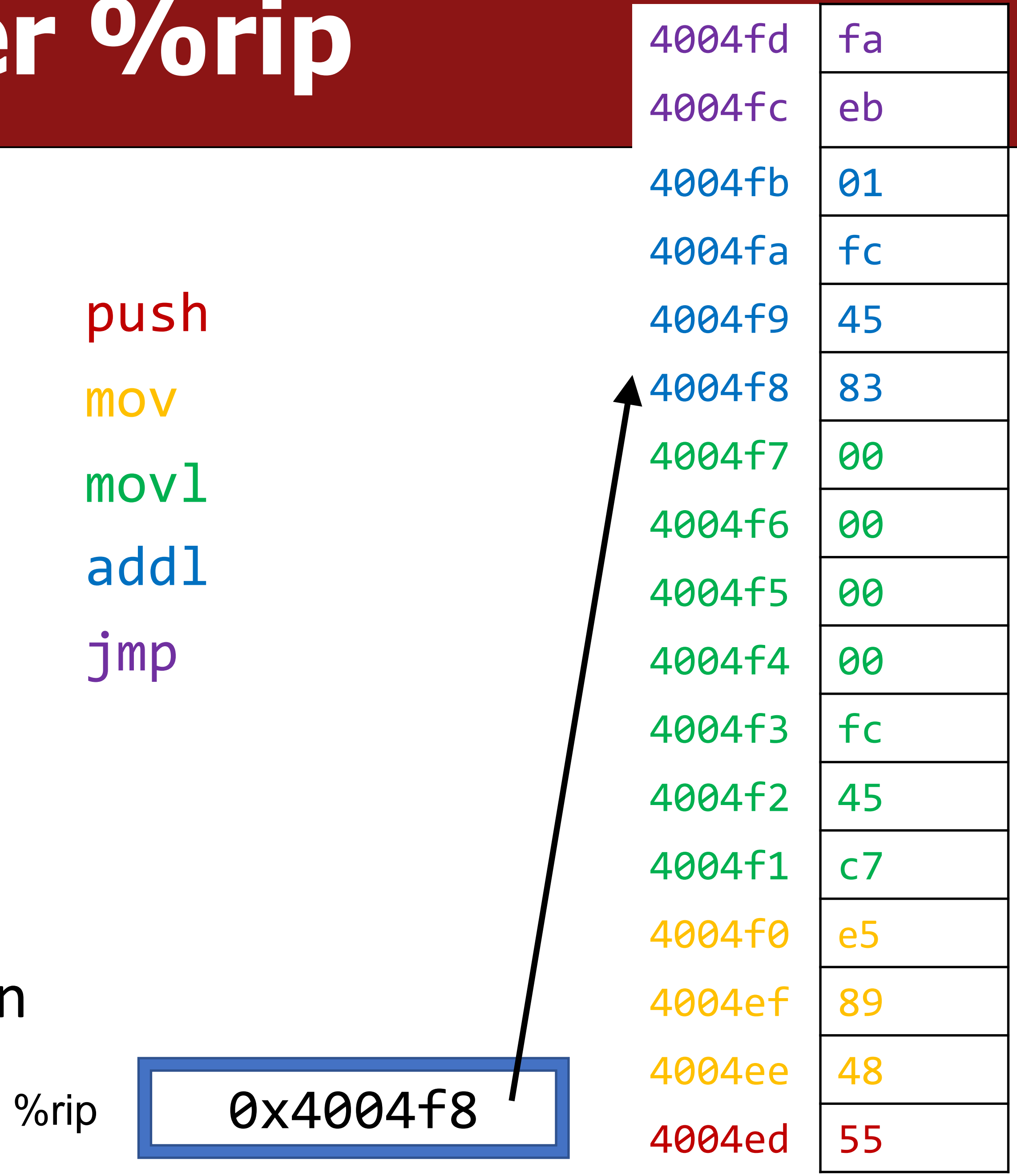


The program counter %rip

```
0000000004004ed <loop>:
  4004ed:   55                      push
  4004ee:   48 89 e5                mov
  4004f1:   c7 45 fc 00 00 00 00    movl
  4004f8:   83 45 fc 01             addl
  4004fc:   eb fa                   jmp
```

Special hardware sets the program counter to the next instruction:

