CS107, Lecture 18
Assembly: Control Flow

Reading: B&O 3.6

Warm-up: Reverse Engineering

```
int elem_arithmetic(int nums[], int y) {
    int z = nums[________] * ________;

    z -= ________;

    return ________;
}
----------
// nums in %rdi, y in %esi
elem_arithmetic:
  movl %esi, %eax
  imull 4(%rdi), %eax
  movslq %esi, %rsi
  subl (%rdi,%rsi,4), %eax
  lea 2(%rax, %rax), %eax
  ret
```

Warm-up: Reverse Engineering

```
int elem_arithmetic(int nums[], int y) {
    int z = nums[1] * y;

    z -= ________;

    return ________;
}
----------
// nums in %rdi, y in %esi
elem_arithmetic:
  movl %esi, %eax           // copy y into %eax
  imull 4(%rdi), %eax       // multiply %eax by nums[1]
  movslq %esi, %rsi         // sign-extend %esi to %rsi
  subl (%rdi,%rsi,4), %eax
  lea 2(%rax, %rax), %eax
  ret
```

Work through the last two blanks in groups and input your answer for the first blank on PollEv:
pollev.com/cs107 or text CS107 to 22333 once to join.

Warm-up: Reverse Engineering

```
int elem_arithmetic(int nums[], int y) {
    int z = nums[1] * y;

    z -= nums[y];

    return 2 * z + 2;
}
----------
// nums in %rdi, y in %esi
elem_arithmetic:
  movl %esi, %eax              // copy y into %eax
  imull 4(%rdi), %eax          // multiply %eax by nums[1]
  movslq %esi, %rsi            // sign-extend %esi to %rsi
  subl (%rdi,%rsi,4), %eax     // subtract nums[y] from %eax
  lea 2(%rax, %rax), %eax      // multiply %rax by 2, and add 2
  ret
```

Learning Assembly

Moving data around → Arithmetic and logical operations → Control flow → Function calls

This Lecture

Reference Sheet: `cs107.stanford.edu/resources/x86-64-reference.pdf`
See more guides on Resources page of course website!

Learning Goals

- Understand how assembly implements loops and control flow
- Learn about how assembly stores comparison and operation results in condition codes

Lecture Plan

- Assembly Execution and %rip
- Control Flow Mechanics
  - Condition Codes
  - Assembly Instructions
- If Statements

Lecture Plan

- **Assembly Execution and %rip**
- Control Flow Mechanics
  - Condition Codes
  - Assembly Instructions
- If Statements

