
photo by unsplash user @carrier_lost

# COMP201

## Computer Systems & Programming

Lecture #21 – Assembly Execution and %rip

KOÇ UNIVERSITY

Aykut Erdem // Koç University // Fall 2020

Image: Professor Farnsworth (Futurama)

# Good news, everyone!

- Mid-semester course evaluations are due 23:59, November 23.
- We will finish early today so that you can use the last 10 mins to complete the evaluation form.

# Recap

- The `lea` Instruction
- Logical and Arithmetic Operations
- Practice: Reverse Engineering

# Recap: Unary Instructions

The following instructions operate on a single operand (register or memory):

| Instruction | Effect | Description |
|---|---|---|
| `inc D` | `D ← D + 1` | Increment |
| `dec D` | `D ← D - 1` | Decrement |
| `neg D` | `D ← -D` | Negate |
| `not D` | `D ← ~D` | Complement |

**Examples:**

```
incq 16(%rax)
dec %rdx
not %rcx
```

# Recap: Binary Instructions

The following instructions operate on two operands (both can be register or memory, source can also be immediate). Both cannot be memory locations. Read it as, e.g. "Subtract S from D":

| Instruction | Effect | Description |
|---|---|---|
| `add S, D` | `D ← D + S` | Add |
| `sub S, D` | `D ← D - S` | Subtract |
| `imul S, D` | `D ← D * S` | Multiply |
| `xor S, D` | `D ← D ^ S` | Exclusive-or |
| `or S, D` | `D ← D \| S` | Or |
| `and S, D` | `D ← D & S` | And |

**Examples:**

```
addq %rcx,(%rax)
xorq $16,(%rax, %rdx, 8)
subq %rdx,8(%rax)
```

# Recap: Large Multiplication

- Multiplying 64-bit numbers can produce a 128-bit result. How does x86-64 support this with only 64-bit registers?
- If you specify two operands to **`imul`**, it multiplies them together and truncates until it fits in a 64-bit register.

```
imul S, D      D ← D * S
```

- If you specify one operand, it multiplies that by **`%rax`**, and splits the product across **2** registers. It puts the high-order 64 bits in **`%rdx`** and the low-order 64 bits in **`%rax`**.

| Instruction | Effect | Description |
|---|---|---|
| `imulq S` | `R[%rdx]:R[%rax] ← S x R[%rax]` | Signed full multiply |
| `mulq S` | `R[%rdx]:R[%rax] ← S x R[%rax]` | Unsigned full multiply |

# Recap: Division and Remainder

| Instruction | Effect | Description |
|---|---|---|
| `idivq S` | `R[%rdx] ← R[%rdx]:R[%rax] mod S;`<br>`R[%rax] ← R[%rdx]:R[%rax] ÷ S` | Signed divide |
| `divq S` | `R[%rdx] ← R[%rdx]:R[%rax] mod S;`<br>`R[%rax] ← R[%rdx]:R[%rax] ÷ S` | Unsigned divide |
| `cqto` | `R[%rdx]:R[%rax] ← SignExtend(R[%rax])` | Convert to oct word |

- Terminology: **dividend / divisor = quotient + remainder**
- The high-order 64 bits of the dividend are in **`%rdx`**, and the low-order 64 bits are in **`%rax`**. The divisor is the operand to the instruction.
- Most division uses only 64-bit dividends. The **`cqto`** instruction sign-extends the 64-bit value in **`%rax`** into **`%rdx`** to fill both registers with the dividend, as the division instruction expects.

