Computer Systems

CS107

Cynthia Lee

Today's Topics

LECTURE:

- More assembly code!
  - Arithmetic and logic operations
  - If-else control
  - Loops (to be continued Friday)

Basic addressing modes (Think: assembly version of VARIABLES)

Op	Source	Dest	Dest Comments
`movl`	`$0,`	`%eax`	Name of a register
`movl`	`$0,`	`0x8f2713e0`	Actual address literal (note address literals are different from other literals—don't need $ in front)
`movl`	`$0,`	`(%rax)`	Look in the register named, find an address there, and use it
`movl`	`$0,`	`-24(%rbp)`	Add -24 to an address in the named register, and use that address
`movl`	`$0`	`8(%rbp, %eax, 2)`	Address to use = (8 + address in rbp) + (2 * index in eax)

Arithmetic and bitwise operations

We've moved data all around, now how do we actually do calculations?

Arithmetic and bitwise operations template

Op	Source1	Source2/Dest	Dest Comments
add	op1	op2	op2 += op1

- op1 and op2 can be any of the addressing modes we've seen
  - though, again, at most one of op1 and op2 can touch memory
  - note we can get a sneaky 2-in-1 memory access if we use a memory location for op2, because it is a source and destination

Arithmetic and bitwise operations

```
add src, dst                    # dst += src
sub src, dst                    # dst -= src
imul src, dst                   # dst *= src
inc dst                         # dst += 1
dec dst                         # dst -= 1
neg dst                         # dst = -dst

and src, dst                    # dst &= src
or src, dst                     # dst |= src
xor src, dst                    # dst ^= src
not dst                         # dst = ~dst

shl count, dst                  # dst <<= count (left shift)
sar count, dst                  # dst >>= count (arithmetic right shift)
shr count, dst                  # dst >>= count (logical right shift)
```

Special-format arithmetic operations

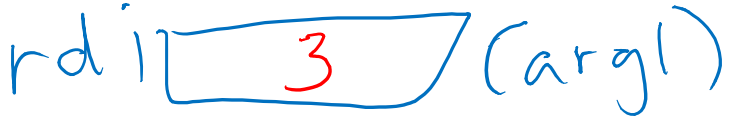
```
# single operand imul assumes other operand in %rax, computes
# 128-bit result and stores high 64-bits in %rdx, low 64-bits
# in %rax

imul src

# dst <<= 1  (no count => assume 1, same for sar, shr, sal)

shl dst
```

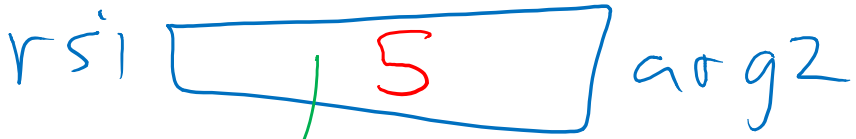
Register uses (includes a few of the most common—for more complete list see reference on course website)

Register	Conventional use	Low 32-bits	Low 16-bits	Low 8-bits
%rax	Return value	%eax	%ax	%al
%rdi	1st argument	%edi	%di	%dil
%rsi	2nd argument	%esi	%si	%sil
%rdx	3rd argument	%edx	%dx	%dl
%r10	Scratch/temporary	%r10d	%r10w	%r10b
%r11	Scratch/temporary	%r11d	%r11w	%r11b
%rip	Instruction pointer			
%rflags	Status/condition code bits			

Preparing for assign5: reading assembly code

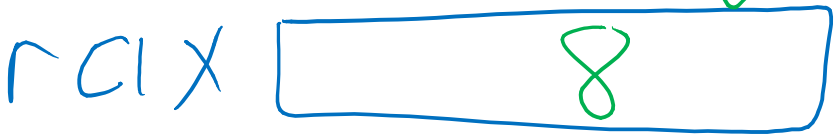
```
0000000004005ac <sum_example1>:
  4005bd:       8b 45 e8          mov    %esi,%eax
  4005c3:       01 d0             add    %edi,%eax
  4005cc:       c3                retq
```






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

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

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

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



Preparing for assign5: reading assembly code

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

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

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

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

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

Preparing for assign5: reading assembly code

```
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;
}
```

Which register or memory address represents the C variable **sum**?