Executing Instructions

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

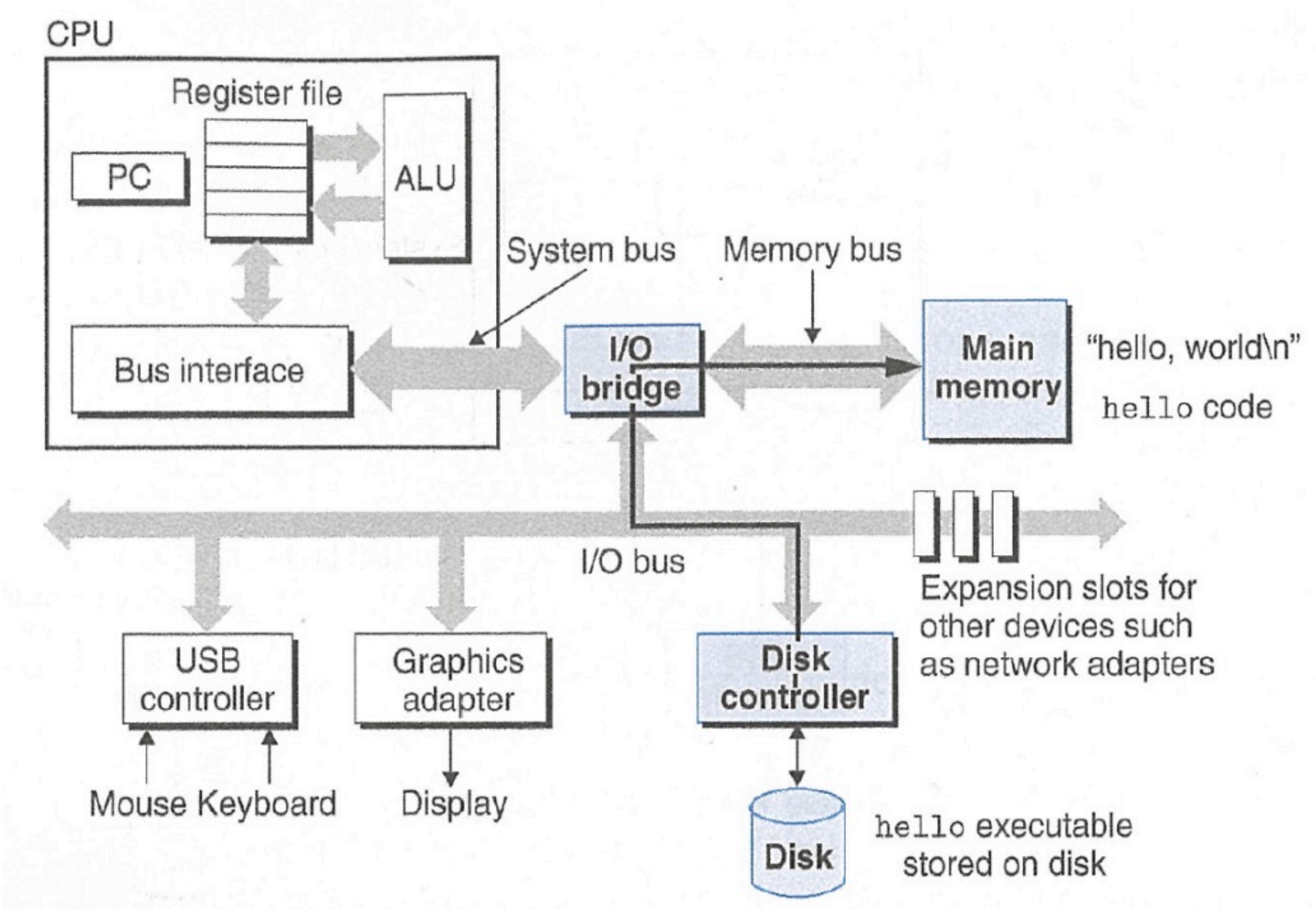
Register Responsibilities

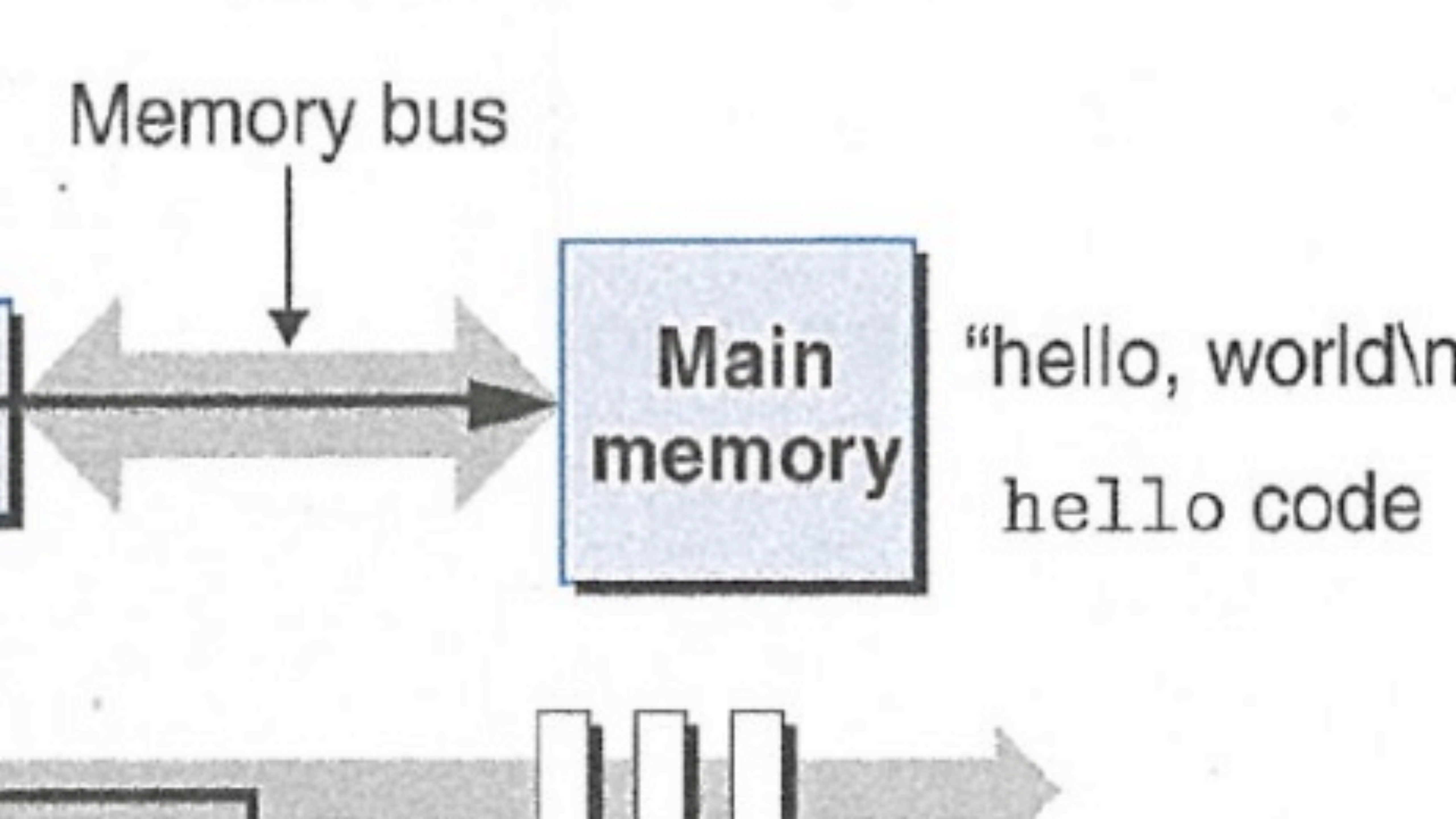