# Recap: Shift Instructions

The following instructions have two operands: the shift amount **k** and the destination to shift, **D**. **k** can be either an immediate value, or the byte register **`%cl`** (and only that register!)

| Instruction | Effect | Description |
|---|---|---|
| `sal k, D` | D ← D << k | Left shift |
| `shl k, D` | D ← D << k | Left shift (same as `sal`) |
| `sar k, D` | D ← D $>>_A$ k | Arithmetic right shift |
| `shr k, D` | D ← D $>>_L$ k | Logical right shift |

**Examples**:

```
shll $3,(%rax)
shrl %cl,(%rax,%rdx,8)
sarl $4,8(%rax)
```

# Lecture Plan

- Practice: Reverse Engineering
- Assembly Execution and `%rip`

**Disclaimer:** Slides for this lecture were borrowed from

—Nick Troccoli's Stanford CS107 class

# Lecture Plan

- Practice: Reverse Engineering
- Assembly Execution and `%rip`

# Recap: Assembly Exercise 1

```
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;
}
```

# Assembly Exercise 2

slido **Event code: 73165**

```
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;
}
```

What location or value in the assembly above represents the C code's **sum** variable?

**%eax**

# Assembly Exercise 3

slido Event code: 73165

```
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;
}
```

What location or value in the assembly code above represents the C code's **6** (as in **`arr[6]`**)?

**0x18**

# Our First Assembly

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

We're 1/2 of the way to understanding assembly!
**What looks understandable right now?**

```
00000000004005b6 <sum_array>:
  4005b6:    ba 00 00 00 00        mov    $0x0,%edx
  4005bb:    b8 00 00 00 00        mov    $0x0,%eax
  4005c0:    eb 09                 jmp    4005cb <sum_array+0x15>
  4005c2:    48 63 ca              movslq %edx,%rcx
  4005c5:    03 04 8f              add    (%rdi,%rcx,4),%eax
  4005c8:    83 c2 01              add    $0x1,%edx
  4005cb:    39 f2                 cmp    %esi,%edx
  4005cd:    7c f3                 jl     4005c2 <sum_array+0xc>
  4005cf:    f3 c3                 repz retq
```

# A Note About Operand Forms

- Many instructions share the same address operand forms that **`mov`** uses.
  - Eg. `7(%rax, %rcx, 2)`.
- These forms work the same way for other instructions, e.g. **`sub`**:
  - `sub 8(%rax,%rdx),%rcx` -> Go to 8 + `%rax` + `%rdx`, subtract what's there from `%rcx`
- The exception is **`lea`**:
  - It interprets this form as just the calculation, *not the dereferencing*
  - `lea 8(%rax,%rdx),%rcx` -> Calculate 8 + `%rax` + `%rdx`, put it in `%rcx`

# Extra Practice

https://godbolt.org/z/QQj77g

# Reverse Engineering 1

```
int add_to(int x, int arr[], int i) {
    int sum = ___?___;
    sum += arr[___?___];
    return ___?___;
}

----------

add_to:
  movslq %edx, %rdx
  movl %edi, %eax
  addl (%rsi,%rdx,4), %eax
  ret
```

# Reverse Engineering 1

```
int add_to(int x, int arr[], int i) {
    int sum = ___?___;
    sum += arr[___?___];
    return ___?___;
}

----------
// x in %edi, arr in %rsi, i in %edx
add_to:
  movslq %edx, %rdx           // sign-extend i into full register
  movl %edi, %eax             // copy x into %eax
  addl (%rsi,%rdx,4), %eax    // add arr[i] to %eax
  ret
```

# Reverse Engineering 1

```
int add_to(int x, int arr[], int i) {
    int sum = x;
    sum += arr[i];
    return sum;
}

----------
// x in %edi, arr in %rsi, i in %edx
add_to_ith:
  movslq %edx, %rdx           // sign-extend i into full register
  movl %edi, %eax             // copy x into %eax
  addl (%rsi,%rdx,4), %eax    // add arr[i] to %eax
  ret
```

# Reverse Engineering 2

```
int elem_arithmetic(int nums[], int y) {
    int z = nums[___?___] * ___?___;
    z -= ___?___;
    z >>= ___?___;
    return ___?___;
}
----------

elem_arithmetic:
  movl %esi, %eax
  imull (%rdi), %eax
  subl 4(%rdi), %eax
  sarl $2, %eax
  addl $2, %eax
  ret
```