%rip += size of bytes of current instruction



%rip 0x4004f8

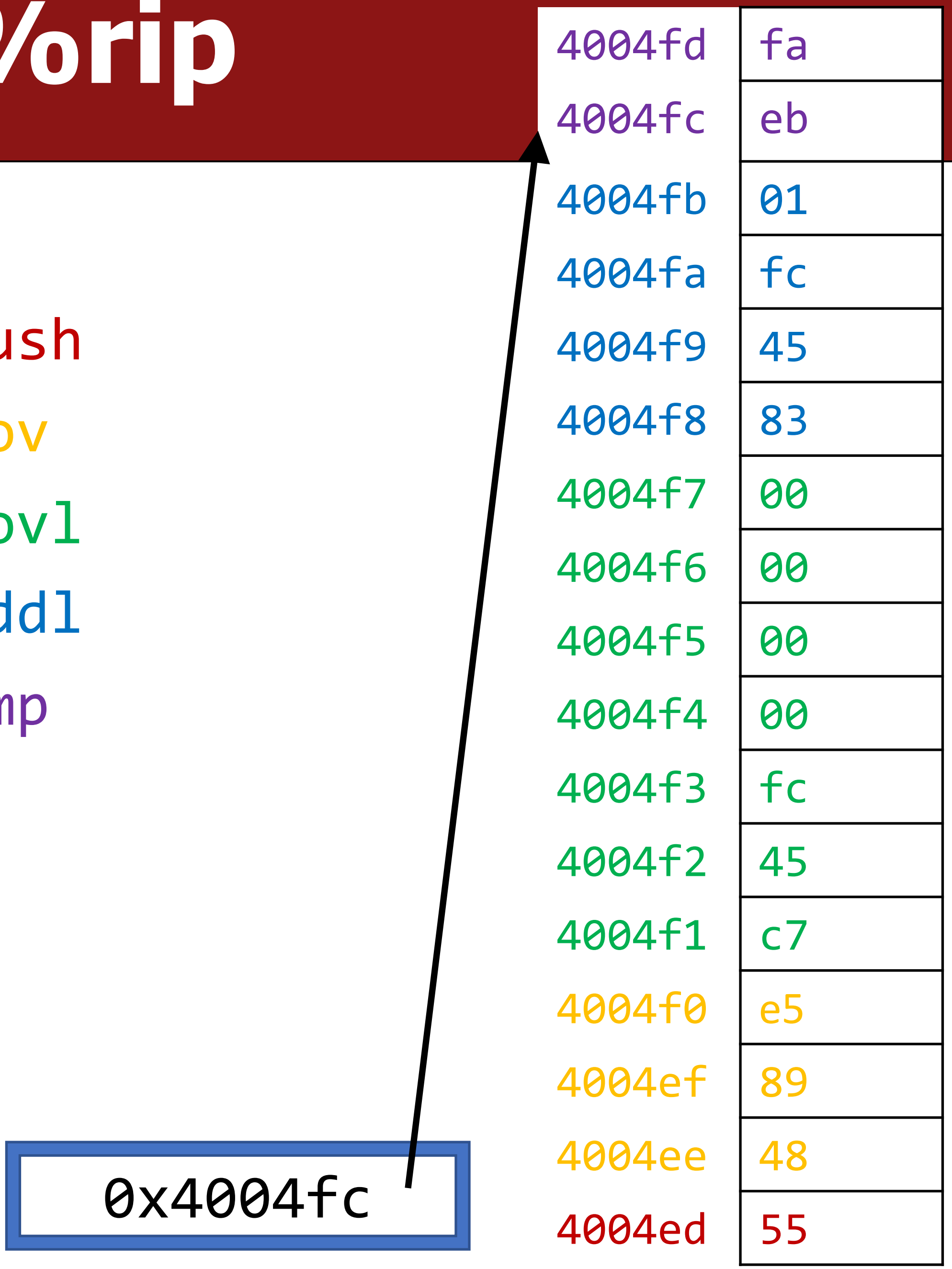
4004fd	fa
4004fc	eb
4004fb	01
4004fa	fc
4004f9	45
4004f8	83
4004f7	00
4004f6	00
4004f5	00
4004f4	00
4004f3	fc
4004f2	45
4004f1	c7
4004f0	e5
4004ef	89
4004ee	48
4004ed	55

The program counter %rip

```
00000000004004ed <loop>:
  4004ed:   55                       push
  4004ee:   48 89 e5                 mov
  4004f1:   c7 45 fc 00 00 00 00     movl
  4004f8:   83 45 fc 01              addl
  4004fc:   eb fa                    jmp
```

Special hardware sets the program counter to the next instruction:

%rip += size of bytes of current instruction



%rip 0x4004fc

4004fd	fa
4004fc	eb
4004fb	01
4004fa	fc
4004f9	45
4004f8	83
4004f7	00
4004f6	00
4004f5	00
4004f4	00
4004f3	fc
4004f2	45
4004f1	c7
4004f0	e5
4004ef	89
4004ee	48
4004ed	55

“Interfering” with %rip

1. How do we repeat instructions in a loop?

The jmp instruction

```
0000000004004ed <loop>:
   4004ed:    push   %rbp
   4004ee:    mov    %rsp,%rbp
   4004f1:    movl   $0x0,-0x4(%rbp)
   4004f8:    addl   $0x1,-0x4(%rbp)
   4004fc:    jmp    4004f8 <loop+0xb>
```

jmp is an **unconditional jump** that sets the program counter to the **jump target**.

%rip 0x4004fc

4004fd	fa
4004fc	eb
4004fb	01
4004fa	fc
4004f9	45
4004f8	83
4004f7	00
4004f6	00
4004f5	00
4004f4	00
4004f3	fc
4004f2	45
4004f1	c7
4004f0	e5
4004ef	89
4004ee	48
4004ed	55

The jmp instruction

```
0000000004004ed <loop>:
   4004ed:    push    %rbp
   4004ee:    mov     %rsp,%rbp
   4004f1:    movl    $0x0,-0x4(%rbp)
   4004f8:    addl    $0x1,-0x4(%rbp)
   4004fc:    jmp     4004f8 <loop+0xb>
```

jmp is an **unconditional jump** that sets the program counter to the **jump target**.