Some registers take on special responsibilities during program execution.

- **%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

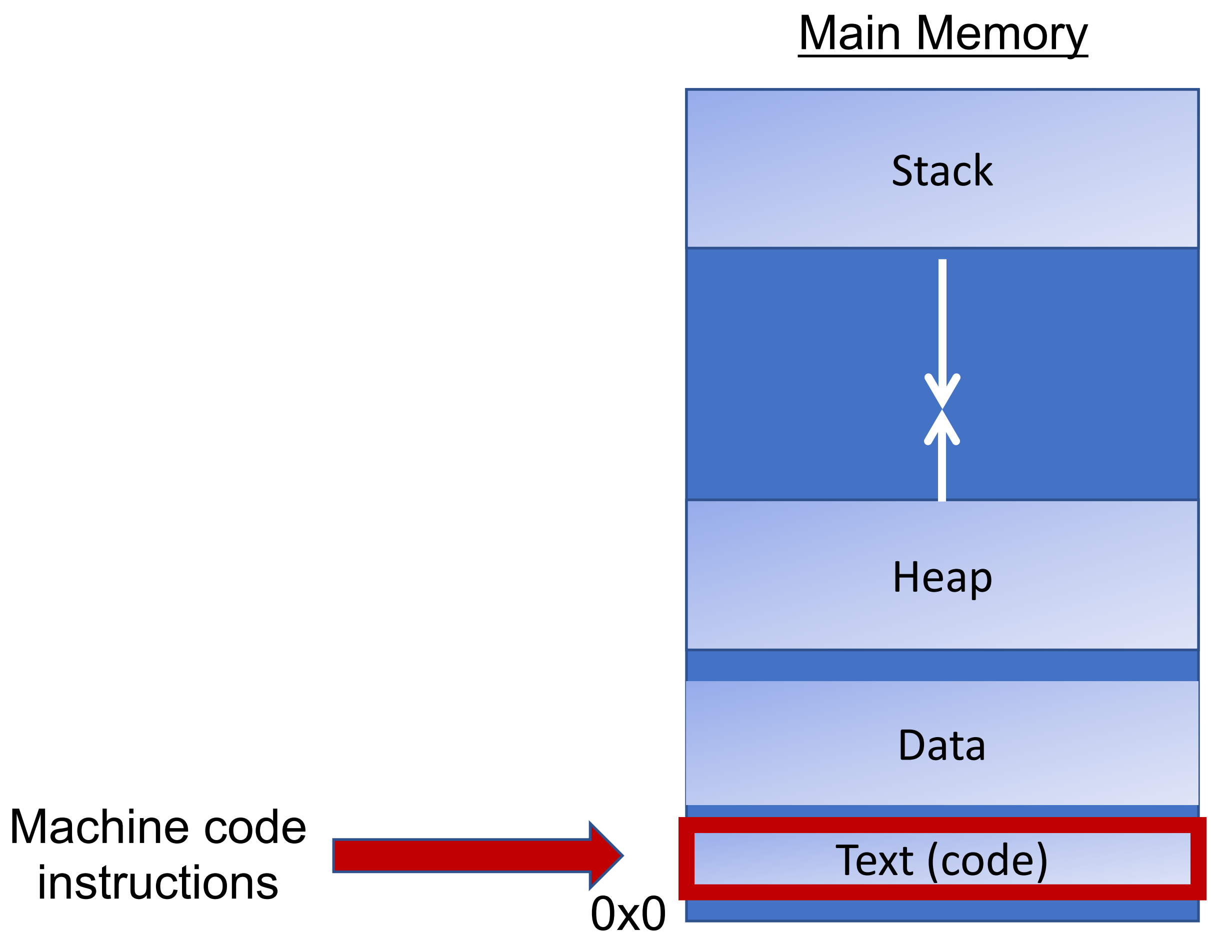
See the x86-64 Guide and Reference Sheet on the Resources webpage for more!

