CS107 Lecture 17
Assembly: Arithmetic and Logic Wrap, Control Flow

Reading: B&O 3.5-3.6

Multiplication Support

Multiplying 64-bit numbers can produce **128-bit results**. How does x86-64 support this with only 64-bit registers?

- If you specify **two operands** to **`imul`** or **`mul`**, it multiplies them and truncates the result to fit in the second of the two.

`imul S, D` is realized as `D ← D * S`

- If you specify one operand, it's multiplied by **`%rax`** and the product is split across **two** registers: the high-order 64 bits go 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

Division and Mod Support

`x86-64` supports **dividing a 128-bit value by a 64-bit value**.

Instruction	Effect	Description

`idivq S`	`R[%rdx] ← R[%rdx]:R[%rax] mod S;`
	`R[%rax] ← R[%rdx]:R[%rax] ÷ S`
		Signed divide

`divq S`	`R[%rdx] ← R[%rdx]:R[%rax] mod S;`
	`R[%rax] ← R[%rdx]:R[%rax] ÷ S`
		Unsigned divide

`cqto`	`R[%rdx]:R[%rax] ← SignExtend(R[%rax])`	Convert to oct word

- The **high-order 64 bits of the dividend** need to be prepared and stored in **`%rdx`**, the **low-order 64 bits** in **`%rax`**. The divisor is the only listed operand.
- The **integer quotient** is stored in **`%rax`**, and the **remainder** in **`%rdx`**.
- Most dividend are just 64-bit. The **`cqto`** instruction **sign-extends the 64-bit** value in **`%rax`** though **`%rdx`** to fill **both registers with the dividend**, as **`idiv`** and **`div`** expect.

Reverse Engineering Etude

Examine the **assembly code** emitted on behalf of the three-argument **`div_and_mod`** function and **reconstruct an equivalent, four-line implementation** in C.

Assembly:

```
div_and_mod:
   movq %rdi, %rax
   movq %rdx, %rcx
   cqto
   idivq %rsi
   movq %rdx, (%rcx)
   ret
```

C:

```
long div_and_mod(long x, long y, long *p_mod) {
   long quotient = x / y;
   long remainder = x % y;
   *p_mod = remainder;
   return quotient;
}
```

Key Insights:

- The **first and third lines of assembly** suggest **`x`**—delivered via **`%rdi`**—is the **dividend** of the forthcoming division.
- The **second line** stores a copy of **`p_mod`** (courtesy of **`%rdx`**) in **`%rcx`**.
- The **fourth line** divides **`x`** by **`y`** (supplied via **`%rsi`**), populating **`%rax`** with the **integer quotient** and **`%rdx`** with the **remainder**.
  - This explains why a **copy** of **`p_mod`** was placed in **`%rcx`**—the **compiler recognized** **`%rdx`** would be **overwritten** by **`idivq`** two lines later.
- The **remainder is written through the address stored in `%rcx`**, suggesting **`*p_mod = x % y`**
- **`%rax`** still holds **the integer quotient when the function returns**, suggesting something akin to **`return x / y`**

Reverse Engineering Etude 1

Examine the **assembly code** emitted on behalf of the three-argument **`div_and_mod`** function and **reconstruct an equivalent, four-line implementation** in C.

Assembly:

```
div_and_mod:
   movq %rdi, %rax
   movq %rdx, %rcx
   cqto
   idivq %rsi
   movq %rdx, (%rcx)
   ret
```

C:

```
long div_and_mod(long x, long y, long *p_mod) {
   long quotient = x / y;
   long remainder = x % y;
   *p_mod = remainder;
   return quotient;
}
```

Key Insights:

- The **first and third lines of assembly** suggest **`x`**—delivered via **`%rdi`**—is the **dividend** of the forthcoming division.
- The **second line** stores a copy of **`p_mod`** (courtesy of **`%rdx`**) in **`%rcx`**.
- The **fourth line** divides **`x`** by **`y`** (supplied via **`%rsi`**), populating **`%rax`** with the **integer quotient** and **`%rdx`** with the **remainder**.
  - This explains why a **copy** of **`p_mod`** was placed in **`%rcx`**—the **compiler recognized** **`%rdx`** would be **overwritten** by **`idivq`** two lines later.
- The **remainder is written through the address stored in `%rcx`**, suggesting **`*p_mod = x % y`**
- **`%rax`** still holds **the integer quotient when the function returns**, suggesting something akin to **`return x / y`**

Reverse Engineering Etude

Examine the **assembly code** emitted on behalf of the three-argument **`tinker_toy`** function and **complete the implementation**.