# Reverse Engineering 2

```
int elem_arithmetic(int nums[], int y) {
    int z = nums[___?___] * ___?___;
    z -= ___?___;
    z >>= ___?___;
    return ___?___;
}
----------
// nums in %rdi, y in %esi
elem_arithmetic:
  movl %esi, %eax            // copy y into %eax
  imull (%rdi), %eax         // multiply %eax by nums[0]
  subl 4(%rdi), %eax         // subtract nums[1] from %eax
  sarl $2, %eax              // shift %eax right by 2
  addl $2, %eax              // add 2 to %eax
  ret
```

# Reverse Engineering 2

```
int elem_arithmetic(int nums[], int y) {
    int z = nums[0] * y;
    z -= nums[1];
    z >>= 2;
    return z + 2;
}
----------
// nums in %rdi, y in %esi
elem_arithmetic:
  movl %esi, %eax           // copy y into %eax
  imull (%rdi), %eax        // multiply %eax by nums[0]
  subl 4(%rdi), %eax        // subtract nums[1] from %eax
  sarl $2, %eax             // shift %eax right by 2
  addl $2, %eax             // add 2 to %eax
  ret
```

# Reverse Engineering 3

```
long func(long x, long *ptr) {
    *ptr = ___?___ + 1;
    long result = x % ___?___;
    return ___?___;
}
----------

func:
  leaq 1(%rdi), %rcx
  movq %rcx, (%rsi)
  movq %rdi, %rax
  cqto
  idivq %rcx
  movq %rdx, %rax
  ret
```

# Reverse Engineering 3

```
long func(long x, long *ptr) {
    *ptr = ___?___ + 1;
    long result = x % ___?___;
    return ___?___;
}
----------
// x in %rdi, ptr in %rsi
func:
  leaq 1(%rdi), %rcx        // put x + 1 into %rcx
  movq %rcx, (%rsi)         // copy %rcx into *ptr
  movq %rdi, %rax           // copy x into %rax
  cqto                      // sign-extend x into %rdx
  idivq %rcx                // calculate x / (x + 1)
  movq %rdx, %rax           // copy the remainder into %rax
  ret
```

# Reverse Engineering 3

```
long func(long x, long *ptr) {
    *ptr = x + 1;
    long result = x % *ptr; // or x + 1
    return result;
}
----------
// x in %rdi, ptr in %rsi
func:
  leaq 1(%rdi), %rcx        // put x + 1 into %rcx
  movq %rcx, (%rsi)         // copy %rcx into *ptr
  movq %rdi, %rax           // copy x into %rax
  cqto                      // sign-extend x into %rdx
  idivq %rcx                // calculate x / (x + 1)
  movq %rdx, %rax           // copy the remainder into %rax
  ret
```

# Lecture Plan

- More practice: Reverse Engineering
- Assembly Execution and `%rip`

# Learning Assembly

| Moving data around | → | Arithmetic and logical operations | → | Control flow | → | Function calls |
|---|---|---|---|---|---|---|
| Lecture 19 | | Lecture20 | | This week | | Lecture 24-26 |