Instructions Are Just Bytes!




Memory bus
Main memory
"hello, world\n
hello code

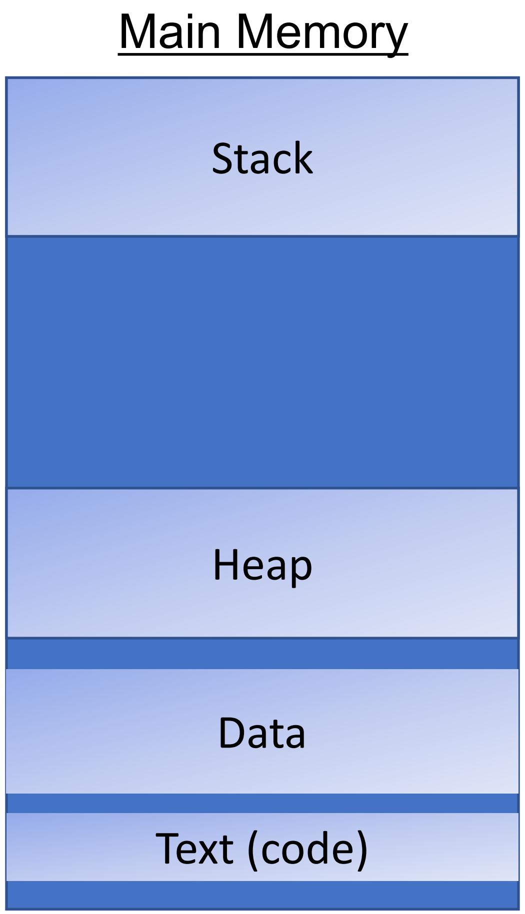
Instructions Are Just Bytes!



%ri

```
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>
```

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

```
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>
```

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

0x4004ed

%rip

%rip

```
0000000004004ed <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.

`0x4004ee`

%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

%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.

`0x4004f1`

`%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

%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>
```

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

0x4004f8

%rip

%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>
```

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

0x4004fc

%rip

%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>
```

Special hardware sets the program counter to the next instruction:

%rip += size of bytes of current instruction

`0x4004fc`

%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

Going In Circles

How can we use this representation of execution to represent e.g. a **loop**?

- **Key Idea:** we can ”interfere” with **%rip** and set it back to an earlier instruction!

Jump!

```
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>
```

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 is an **unconditional jump** that sets the program counter to the **jump target** (the operand).

`0x4004fc`

%rip

Jump!

```
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>
```

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 is an **unconditional jump** that sets the program counter to the **jump target** (the operand).