(a) 0xc(%rdi)
(b) %rdi
(c) (%rdi)
(d) 0x18(%rdi)
(e) %eax

Preparing for assign5: reading assembly code

```
0000000000400578 <sum_example2>:
  400578:       8b 47 0c
  40057b:       03 07
  40057d:       2b 47 18
  400580:       c3
```

```
mov     0xc(%rdi),%eax
add     (%rdi),%eax
sub     0x18(%rdi),%eax
retq
```

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

    return sum;
}
```

Which register or memory address represents the C value **6** (as in arr[6])?

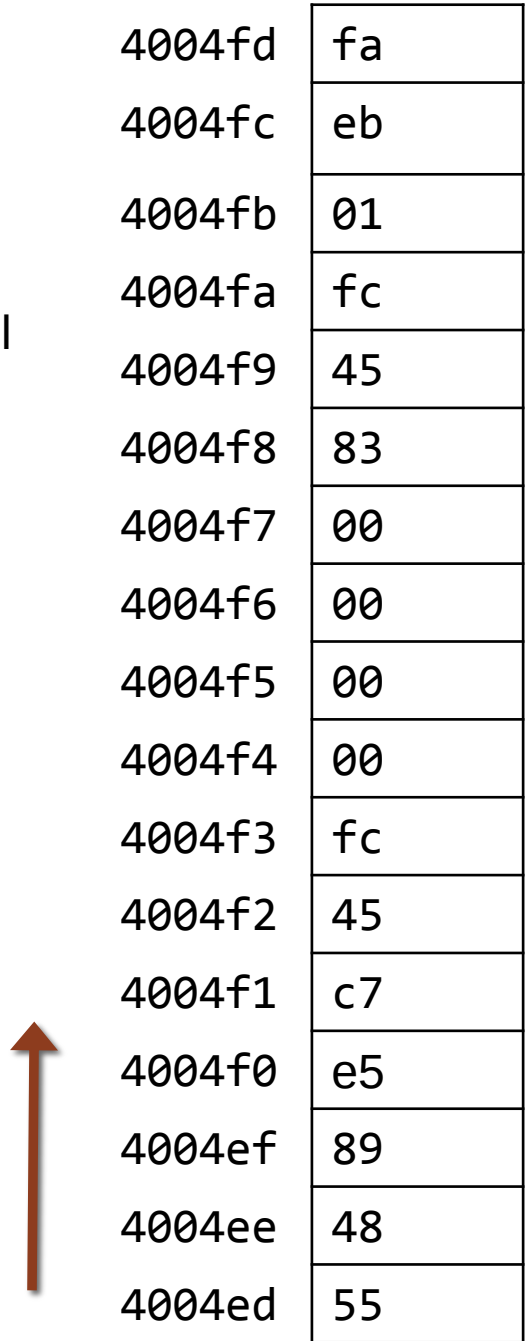
(a) 0xc
(b) %rdi
(c) (%rdi)
(d) 0x18
(e) %eax

What does it mean for a program to execute?

How do we move from one instruction to the next?
How do computers "do stuff"?

Data storage vs. “doing stuff” on a computer

- Data sits in memory (and, we now know, registers, when it is about to be operated on by the CPU)
- We understand that there are instructions that control movement of the data and operations on it
- …
- **But who controls the instructions? How do we know what to do now?**
- **…or do next?**



Progress through a function

```
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>
```

Progress through a function

```
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>
```

Progress through a function