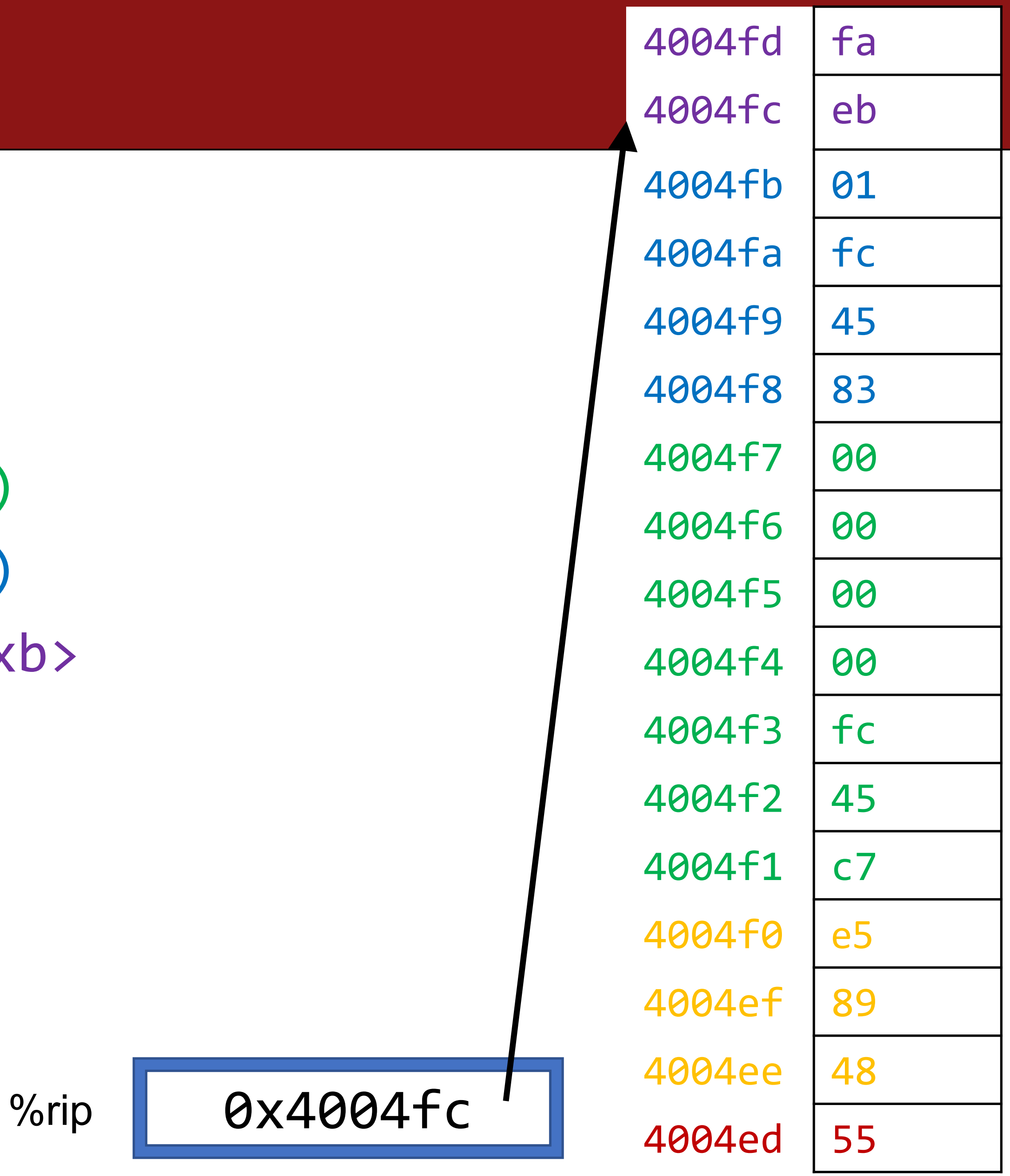
%rip 0x4004f8

4004fd	fa
4004fc	eb
4004fb	01
4004fa	fc
4004f9	45
4004f8	83
4004f7	00
4004f6	00
4004f5	00
4004f4	00
4004f3	fc
4004f2	45
4004f1	c7
4004f0	e5
4004ef	89
4004ee	48
4004ed	55

The jmp instruction

```
0000000004004ed <loop>:
  4004ed:    push    %rbp
  4004ee:    mov     %rsp,%rbp
  4004f1:    movl    $0x0,-0x4(%rbp)
  4004f8:    addl    $0x1,-0x4(%rbp)
  4004fc:    jmp     4004f8 <loop+0xb>
```

jmp is an **unconditional jump** that sets the program counter to the **jump target**.



4004fd	fa
4004fc	eb
4004fb	01
4004fa	fc
4004f9	45
4004f8	83
4004f7	00
4004f6	00
4004f5	00
4004f4	00
4004f3	fc
4004f2	45
4004f1	c7
4004f0	e5
4004ef	89
4004ee	48
4004ed	55

%rip 0x4004fc

The jmp instruction

```
0000000004004ed <loop>:
  4004ed:   push   %rbp
  4004ee:   mov    %rsp,%rbp
  4004f1:   movl   $0x0,-0x4(%rbp)
  4004f8:   addl   $0x1,-0x4(%rbp)
  4004fc:   jmp    4004f8 <loop+0xb>
```

4004fd	fa
4004fc	eb
4004fb	01
4004fa	fc
4004f9	45
4004f8	83
4004f7	00
4004f6	00
4004f5	00
4004f4	00
4004f3	fc
4004f2	45
4004f1	c7
4004f0	e5
4004ef	89
4004ee	48
4004ed	55

jmp is an **unconditional jump** that sets the program counter to the **jump target**.

control_intro.c

%rip 0x4004f8

“Interfering” with %rip

1. How do we repeat instructions in a loop?

```
jmp [target]
```

- Gives us an infinite loop
- A 1-step unconditional jump (always jump when we execute this instruction)

What if we want a **conditional jump**?

“Interfering” with %rip

1. How do we repeat instructions in a loop?
2. How do we skip instructions in an if/if-else statement?
3. How do we loop while some condition is true?

Answer:
condition codes
+ conditional jumps!

Typical 2-instruction control flow

1. Compare two values to **write** condition codes
   (implicit destination register)

   ```
   cmp S1, S2
   test S1, S2
   ```

2. Conditionally jump based on **reading** condition codes
   (implicit source register)

   `je/jz`, `jne/jnz`, `jl`, `jg`, …,

Condition codes are special registers that auto-store the results of the most recent arith/logical operation.

in gdb:
`%eflags`

Condition codes

1. Compare two values to **write** condition codes (implicit destination register)

```
cmp S1, S2          S2 - S1
test S1, S2         S2 & S1
```

Do not store result;
just set condition codes

Condition codes are single-bit registers, packed into `%eflags` for convenience. If `t` is the result of cmp/test arithmetic operations:

- ZF = zero flag (`t == 0`)
- SF = sign flag (`t < 0`)
- CF = carry flag (there was a carry out of MSB*, i.e., unsigned overflow)
- OF = overflow flag (MSB* changed from 0 to 1, i.e., signed overflow)

*MSB: Most Significant Bit

Exercise: Condition codes

```
00000000004004d6 <if_then>:
  4004d6:   83 ff 06    cmp    $0x6,%edi
  4004d9:   75 03       jne    4004de <if_then+0x8>
  400rdb:   83 c7 01    add    $0x1,%edi
  4004de:   8d 04 3f    lea    (%rdi,%rdi,1),%eax
  4004e1:   c3          retq
```

%edi `0x5`

1. **After** the cmp instruction, which of the below condition codes are set?
   - ZF = zero flag (`t == 0`)
   - SF = sign flag (`t < 0`)
   - CF = carry flag (unsigned overflow)
   - OF = overflow flag (signed overflow)
2. **After** the cmp instruction, what is `%edi`?