`0x4004fc`

`%rip`

Jump!

```
0000000004004ed <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 **jmp** instruction is an **unconditional jump** that sets the program counter to the **jump target** (the operand).

`0x4004fc`

%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

Jump!

```
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>
```

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 is an **unconditional jump** that sets the program counter to the **jump target** (the operand).

```
0x4004fc
%rip
```

Jump!

```
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>
```

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

This assembly represents an infinite loop in C!

```
while (true) {…}
```

0x4004fc

%rip

jmp

The **jmp** instruction jumps to another instruction in the assembly code ("Unconditional Jump").

```
jmp Label        (Direct Jump)
jmp *Operand     (Indirect Jump)
```

The destination can be hardcoded into the instruction (direct jump):

```
jmp 404f8 <loop+0xb>  # jump to instruction at 0x404f8
```

The destination can also be one of the usual operand forms (indirect jump):

```
jmp *%rax      # jump to instruction at address in %rax
```

“Interfering” with %rip

1. How do we repeat instructions in a loop?

`jmp [target]`

- A 1-step unconditional jump (always jump when we execute this instruction)

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

Lecture Plan

- Assembly Execution and %rip
- **Control Flow Mechanics**
  - Condition Codes
  - Assembly Instructions
- If Statements

Control

- In C, we have control flow statements like **if**, **else**, **while**, **for**, etc. to write programs that are more expressive than just one instruction following another.
- This is *conditional execution of statements*: executing statements if one condition is true, executing other statements if one condition is false, etc.
- How is this represented in assembly?

Control

```
if (x > y) {
      // a
} else {
      // b
}
```

In Assembly:

1. Calculate the condition result
2. Based on the result, go to a or b

Control

- In assembly, it takes more than one instruction to do these two steps.
- Most often: 1 instruction to calculate the condition, 1 to conditionally jump

Common Pattern:

1. `cmp S1, S2` // compare two values
2. **je [target]** *or* **jne [target]** *or* **jl [target]** *or* ... // conditionally jump

"jump if equal"

"jump if not equal"

"jump if less than"

Conditional Jumps

There are also variants of **jmp** that jump only if certain conditions are true ("Conditional Jump"). The jump location for these must be hardcoded into the instruction.

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

Control

Read **cmp S1,S2** as *"compare S2 to S1"*:

```
// Jump if %edi > 2
cmp $2, %edi
jg [target]
```

```
// Jump if %edi == 4
cmp $4, %edi
je [target]
```

```
// Jump if %edi != 3
cmp $3, %edi
jne [target]
```

```
// Jump if %edi <= 1
cmp $1, %edi
jle [target]
```

Wait a minute – how does the jump instruction know anything about the compared values in the earlier instruction?

Control

- The CPU has special registers called *condition codes* that are like "global variables". They *automatically* keep track of information about the most recent arithmetic or logical operation.
  - **cmp** compares via calculation (subtraction) and info is stored in the condition codes
  - conditional jump instructions look at these condition codes to know whether to jump
- What exactly are the condition codes? How do they store this information?

Condition Codes

Alongside normal registers, the CPU also has single-bit *condition code* registers. They store the results of the most recent arithmetic or logical operation.

Most common condition codes:

- **CF:** Carry flag. The most recent operation generated a carry out of the most significant bit. Used to detect overflow for unsigned operations.
- **ZF:** Zero flag. The most recent operation yielded zero.
- **SF:** Sign flag. The most recent operation yielded a negative value.
- **OF:** Overflow flag. The most recent operation caused a two's-complement overflow-either negative or positive.

Setting Condition Codes

The **cmp** instruction is like the subtraction instruction, but it does not store the result anywhere. It just sets condition codes. (**Note** the operand order!)

```
CMP S1, S2          S2 – S1
```

Instruction	Description
`cmpb`	Compare byte
`cmpw`	Compare word
`cmpl`	Compare double word
`cmpq`	Compare quad word

Conditional Jumps

Conditional jumps can look at subsets of the condition codes in order to check their condition of interest.

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 >) (ZF = 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)

Setting Condition Codes

The different conditional jumps look at appropriate combinations of condition codes to know whether the condition it cares about is true.

- E.g. **je** (“jump equal”) really checks if the ZF (zero flag) is 1
- E.g. **jns** (“jump not signed”) really checks if the SF (sign flag) is 1
- E.g. **jl** (“jump less than”) really checks if SF (sign flag) != OF (overflow flag)
  - SF = 1 and OF = 0 means no signed overflow, and the result was negative
  - SF = 0 and OF = 1 means signed overflow, and the result was positive, meaning it overflowed from the negative direction.

Control

Read **cmp S1,S2** as *"compare S2 to S1"*. It calculates S2 – S1 and updates the condition codes with the result.