```
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>
```

Progress through a function

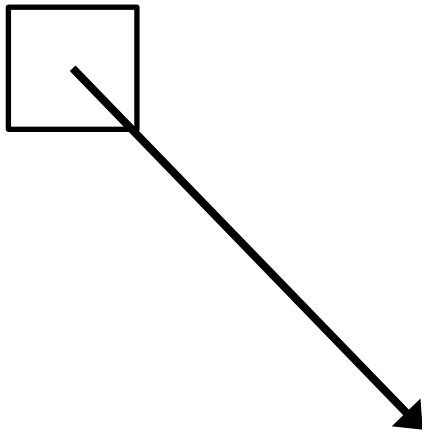
```
0000000004004ed <loop>:
  4004ed:       55                          push   %rbp
  4004ee:       48 89 e5                    mov    %rsp,%rbp
  4004f1:       c7 45 fc 00 00 00 00        movl   $0x0,-0x4(%rbp)
  4004f8:       83 45 fc 01                 addl   $0x1,-0x4(%rbp)
  4004fc:       eb fa                       jmp    4004f8 <loop+0xb>
```

Progress through a function

```
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>
```

Progress through a function

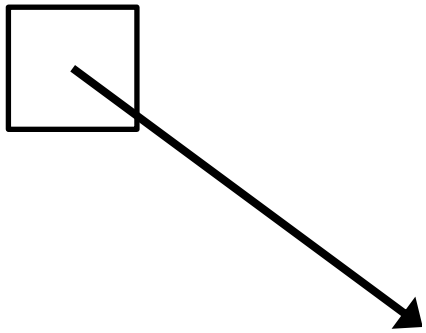
```
0000000000004004ed <loop>:
  4004ed:       55                      push
  4004ee:       48 89 e5                mov
  4004f1:       c7 45 fc 00 00 00 00    movl
  4004f8:       83 45 fc 01             addl
  4004fc:       eb fa                   jmp
```



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

Progress through a function

```
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

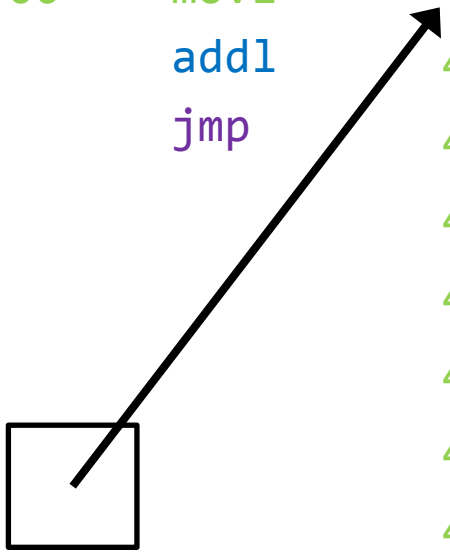
Progress through a function

```
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

Progress through a function

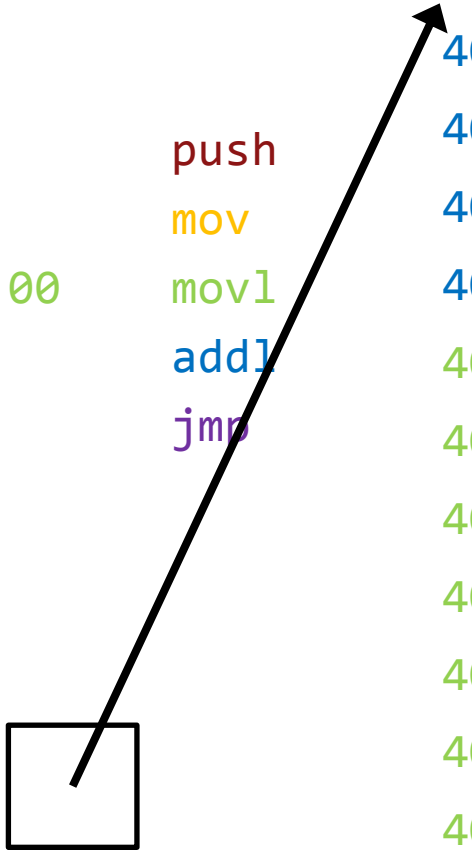
```
0000000004004ed <loop>:
  4004ed:       55                      push
  4004ee:       48 89 e5                mov
  4004f1:       c7 45 fc 00 00 00 00    movl
  4004f8:       83 45 fc 01             addl
  4004fc:       eb fa                   jmp
```



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

Progress through a function

```
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

This special register has a name:

`%rip`

Special hardware is responsible for setting its value to advance to the next instruction—very similar logic to the disassemble part of your assign4 that looks at opcode to determine how many subsequent bytes are there.