Exercise: Condition codes

```
00000000004004d6 <if_then>:
  4004d6:   83 ff 06    cmp    $0x6,%edi
  4004d9:   75 03       jne    4004de <if_then+0x8>
  400rdb:   83 c7 01    add    $0x1,%edi
  4004de:   8d 04 3f    lea    (%rdi,%rdi,1),%eax
  4004e1:   c3          retq
```

%edi `0x5`

1. **After** the cmp instruction, which of the below condition codes are set?
   - ZF = zero flag (`t == 0`)
   - SF = sign flag (`t < 0`)
   - CF = carry flag (unsigned overflow) (fixed since lecture)
   - OF = overflow flag (signed overflow)
2. **After** the cmp instruction, what is `%edi`? `%edi` is unchanged

Step 1, Control flow: `cmp`, `test` ⚠

1. Compare two values to **write** condition codes (implicit destination register)

```
cmp S1, S2             S2 - S1
test S1, S2            S2 & S1
```

⚠ Note the operand order!

⚠ `cmp`/`test` **do not** store the result (unlike `sub`/`and`)! They just set condition codes.

Cool tip: `testq %rax,%rax` checks if `%rax` is positive, negative, or zero.

Step 2, Control flow: Conditional jump

2. Conditionally jump based on **reading** condition codes (implicit source register)

```
je target              jump if ZF is 1
```

- Target is a memory address—the address of instruction.
- We jump to target if specific condition codes are on (ZF, SF, CF, OF).

- Jumps are also known as **branch instructions**.

Exercise 1: Conditional jump

```
je target          jump if ZF is 1
```

Let %edi store 0x10. Will we jump in the following cases?

%edi 0x10

1.
```
cmp $0x10,%edi
je  40056f
add  $0x1,%edi
```

2.
```
test $0x10,%edi
je   40056f
add  $0x1,%edi
```

Exercise 1: Conditional jump

```
je target                   jump if ZF is 1
```

Let %edi store 0x10. Will we jump in the following cases?

%edi | 0x10

1.
```
cmp $0x10,%edi
je  40056f
add  $0x1,%edi
```

S2 - S1 == 0, so jump

2.
```
test $0x10,%edi
je   40056f
add  $0x1,%edi
```

S2 & S1 != 0, so don’t jump

Step 2, Control flow: conditional jump

2. Conditionally jump based on **reading** condition codes (implicit source register)

Instruction	Synonym	Set Condition
`je` *Label*	`jz`	Equal / zero (ZF = 1)
`jne` *Label*	`jnz`	Not equal / not zero (ZF = 0)
`js` *Label*		Negative (SF = 1)
`jns` *Label*		Nonnegative (SF = 0)
`jg` *Label*	`jnle`	Greater (signed >) (SF = 0 and SF = OF)
`jge` *Label*	`jnl`	Greater or equal (signed >=) (SF = OF)
`jl` *Label*	`jnge`	Less (signed <) (SF != OF)
`jle` *Label*	`jng`	Less or equal (signed <=) (ZF = 1 or SF! = OF)
`ja` *Label*	`jnbe`	Above (unsigned >) (CF = 0 and ZF = 0)
`jae` *Label*	`jnb`	Above or equal (unsigned >=) (CF = 0)
`jb` *Label*	`jnae`	Below (unsigned <) (CF = 1)
`jbe` *Label*	`jna`	Below or equal (unsigned <=) (CF = 1 or ZF = 1)

Exercise 2: Conditional jump

```
0000000004004d6 <if_then>:
  4004d6:   83 ff 06    cmp    $0x6,%edi
  4004d9:   75 03       jne    4004de <if_then+0x8>
  400rdb:   83 c7 01    add    $0x1,%edi
  4004de:   8d 04 3f    lea    (%rdi,%rdi,1),%eax
  4004e1:   c3          retq
```

%edi 0x5

1. What is the value of %rip after executing the jne instruction?
   A. 4004d9
   B. 4004db
   C. 4004de
   D. Other

2. What is the value of %eax when we hit the retq instruction?
   A. 4004e1
   B. 0x2
   C. 0xa
   D. 0xc
   E. Other

Condition code details and other conditional ops

Reference

- Condition codes are set for many operations other than **`test`** and **`set`**, and there are many details as to which instructions set what condition codes.
- There exist conditional operators other than jump: **`setx`** and **`cmov`**.

I want to cover more conceptually challenging material in today's lecture, so the following slides are here for your reference.

Please read B&O 3.6 for more information if you find it useful.

Details about condition codes

Reference

- Different combinations of condition codes can indicate different things.
  - To check equality, we can cmp and look at the ZF flag (a = b means a – b = 0).
  - To check sign of %eax, we can test %eax,%eax and look at the SF or ZF flag
- Previously-discussed arithmetic and logical instructions update these flags. **lea** does not (it was intended only for address computations).
- Logical operations (**xor**, etc.) set carry and overflow flags to zero.
- Shift operations set the carry flag to the last bit shifted out and set the overflow flag to zero.
- For more complicated reasons, **inc** and **dec** set the overflow and zero flags, but leave the carry flag unchanged.

`set`: Read condition codes

Reference

`set` instructions conditionally set a byte to 0 or 1.