```
// Jump if %edi > 2
// calculates %edi - 2
cmp $2, %edi
jg [target]
```

```
// Jump if %edi == 4
// calculates %edi - 4
cmp $4, %edi
je [target]
```

```
// Jump if %edi != 3
// calculates %edi - 3
cmp $3, %edi
jne [target]
```

```
// Jump if %edi <= 1
// calculates %edi - 1
cmp $1, %edi
jle [target]
```

Setting Condition Codes

Usually when **cmp** is paired with conditional jumps, we can read them together. But other instructions use the condition codes in different ways. Example:

The **test** instruction is like **cmp**, but for AND. It does not store the & result anywhere. It just sets condition codes.

```
TEST S1, S2          S2 & S1
```

Instruction	Description
`testb`	Test byte
`testw`	Test word
`testl`	Test double word
`testq`	Test quad word

Cool trick: if we pass the same value for both operands, we can check the sign of that value using the **Sign Flag** and **Zero Flag** condition codes!

The test Instruction

- `TEST S1, S2` is `S2 & S1`

```
test %edi, %edi
jns  …
```

```
%edi & %edi is nonnegative
%edi is nonnegative
```

Condition Codes

- 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.

Lecture Plan

- Assembly Execution and %rip
- Control Flow Mechanics
  - Condition Codes
  - Assembly Instructions
- **If Statements**

Practice: Fill In The Blank

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

  return __________;
}
```

```
0000000000401126 <if_then>:
  401126:   cmp    $0x6,%edi
  401129:   je     40112f
  40112b:   lea    (%rdi,%rdi,1),%eax
  40112e:   retq
  40112f:   add    $0x1,%edi
  401132:   jmp    40112b
```

Practice: Fill In The Blank

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

   return param1 * 2;
}
```

```
0000000000401126 <if_then>:
  401126:    cmp    $0x6,%edi
  401129:    je     40112f
  40112b:    lea    (%rdi,%rdi,1),%eax
  40112e:    retq
  40112f:    add    $0x1,%edi
  401132:    jmp    40112b
```

Common If-Else Construction

If-Else In C

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

    return result;
}
```

If-Else In Assembly pseudocode

```
Check opposite of code condition
Jump to else-body if test passes
If-body
Jump to past else-body
Else-body
Past else body
```

Practice: Fill in the Blank

If-Else In C

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

```
401134 <+0>:   mov    %rsi,%rax
401137 <+3>:   cmp    %rsi,%rdi
40113a <+6>:   jge    0x401140 <absdiff+12>
40113c <+8>:   sub    %rdi,%rax
40113f <+11>:  retq
401140 <+12>:  sub    %rsi,%rdi
401143 <+15>:  mov    %rdi,%rax
401146 <+18>:  retq
```

If-Else In Assembly pseudocode

```
Check opposite of code condition
Jump to else-body if test passes
If-body
Jump to past else-body
Else-body
Past else body
```

If-Else Construction Variations

C Code

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

    ret++;
    return ret;
}
```

Assembly

```
401134 <+0>:   cmp    $0x3,%edi
401137 <+3>:   jle    0x401142 <test+14>
401139 <+5>:   mov    $0xa,%eax
40113e <+10>:  add    $0x1,%eax
401141 <+13>:  retq
401142 <+14>:  mov    $0x0,%eax
401147 <+19>:  jmp    0x40113e <test+10>
```

Recap

- Assembly Execution and %rip
- Control Flow Mechanics
  - Condition Codes
  - Assembly Instructions
- If Statements

Lecture 18 takeaway: We represent control flow in assembly by storing information in condition codes and having instructions that act differently depending on the condition code values. Conditionals commonly use cmp or test along with jumps to conditionally skip over assembly instructions.