“Interfering” with `%rip`

If `%rip` always advances to next instruction, how do we “skip” instructions in an if-else, or 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>
```

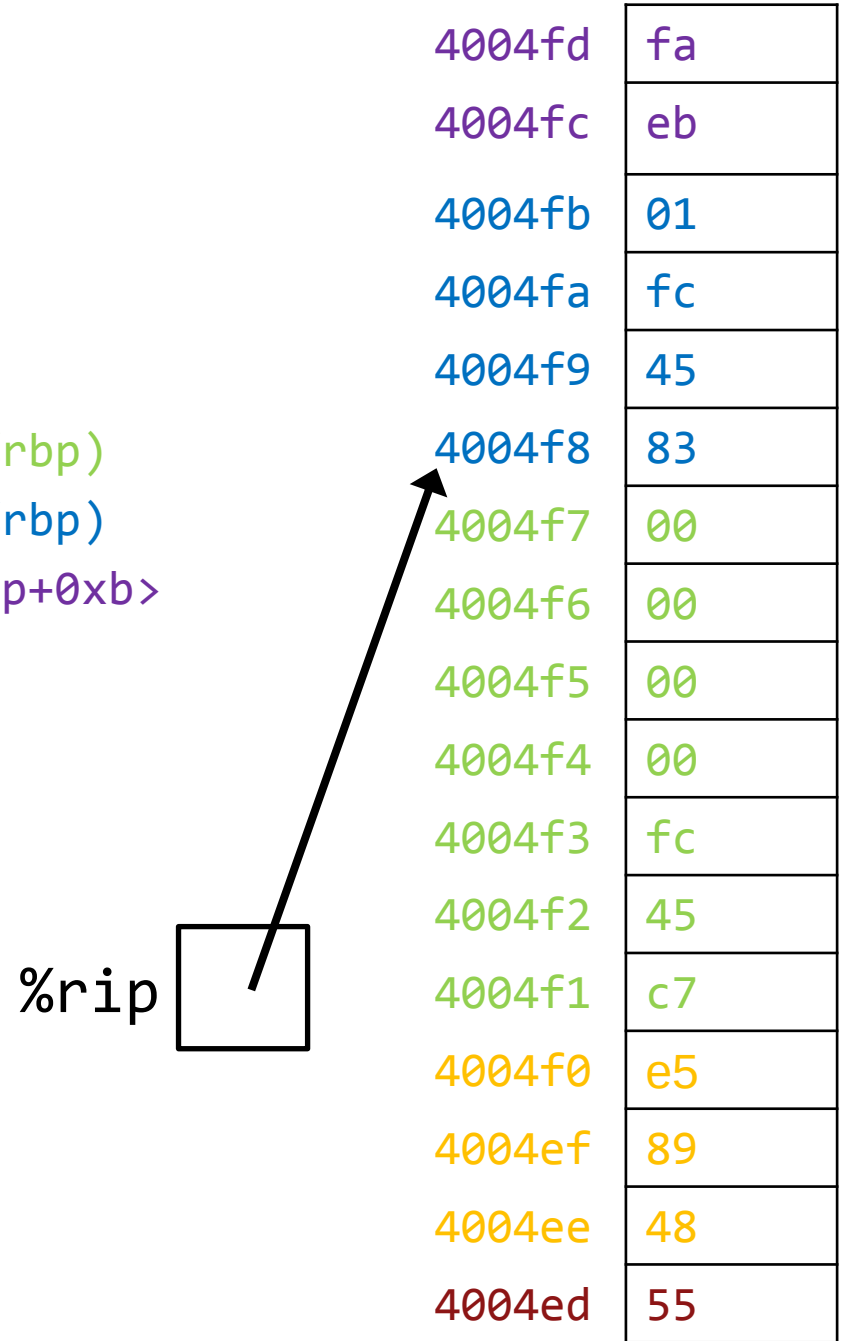