- Reads current state of flags
- Destination is a single-byte sub-register (e.g., %al)
- Does not perturb other bytes of register
- Typically followed by movzbl to zero those bytes

```
int small(int x) {
    return x < 16;
}
```

```
cmp $0xf,%edi
setle %al
movzbl %al, %eax
retq
```

`set`: Read condition codes

Reference

Instruction	Synonym	Set Condition (1 if true, 0 if false)
`sete D`	`setz`	Equal / zero
`setne D`	`setnz`	Not equal / not zero
`sets D`		Negative
`setns D`		Nonnegative
`setg D`	`setnle`	Greater (signed >)
`setge D`	`setnl`	Greater or equal (signed >=)
`setl D`	`setnge`	Less (signed <)
`setle D`	`setng`	Less or equal (signed <=)
`seta D`	`setnbe`	Above (unsigned >)
`setae D`	`setnb`	Above or equal (unsigned >=)
`setb D`	`setnae`	Below (unsigned <)
`setbe D`	`setna`	Below or equal (unsigned <=)

`cmov`: Conditional move

Reference

`cmovx src,dst` conditionally moves data in `src` to data in `dst`.

- Mov `src` to `dst` if condition x holds; no change otherwise
- `src` is memory address/register, `dst` is register
- May be more efficient than branch (i.e., jump)
- Often seen with C ternary operator: `result = test ? then: else;`

```
int max(int x, int y) {
    return x > y ? x : y;
}
```

```
cmp    %edi,%esi
mov    %edi, %eax
cmovge %esi, %eax
retq
```

cmov: Conditional move

Reference

Instruction	Synonym	Move Condition
`cmove S,R`	`cmovz`	Equal / zero (ZF = 1)
`cmovne S,R`	`cmovnz`	Not equal / not zero (ZF = 0)
`cmovs S,R`		Negative (SF = 1)
`cmovns S,R`		Nonnegative (SF = 0)
`cmovg S,R`	`cmovnle`	Greater (signed >) (SF = 0 and SF = OF)
`cmovge S,R`	`cmovnl`	Greater or equal (signed >=) (SF = OF)
`cmovl S,R`	`cmovnge`	Less (signed <) (SF != OF)
`cmovle S,R`	`cmovng`	Less or equal (signed <=) (ZF = 1 or SF! = OF)
`cmova S,R`	`cmovnbe`	Above (unsigned >) (CF = 0 and ZF = 0)
`cmovae S,R`	`cmovnb`	Above or equal (unsigned >=) (CF = 0)
`cmovb S,R`	`cmovnae`	Below (unsigned <) (CF = 1)
`cmovbe S,R`	`cmovna`	Below or equal (unsigned <=) (CF = 1 or ZF = 1)

lab6’s conditional move

Reference

```
int signed_division(int x) {
     return x / 4;
}
```

```
signed_division:
  leal 3(%rdi), %eax
  testl %edi, %edi
  cmovns %edi, %eax
  sarl $2, %eax
  ret
```

Put x + 3 into %eax

Check the sign of x

If x is positive, put x into %eax

Divide %eax by 4

Condition code details and other conditional ops

Reference

(end of reference slides)

Please read B&O 3.6 for more information if you find it useful.

Plan For Today

- Control Flow
  - Condition Codes
  - Assembly Instructions
- Conditional branches: If statements
- Announcements
- Loops
  - While loops
  - For loops

“Interfering” with %rip

1. How do we repeat instructions in a loop?
2. How do we skip instructions in an if/if-else statement?
3. How do we loop while some condition is true?

The code we’ve been working with

```
0000000004004d6 <if_then>:
  4004d6:  83 ff 06   cmp    $0x6,%edi
  4004d9:  75 03      jne    4004de <if_then+0x8>
  400rdb:  83 c7 01   add    $0x1,%edi
  4004de:  8d 04 3f   lea    (%rdi,%rdi,1),%eax
  4004e1:  c3         retq
```

This code can be translated into C function code containing a branch statement (`if`)!

Practice: Fill In The Blank

```
int if_then(int param1)
{
  if ( __________ ) {
      _________;
  }

  return __________;
}
```

```
0000000000004004d6 <if_then>:
  4004d6:    cmp   $0x6,%edi
  4004d9:    jne   4004de
  4004db:    add   $0x1,%edi
  4004de:    lea   (%rdi,%rdi,1),%eax
  4004e1:    retq
```

🤔

Practice: Fill In The Blank

```
int if_then(int param1)
{
  if ( param1 == 6 ) {
       param1++;
  }

  return param1 * 2;
}
```

```
0000000000004004d6 <if_then>:
  4004d6:   cmp  $0x6,%edi
  4004d9:   jne  4004de
  4004db:   add  $0x1,%edi
  4004de:   lea  (%rdi,%rdi,1),%eax
  4004e1:   retq
```

Plan For Today

- Control Flow
  - Condition Codes
  - Assembly Instructions
- Conditional branches: If statements
- Announcements
- Loops
  - While loops
  - For loops

Joke break

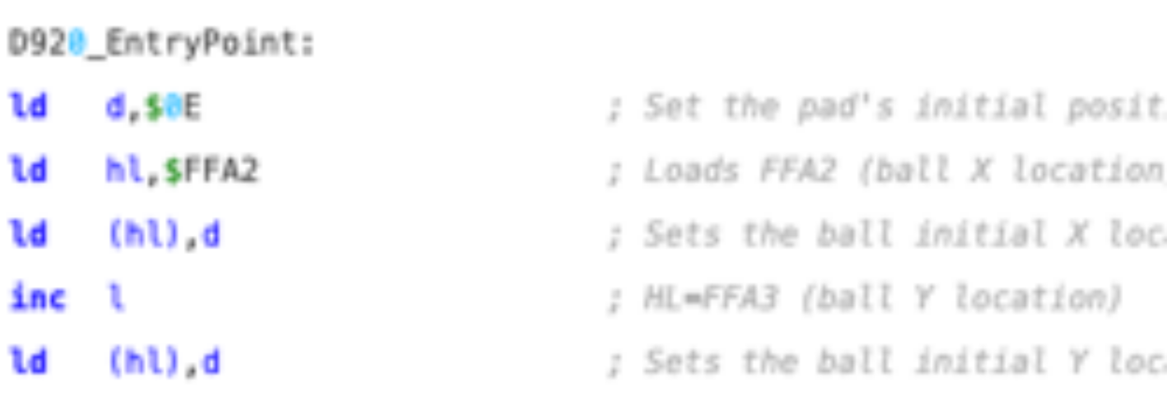
https://www.youtube.com/watch?v=D3EvpRHL_vk

Hacking Pokemon Blue into Pong

- Instructions are bytes in memory!
- Find an exploit that lets you change the **program counter**:

  > “8F is an item executing machine code starting from $D163 (Number of Pokemon) upon use."