Assembly:

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

C:

```
int tinker_toy(int x, int arr[], int y) {
    int sum = _____x_____;
    sum += arr[_____y_____];
    return ____sum____;
}
```

Key Insights:

- Because the third argument **y** is an **`int`**, it's really passed in **`%edx`**, and the **upper half** of **`%rdx`** is **irrelevant garbage**.
- However, a **sign-extended copy** of **y** is placed in **`%rdx`** via the first instruction, since the **addressing mode of the third line requires full registers**. **`%edx`** itself doesn't qualify.
- Because **`sum`** is declared as an **`int`** and is the **focus of the implementation**, it's reasonable to assume **`sum`** is backed by **`%eax`** and that its **final value is what's returned**.
- The second line **initializes** **`sum`** to **`x`**, and the third **adds** **`arr[y]`** to **`sum`**. Note the **scale factor** in line three is a 4, and that's what you'd expect with **`int *`** pointer arithmetic.

Executing Instructions

So far:

- Program values **can be stored in memory or in registers**.
- Assembly instructions **read and write values** back and forth between **registers and main memory**.
- Assembly instructions are **also stored in memory**.

Big Questions:

- **What controls execution flow**? How does a process know what instruction to **execute next**?

Answer:

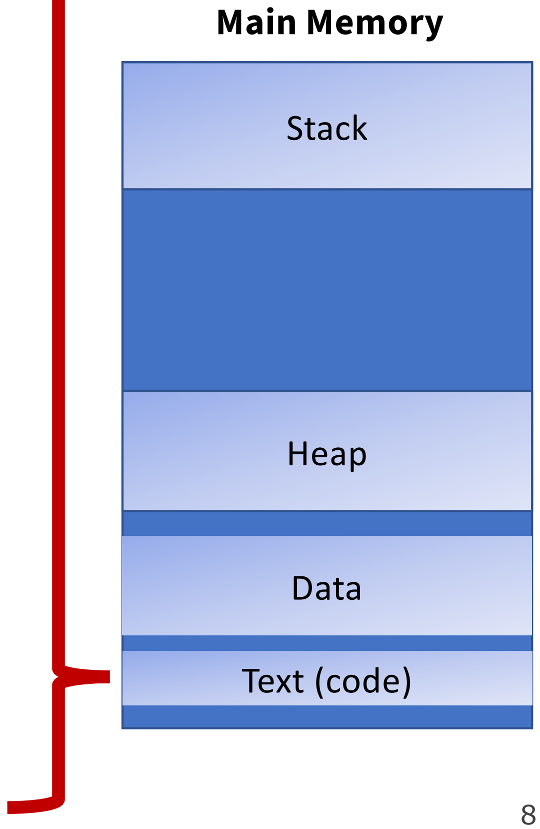
- The **program counter**, stored in `%rip`.

