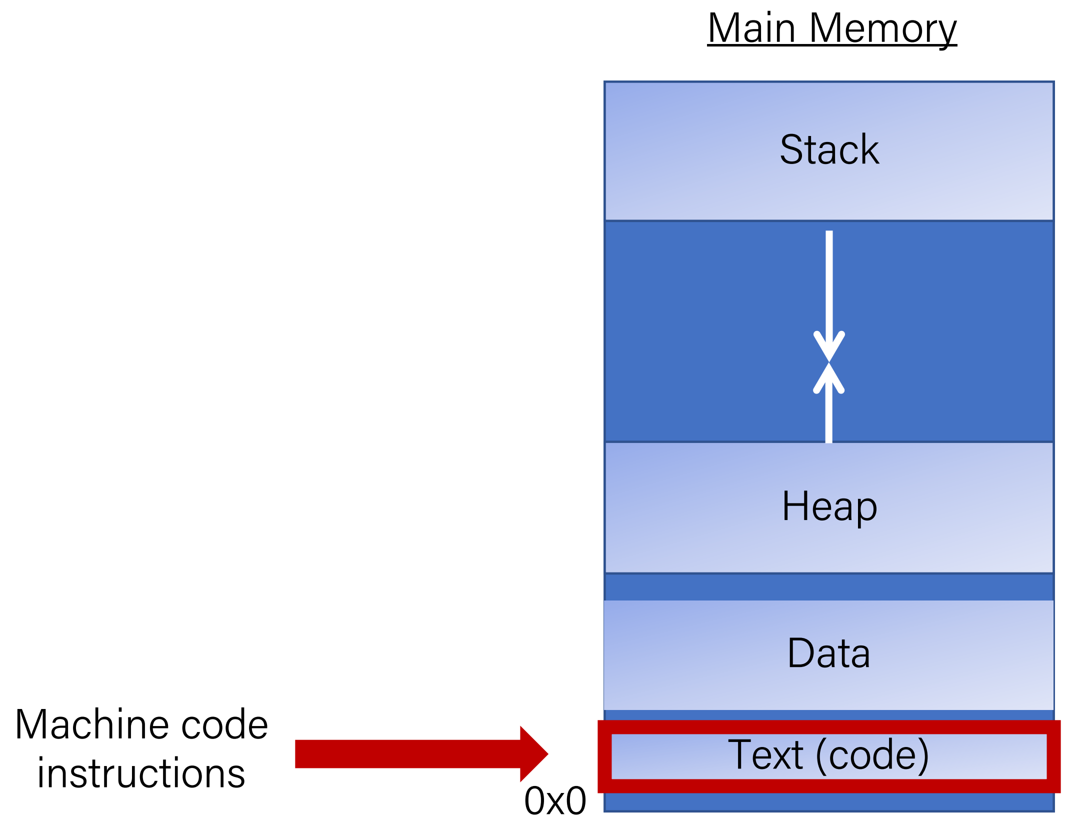
Instructions Are Just Bytes!

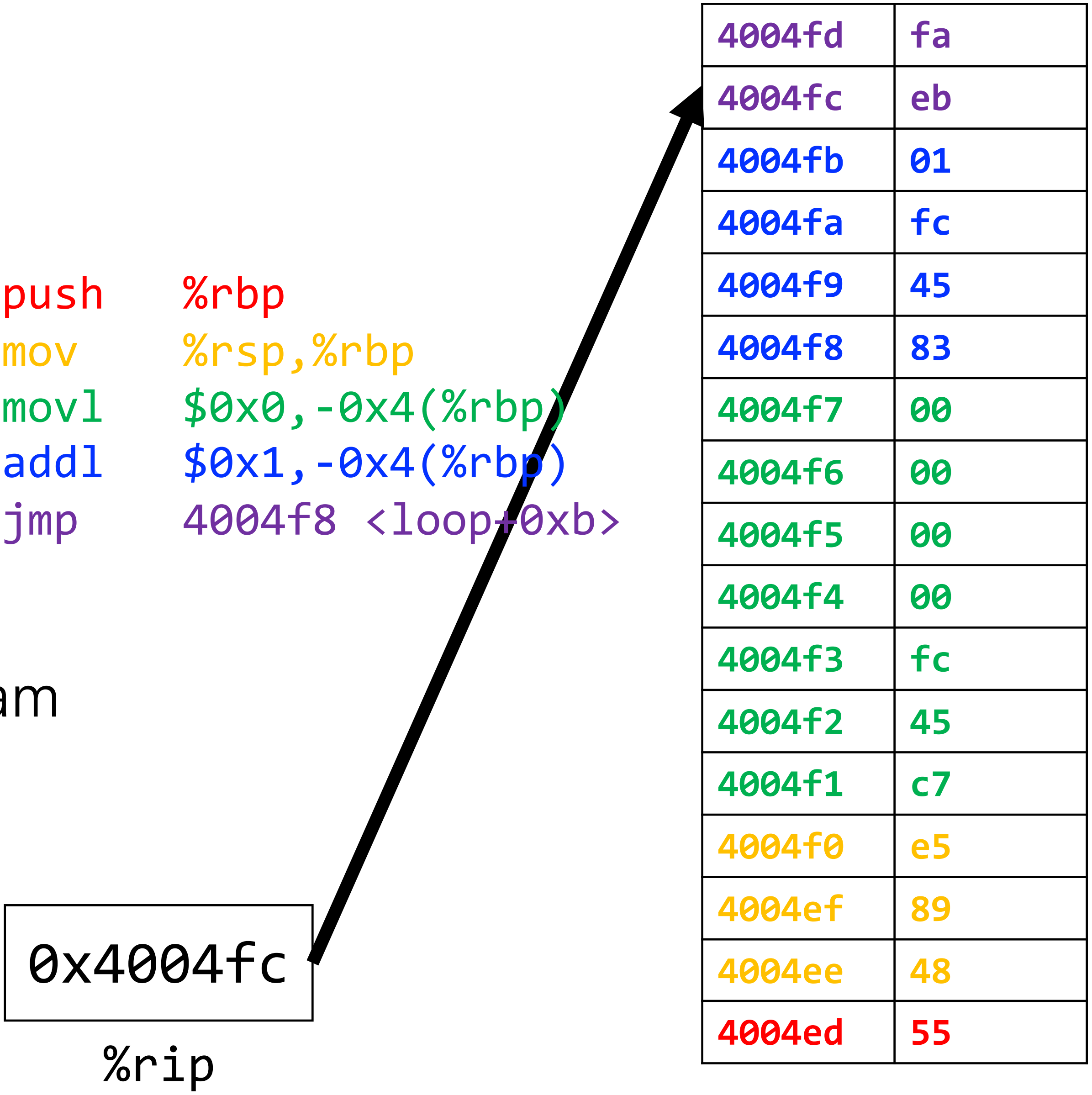


%rip

```
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>
```

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

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
```

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

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

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

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

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 != 3
// calculates %edi – 3
cmp $3, %edi
jne [target]
```

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

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

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 **`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!

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.