- **Write** executable code by navigating the world and moving around items



```
D920_EntryPoint:
ld   d,$0E                              ; Set the pad's initial posit
ld   hl,$FFA2                           ; Loads FFA2 (ball X location
ld   (hl),d                             ; Sets the ball initial X loc
inc  l                                  ; HL=FFA3 (ball Y location)
ld   (hl),d                             ; Sets the ball initial Y loc
```

Announcements

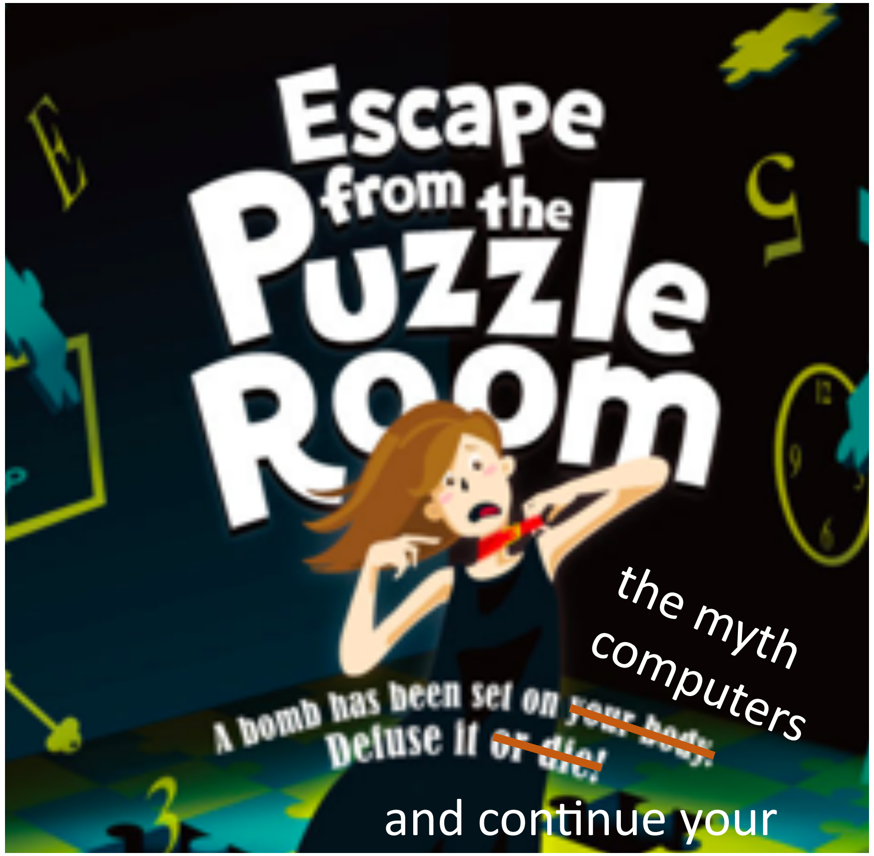
- Midterm scores will be on website gradebook after regrade deadline closes on **Monday 2/24, 11:59pm**
- Note about makeup labs:
  - If you cannot attend your assigned lab, you may go to a different lab that same week *if space is available*.
  - We want to note that we may have to turn people away who are making up a lab if there is not enough space to accommodate them.
  - If you need to attend a makeup, you should plan ahead accordingly to ensure you can get lab credit for that week.
- Assignment 6 released today ☺

Preparing for assign6

- atm.c Security and Robustness
- Binary Bomb

Binary Bomb is like an escape game:
- Secret codes (assembly instructions)
- Fun tools (gdb, objdump)
- Unlock each level to continue
- Catharsis with successful escape
- Time limit (due M 3/2, grace W 3/4)

- **Please start early.**
- **Please use gdb.**
- **Read the textbook (B&O) if you find it useful.**


Escape from the Puzzle Room
A bomb has been set on ~~your body.~~
Defuse it ~~or die!~~

the myth computers

and continue your wholesome CS education!

Tips for assign6

- **Registers do not have addresses**. They are not located in memory.
- Draw pictures clearly depicting what is in memory and what is in registers.
- **Deadlisting** (reading assembly without executing anything) will be tedious.
- **Use gdb**.

- `lea` **does not** access memory; it performs arithmetic and stores the result.
- `test/cmp s1,s2` work like `and/sub s1,s2`. However, `test/cmp` **do not** store the result anywhere; they only update condition codes.
- `test/cmp` + conditional jump can often be translated into a single C control statement.
- `testq %rax, %rax` checks if `%rax` is positive, negative, or zero.