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

Followin

```
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>
```

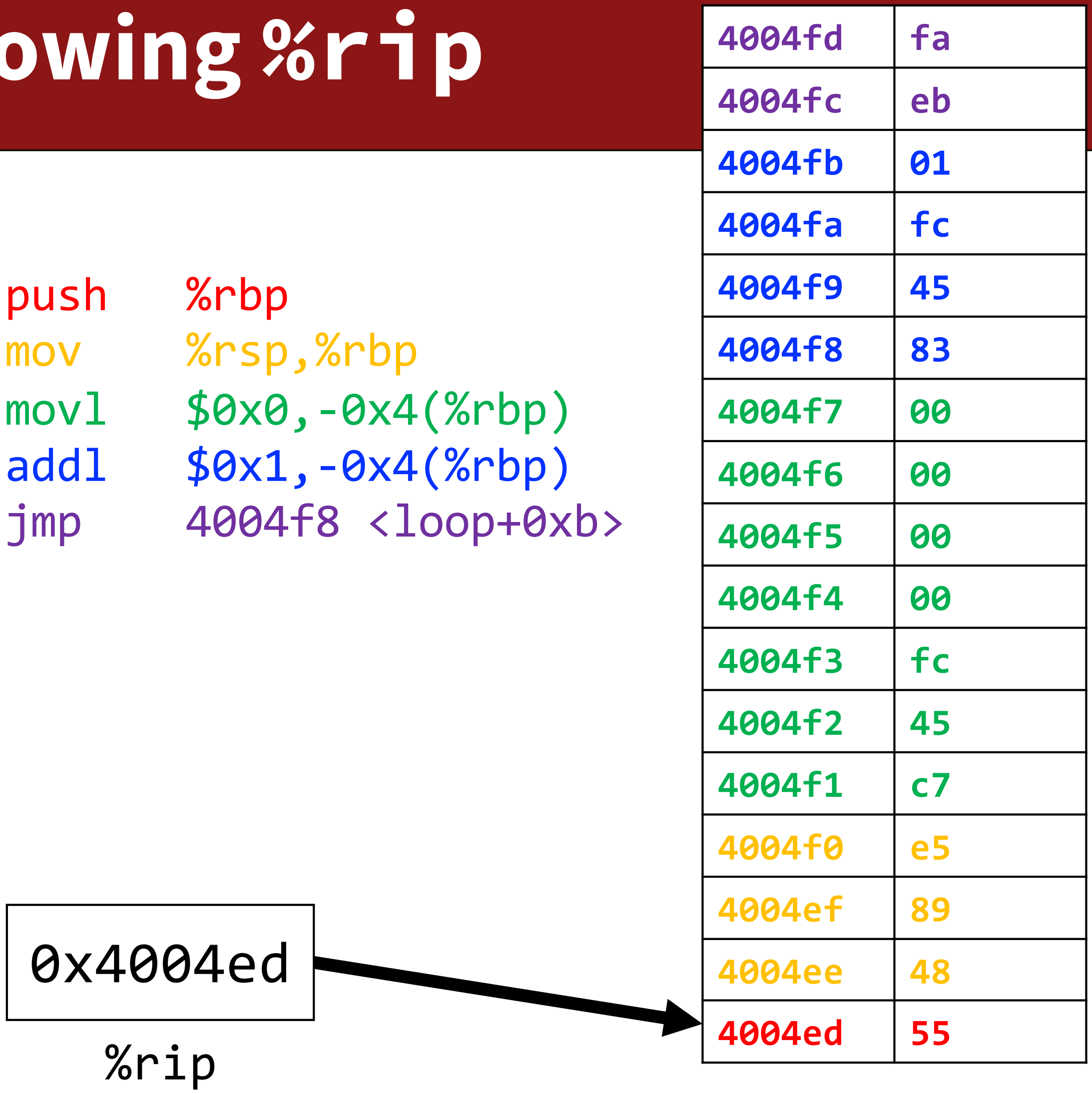
4004fd	fa
4004fc	eb
4004fb	01
4004fa	fc
4004f9	45
4004f8	83
4004f7	00
4004f6	00
4004f5	00
4004f4	00
4004f3	fc
4004f2	45
4004f1	c7
4004f0	e5
4004ef	89
4004ee	48
4004ed	55