# Learning Goals

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

# 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` |

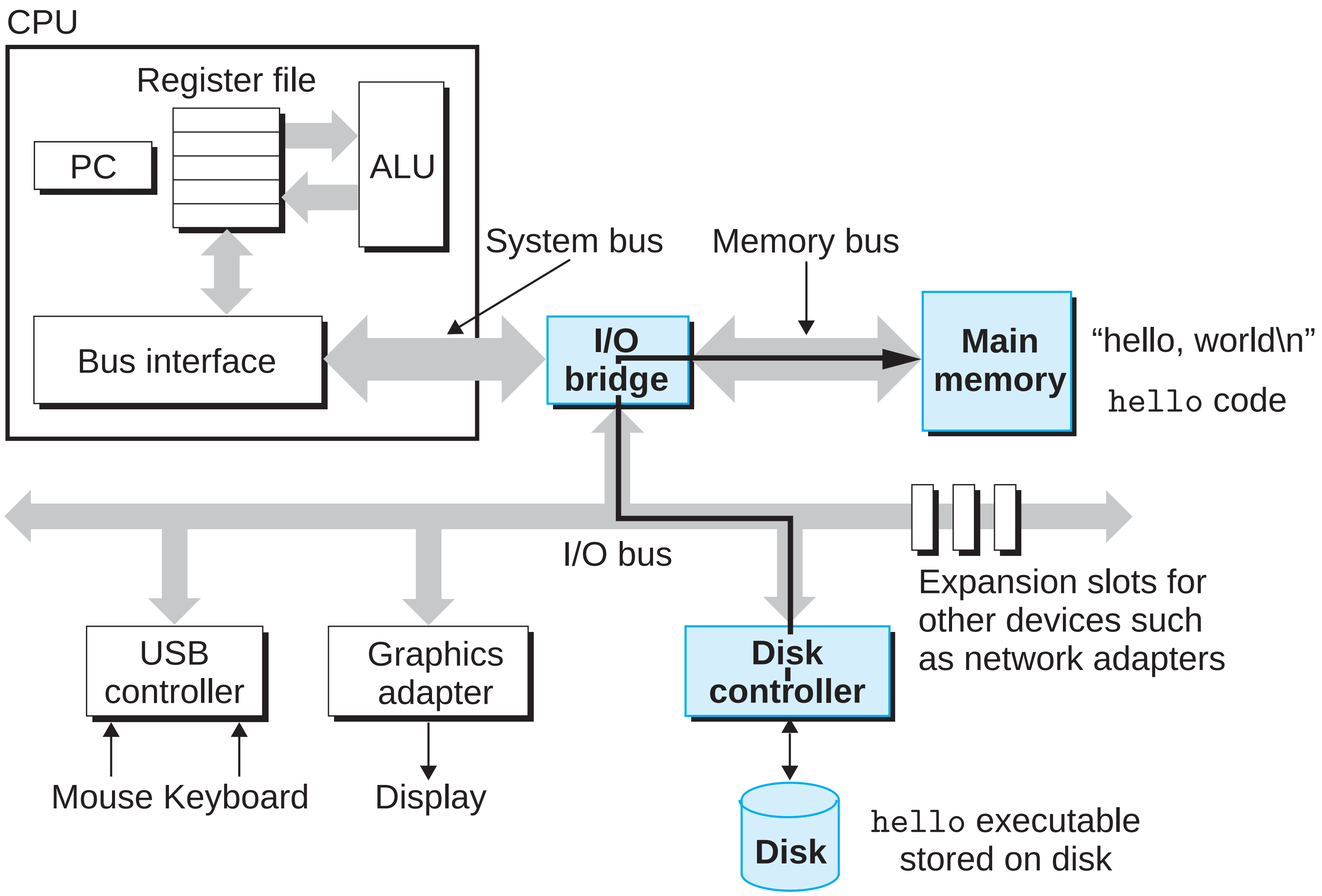