Plan For Today

- Control Flow
  - Condition Codes
  - Assembly Instructions
- Conditional branches: If statements
- Announcements
- Loops
  - While loops
  - For loops

Common If-Else Construction ⭐

If-Else In C

```
if (arg > 3) {
     ret = 10;
} else {
     ret = 0;
}

ret++;
```

If-Else In Assembly pseudocode

```
Test
Jump to else-body if test fails
If-body
Jump to past else-body
Else-body
Past else body
```

Practice: Fill in the Blank

If-Else In C

```
if ( __________ ) {
    _________;
} else {
    ________;
}
______;
```

```
400552 <+0>:   cmp   $0x3,%edi
400555 <+3>:   jle   0x40055e <if_else+12>
400557 <+5>:   mov   $0xa,%eax
40055c <+10>:  jmp   0x400563 <if_else+17>
40055e <+12>:  mov   $0x0,%eax
400563 <+17>:  add   $0x1,%eax
```

If-Else In Assembly pseudocode

```
Test
Jump to else-body if test fails
If-body
Jump to past else-body
Else-body
Past else body
```

ifelse.c

Practice: Fill in the Blank

If-Else In C

```
if (  arg > 3   ) {
      ret = 10;
} else {
      ret = 0;
}
ret++;
```

```
400552 <+0>:   cmp  $0x3,%edi
400555 <+3>:   jle  0x40055e <if_else+12>
400557 <+5>:   mov  $0xa,%eax
40055c <+10>:  jmp  0x400563 <if_else+17>
40055e <+12>:  mov  $0x0,%eax
400563 <+17>:  add  $0x1,%eax
```

If-Else In Assembly pseudocode

```
Test
Jump to else-body if test fails
If-body
Jump to past else-body
Else-body
Past else body
```

ifelse.c

“Interfering” with %rip

1. How do we repeat instructions in a loop?
2. How do we skip instructions in an in-else statement?
3. How do we loop while some condition is true?

Plan For Today

- Control Flow
  - Condition Codes
  - Assembly Instructions
- Conditional branches: If statements
- Announcements
- Loops
  - While loops
  - For loops

While loops and assembly

```
void loop() {
  int i = 0;
  while (i < 100) {
    i++;
  }
}
```

```
00000000004004d6 <loop>:
  4004d6 <+0>:   mov   $0x0,%eax
  4004db <+5>:   jmp   4004e0 <loop+0xa>
  4004dd <+7>:   add   $0x1,%eax
  4004e0 <+10>:  cmp   $0x63,%eax
  4004e3 <+13>:  jle   4004dd <loop+0x7>
  4004e5 <+15>:  repz  retq
```

1. Which register is C code's `i`?
2. What is the unconditional **`jmp`** instruction doing?
3. What are the **`cmp`** and **`jle`** instructions doing?
   (**`jle`**: jump less equal; signed <=)

while_loop

`while_loop.c`

While loops and assembly: Recap

```
void loop() {
  int i = 0;
  while (i < 100) {
    i++;
  }
}
```

```
00000000004004d6 <loop>:
  4004d6 <+0>:   mov   $0x0,%eax
  4004db <+5>:   jmp   4004e0 <loop+0xa>
  4004dd <+7>:   add   $0x1,%eax
  4004e0 <+10>:  cmp   $0x63,%eax
  4004e3 <+13>:  jle   4004dd <loop+0x7>
  4004e5 <+15>:  repz  retq
```

1. Jumps to while loop conditional

2. If %eax – 0x63 <= 0 (i.e., %eax <= 99), then jump to execute loop body. Else, execute next instruction (i.e., exit loop)

gdb tips

`layout split` (ctrl-x a: exit, ctrl-l: resize)	View C, assembly, and gdb (lab6)
`info reg`	Print all registers
`p $eax`	Print register value
`p $eflags`	Print all condition codes currently set
`b *0x400546`	Set breakpoint at assembly instruction
`b *0x400550 if $eax > 98`	Set **conditional breakpoint**
`ni`	Next assembly instruction
`si`	Step into assembly instruction (will step into function calls)

Common While Loop Construction ⭐

```
void loop() {
  int i = 0;
  while (i < 100) {
    i++;
  }
}
```

```
00000000004004d6 <loop>:
  4004d6 <+0>:   mov   $0x0,%eax
  4004db <+5>:   jmp   4004e0 <loop+0xa>
  4004dd <+7>:   add   $0x1,%eax
  4004e0 <+10>:  cmp   $0x63,%eax
  4004e3 <+13>:  jle   4004dd <loop+0x7>
  4004e5 <+15>:  repz  retq
```

C pseudocode

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

Assembly pseudocode

```
Jump to test
Body
Test
Jump to body if success
```

Plan For Today

- Control Flow
  - Condition Codes
  - Assembly Instructions
- Conditional branches: If statements
- Announcements
- Loops
  - While loops
  - For loops

Common For Loop Construction ⭐

C For loop

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

C Equivalent While Loop

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

Assembly pseudocode

```
Init
Jump to test
Body
Update
Test
Jump to body if success
```

For loops and while loops are treated (essentially) the same when compiled down to assembly.

Back to Our First Assembly

```
int sum_array(int arr[], int nelems) {
   int sum = 0;
   for (int i = 0; i < nelems; i++) {
      sum += arr[i];
   }
   return sum;
}
```

```
00000000004005b6 <sum_array>:
  4005b6:         mov    $0x0,%edx
  4005bb<+5>:     mov    $0x0,%eax
  4005c0<+10>:    jmp    4005cb <sum_array+21>
  4005c2<+12>:    movslq %edx,%rcx
  4005c5<+15>:    add    (%rdi,%rcx,4),%eax
  4005c8<+18>:    add    $0x1,%edx
  4005cb<+21>:    cmp    %esi,%edx
  4005cd<+23>:    jl     4005c2 <sum_array+12>
  4005cf<+25>:    repz retq
```

1. Which register is C code's `sum`?
2. Which register is C code's `i`?
3. Which assembly instruction is C code's `sum += arr[i]`?
4. What are the **`cmp`** and **`jl`** instructions doing?
   (**`jl`**: jump less; signed <)

sum_array

`sum_array.c`

gdb tips ⭐⭐⭐

`p/x $rdi`	Print register value in hex
`p/t $rsi`	Print register value in binary
`x $rdi`	Examine the byte stored at this address
`x/4bx $rdi`	Examine 4 bytes starting at this address
`x/4wx $rdi`	Examine 4 ints starting at this address

Plan For Today

- Control Flow
  - Condition Codes
  - Assembly Instructions
- Conditional branches: If statements
- Announcements
- Loops
  - While loops
  - For loops
- Optimizations

Optimizations you'll see

`nop`

- **`nop/nopl`** are "no-op" instructions – they do nothing!
- Intent: Make functions align on address boundaries that are nice multiples of 8.