Following `%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>
```

In x86-64, **`%rip`** serves as the **program counter** and holds the address of the **next instruction to be executed**.



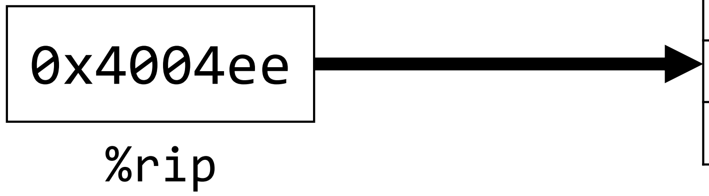
4004fd	fa
4004fc	eb
4004fb	01
4004fa	fc
4004f9	45
4004f8	83
4004f7	00
4004f6	00
4004f5	00
4004f4	00
4004f3	fc
4004f2	45
4004f1	c7
4004f0	e5
4004ef	89
4004ee	48
4004ed	55

Following `%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>
```

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

In x86-64, **`%rip`** serves as the **program counter** and holds the address of the **next instruction to be executed**.

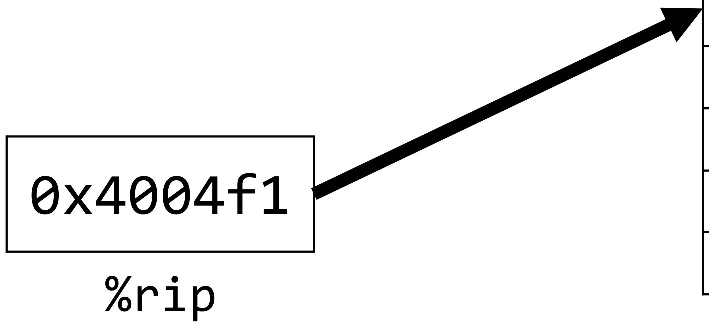


Following %rip

```
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

In x86-64, **%rip** serves as the **program counter** and holds the address of the **next instruction to be executed**.



Following %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

In x86-64, **`%rip`** serves as the **program counter** and holds the address of the **next instruction to be executed**.

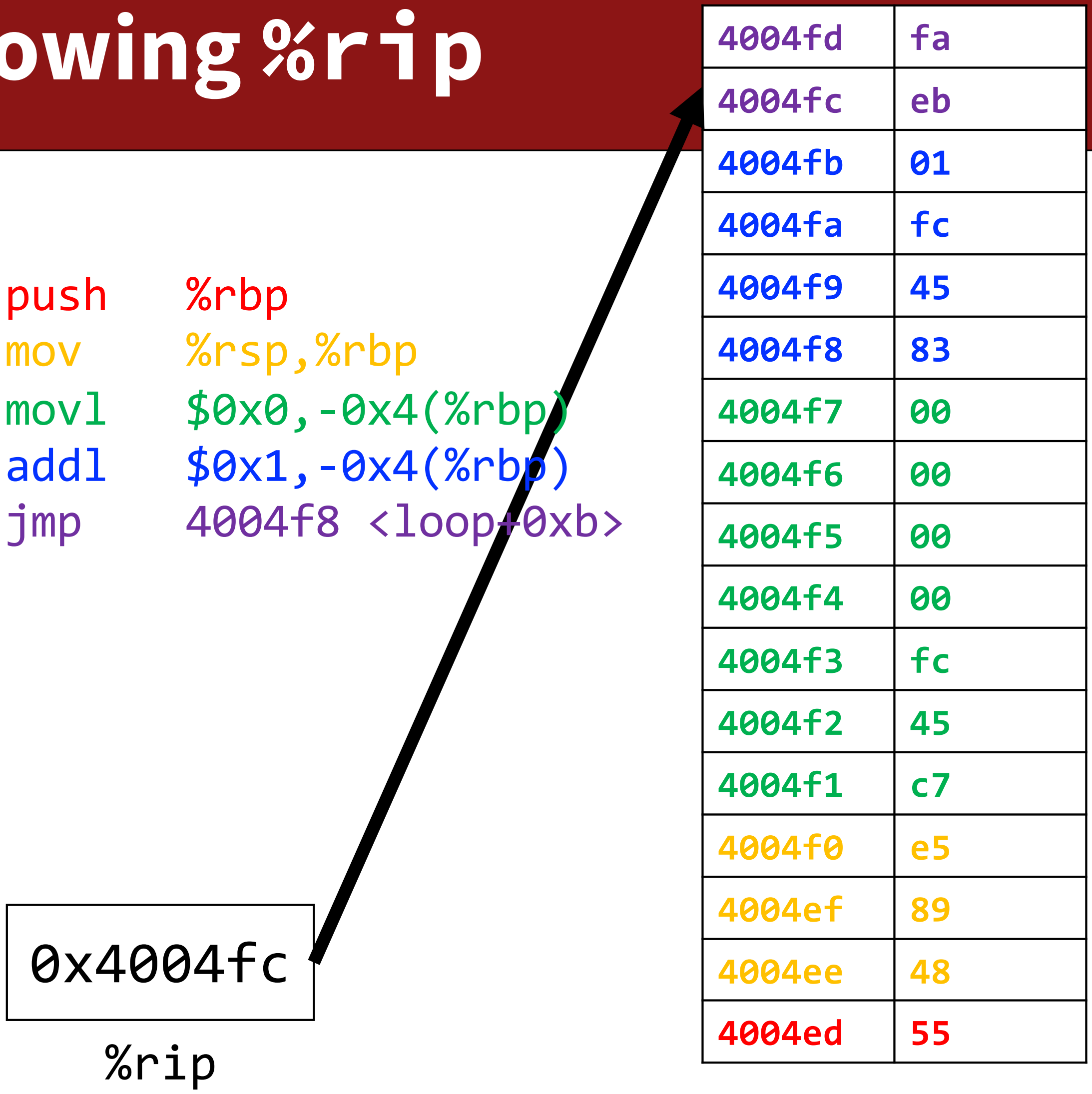
```
0x4004f8
```

%rip

Following %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>
```