# Register Responsibilities

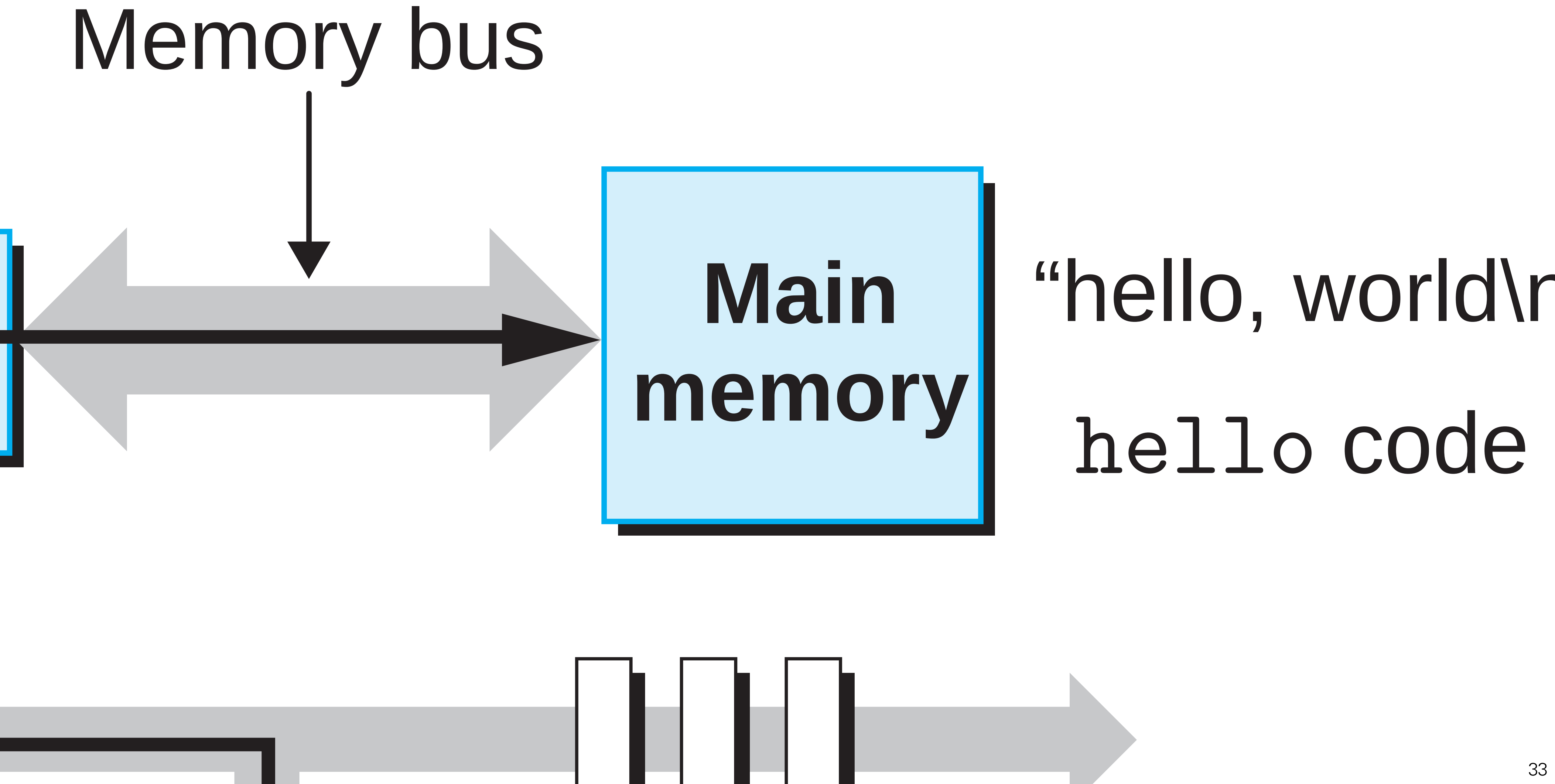
Some registers take on special responsibilities during program execution.