`mov %ebx,%ebx`

- Zeros out the top 32 register bits
  (because a `mov` on an e-register zeros out rest of 64 bits).

`xor %ebx,%ebx`

- Optimizes for performance as well as code size (read more here):

```
        b8 00 00 00 00            mov $0x0,%eax
        31 c0                     xor %eax,%eax
```

Loop optimization in GCC

C For loop

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

?

GCC assembly pseudocode

```
Init
Jump to test
Body
Update
Test
Jump to body if success
```

Possible alternative?

```
Init
Test
Jump past loop is fails
Body
Update
Jump to Test
```

Loop optimization in GCC

C For loop

```
for (int i = 0; i < n; i++) {
    ;
}
```

Are the number of instructions executed in the left greater, less than, or equal to those executed on the right if…

1. n = 0?
2. n = 1000?

GCC assembly pseudocode

```
Init
Jump to test
Body
Update
Test
Jump to body if success
```

Possible alternative?

```
Init
Test
Jump past loop is fails
Body
Update
Jump to Test
```

Optimizing Instruction Counts

- Both versions have the same static **instruction count** (# of written instructions).
- But they have different dynamic **instruction counts** (# of executed instructions when program is run).
  - If n = 0, right (possible alternative) is best b/c fewer instructions
  - If n is large, left (gcc is best) is best b/c fewer instructions
- The compiler may emit a static instruction count that is several times longer than an alternative, but it may be more efficient if loop executes many times.
- Does the compiler *know* that a loop will execute many times? (in general, no)
- So what if our code had loops that always execute a small number of times? How do we know when gcc makes a bad decision?
- (take EE108, EE180, CS316 for more!)

Bonus assembly exercises

Practice: Parameters

```
00000000004005ac <sum_example1>:
     4005bd:   8b 45 e8       mov  %esi,%eax
     4005c3:   01 d0          add  %edi,%eax
     4005cc:   c3             retq
```

Which of the following is most likely to have generated the above assembly?

```
// A)
void sum_example1() {
    int x;
    int y;
    int sum = x + y;
}
```

```
// B)
int sum_example1(int x, int y) {
    return x + y;
}
```

```
// C)
void sum_example1(int x, int y) {
    int sum = x + y;
}
```

Practice: Parameters

```
00000000004005ac <sum_example1>:
    4005bd:   8b 45 e8        mov  %esi,%eax
    4005c3:   01 d0           add  %edi,%eax
    4005cc:   c3              retq
```

Which of the following is most likely to have generated the above assembly?

```
// A)
void sum_example1() {
    int x;
    int y;
    int sum = x + y;
}
```

```
// B)
int sum_example1(int x, int y) {
    return x + y;
}
```

```
// C)
void sum_example1(int x, int y) {
    int sum = x + y;
}
```

B

Practice: Intermediates/registers

```
0000000000400578 <sum_example2>:
    400578:   8b 47 0c        mov  0xc(%rdi),%eax
    40057b:   03 07           add  (%rdi),%eax
    40057d:   2b 47 18        sub  0x18(%rdi),%eax
    400580:   c3              retq
```

```c
int sum_example2(int arr[]) {
    int sum = 0;
    sum += arr[0];
    sum += arr[3];
    sum -= arr[6];
    return sum;
}
```

bonus.c

1. What memory location, register, or immediate above represents the C code's sum variable?

2. What memory location, register, or immediate in the assembly code above represents the C code's 6 (as in arr[6])? 🤔

Practice: Intermediates/registers

```
0000000000400578 <sum_example2>:
    400578:   8b 47 0c       mov  0xc(%rdi),%eax
    40057b:   03 07          add  (%rdi),%eax
    40057d:   2b 47 18       sub  0x18(%rdi),%eax
    400580:   c3             retq
```

```
int sum_example2(int arr[]) {
    int sum = 0;
    sum += arr[0];
    sum += arr[3];
    sum -= arr[6];
    return sum;
}
```

1. What memory location, register, or immediate above represents the C code's `sum` variable?

   `%eax`

2. What memory location, register, or immediate in the assembly code above represents the C code's 6 (as in arr[6])?

   `0x18`

Practice: while loop

```
long loop(long a, long b) {
    long result = _______;
    while (_________) {
      result = _________;
      a = _________;
    }
    return result;
}
```

```
loop:
        movl $1, %eax
        jmp .L2
.L3
        leaq (%rdi,%rsi), %rdx
        imulq %rdx, %rax
        addq $1, %rdi
.L2
        cmpq %rsi, %rdi
        jl .L3
rep; ret
```

A label for this instruction location.

- Used in place of instruction addresses in assembly files (.s).
- These labels get compiled to addresses in object files (.o).

bonus.c

Practice: while loop

C Code

```
long loop(long a, long b) {
    long result = _______;
    while (_________) {
      result = __________;
      a = _________;
    }
    return result;
}
```

Assembly pseudocode

```
Jump to test
Body
Test
Jump to body if success
```

What does this assembly code translate to?

```
// a in %rdi, b in %rsi
loop:
    movl $1, %eax
    jmp .L2
.L3
    leaq (%rdi,%rsi), %rdx
    imulq %rdx, %rax
    addq $1, %rdi
.L2
    cmpq %rsi, %rdi
    jl .L3
rep; ret
```

What does this assembly code translate to?

Practice: while loop

C Code

```
long loop(long a, long b) {
    long result =    1    ;
    while (  a < b   ) {
      result =  result*(a+b);
      a =   a + 1   ;
    }
    return result;
}
```

Assembly pseudocode

```
Jump to test
Body
Test
Jump to body if success
```

What does this assembly code translate to?

```
// a in %rdi, b in %rsi
loop:
    movl $1, %eax
    jmp .L2
.L3
    leaq (%rdi,%rsi), %rdx
    imulq %rdx, %rax
    addq $1, %rdi
.L2
    cmpq %rsi, %rdi
    jl .L3
rep; ret
```

What does this assembly code translate to?