In x86-64, **`%rip`** serves as the **program counter** and holds the address of the **next instruction to be executed**.



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

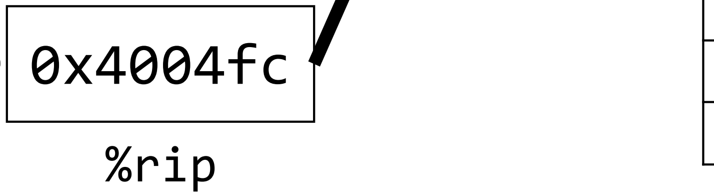
Following %rip

```
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

Normally, **dedicated hardware** sets the program counter to the **address of the next instruction**:

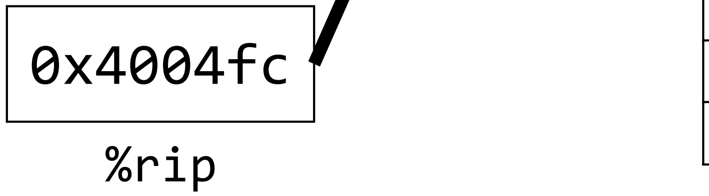
`%rip` += current instruction size



Interrupting Control Flow

```
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).

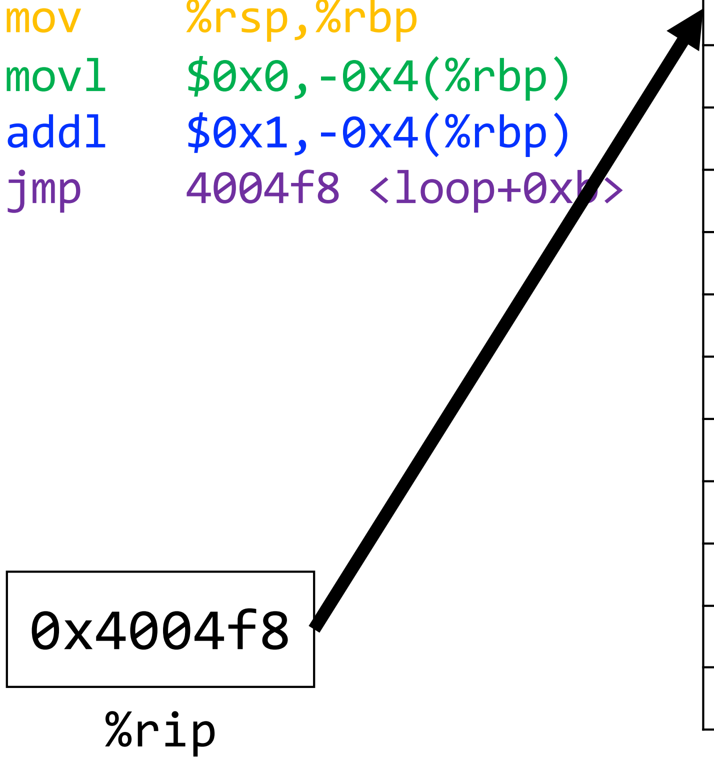


4004fd	fa
4004fc	eb
4004fb	01
4004fa	fc
4004f9	45
4004f8	83
4004f7	00
4004f6	00
4004f5	00
4004f4	00
4004f3	fc
4004f2	45
4004f1	c7
4004f0	e5
4004ef	89
4004ee	48
4004ed	55

Interrupting Control Flow

```
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).

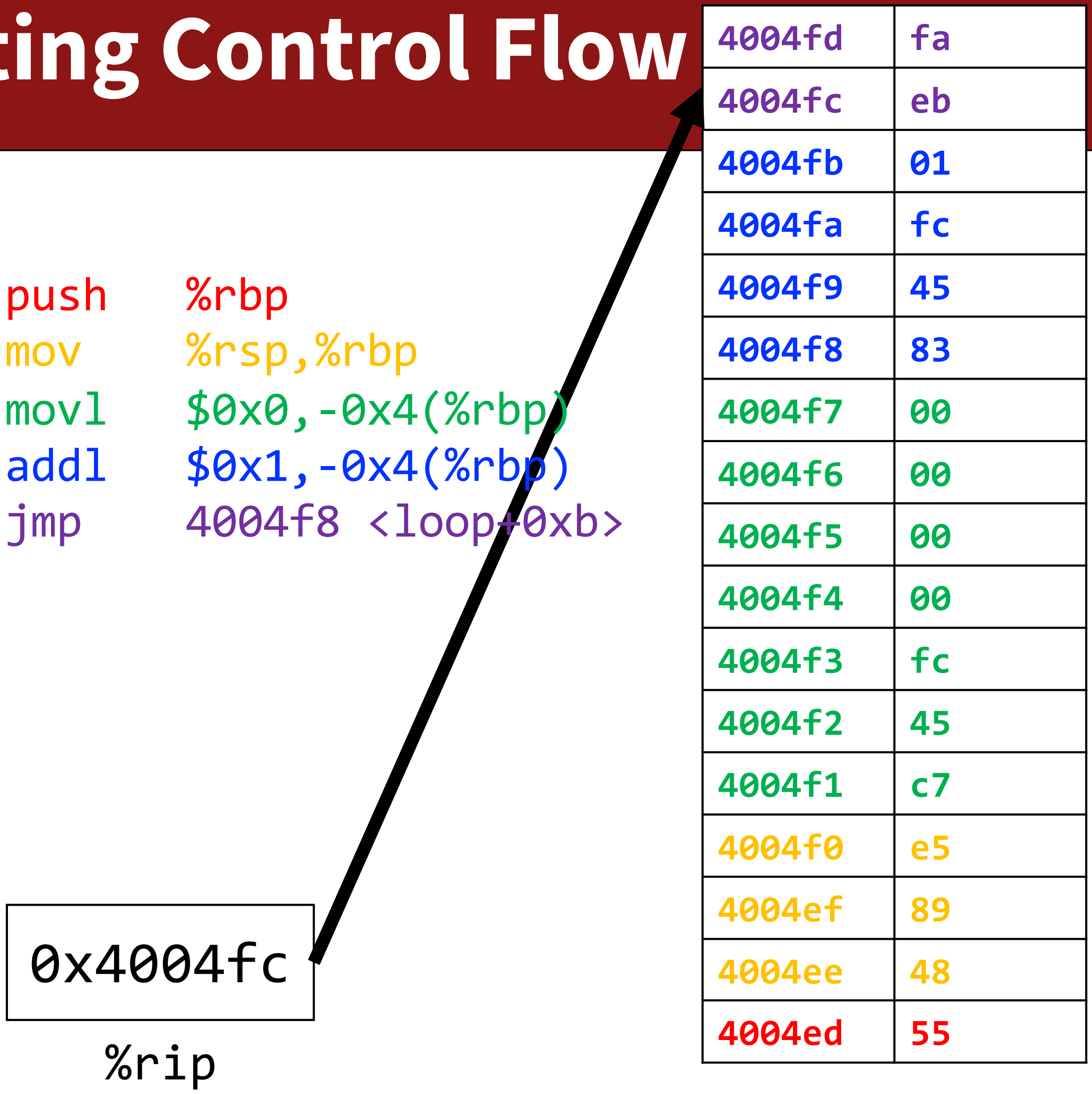


4004fd	fa
4004fc	eb
4004fb	01
4004fa	fc
4004f9	45
4004f8	83
4004f7	00
4004f6	00
4004f5	00
4004f4	00
4004f3	fc
4004f2	45
4004f1	c7
4004f0	e5
4004ef	89
4004ee	48
4004ed	55

Interrupting Control Flow