- **`%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

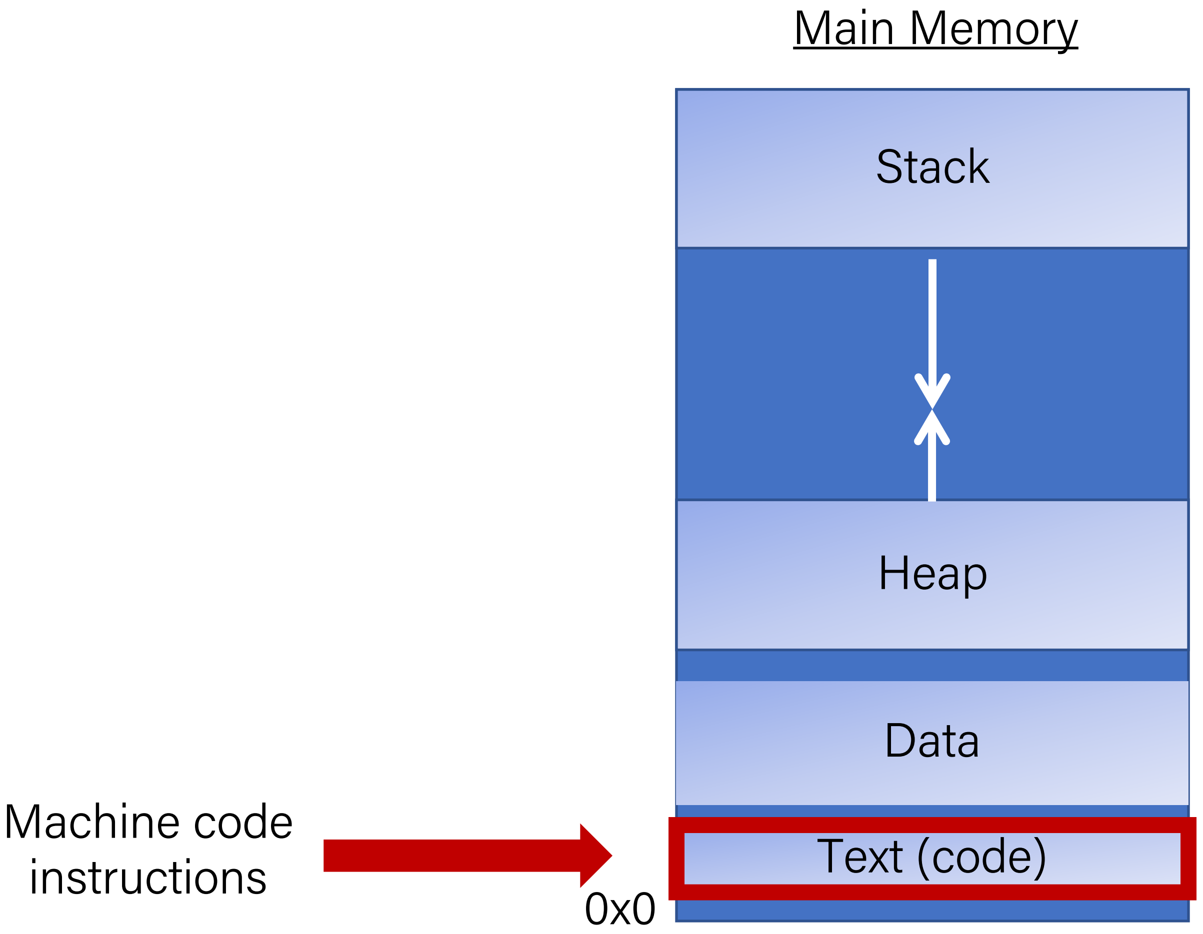
See the x86-64 Guide and Reference Sheet on the Resources webpage for more!

# Instructions Are Just Bytes!




Memory bus
Main memory
“hello, world\
hello code

# Instructions Are Just Bytes!



# %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 |

# %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.

0x4004ed

%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.

`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>
```

| | |
|---|---|
| 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.

`0x4004f1`

`%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>
```

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

`0x4004f8`

`%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.

`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>
```

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

| | |
|---|---|
| 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 |

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!

```
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 **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>
```

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

| | |
|---|---|
| 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 |

| |
|---|
| 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>
```

This assembly represents an infinite loop in C!

```
while (true) {…}
```

| | |
|---|---|
| 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 |

| 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**?

# Recap:

- More practice: Reverse Engineering
- Assembly Execution and `%rip`

**Next time:** Condition codes, conditional branches