```
0000000000004004ed <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

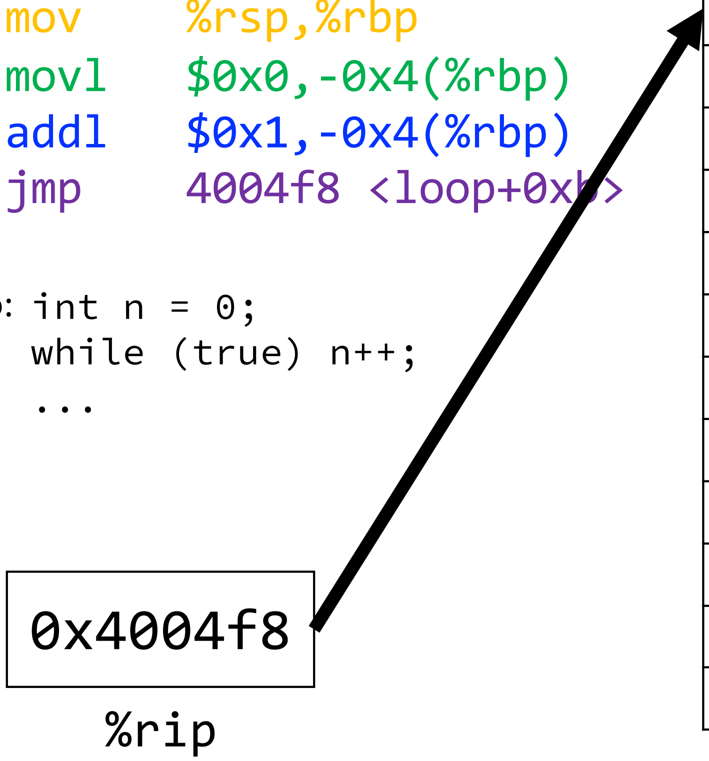
Interrupting Control Flow

```
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 above might reverse compile to:

```
int n = 0;
while (true) n++;
...
```

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

The `jmp` Instruction

The **`jmp`** instruction jumps to another instruction in the assembly code—an **unconditional jump**.

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

The single operand can be **encoded** directly into the instruction as a **direct jump**:

```
    jmp 404f8 <loop+0xb>
```

The operand can be expressed as an **indirect jump** using one of the **many addressing modes**

```
    jmp *%rax
```

Aside: **Direct, unconditional jumps** are frequently used for **loopbacks** in **for** and **while** loops and for **skipping** around **else** blocks.

Another Aside: **Indirect unconditional jumps** are much less common, used on behalf of very large **switch** statements and function pointers.

Next Question: What if we want to jump **conditionally**?