mov $4004f8, %rip

%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

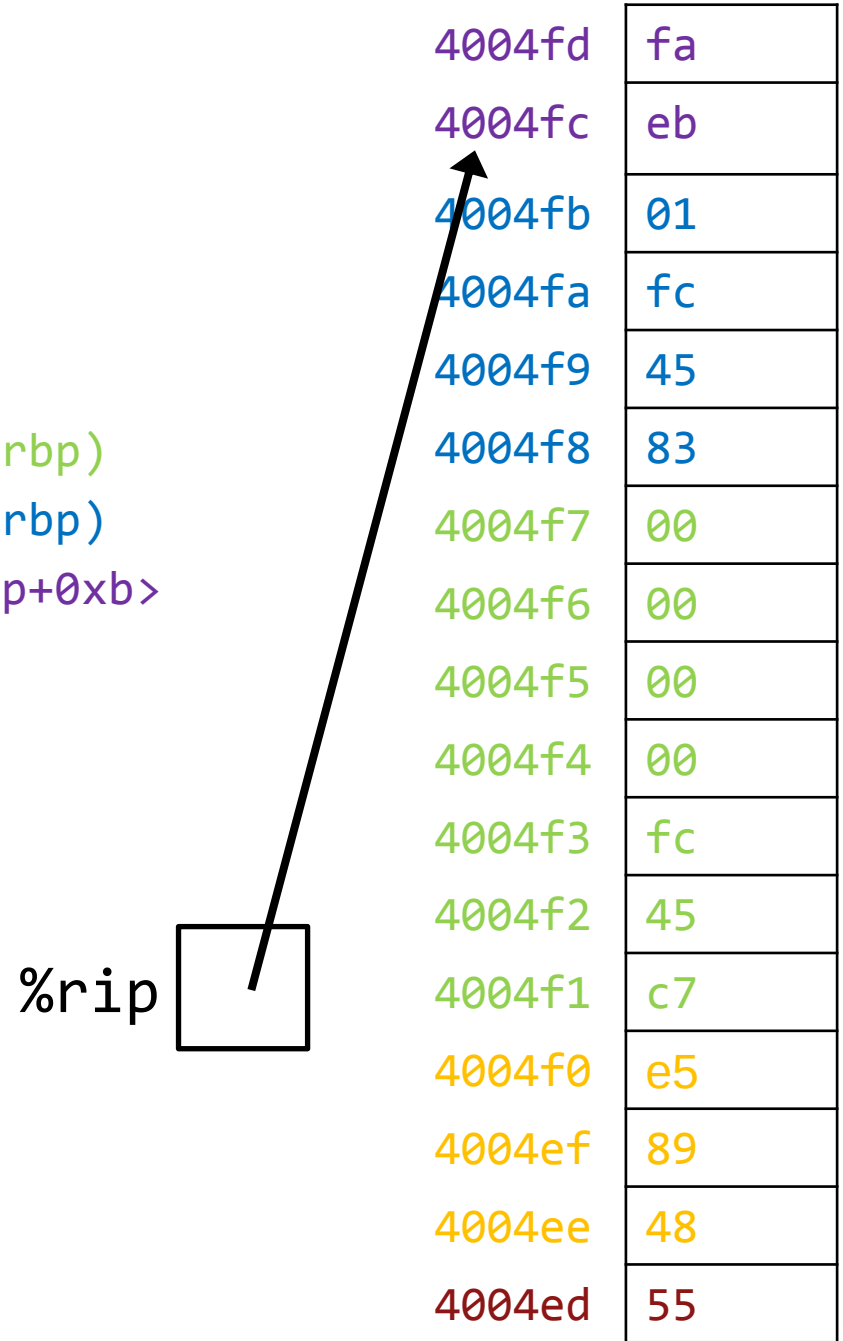
The jmp instruction

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



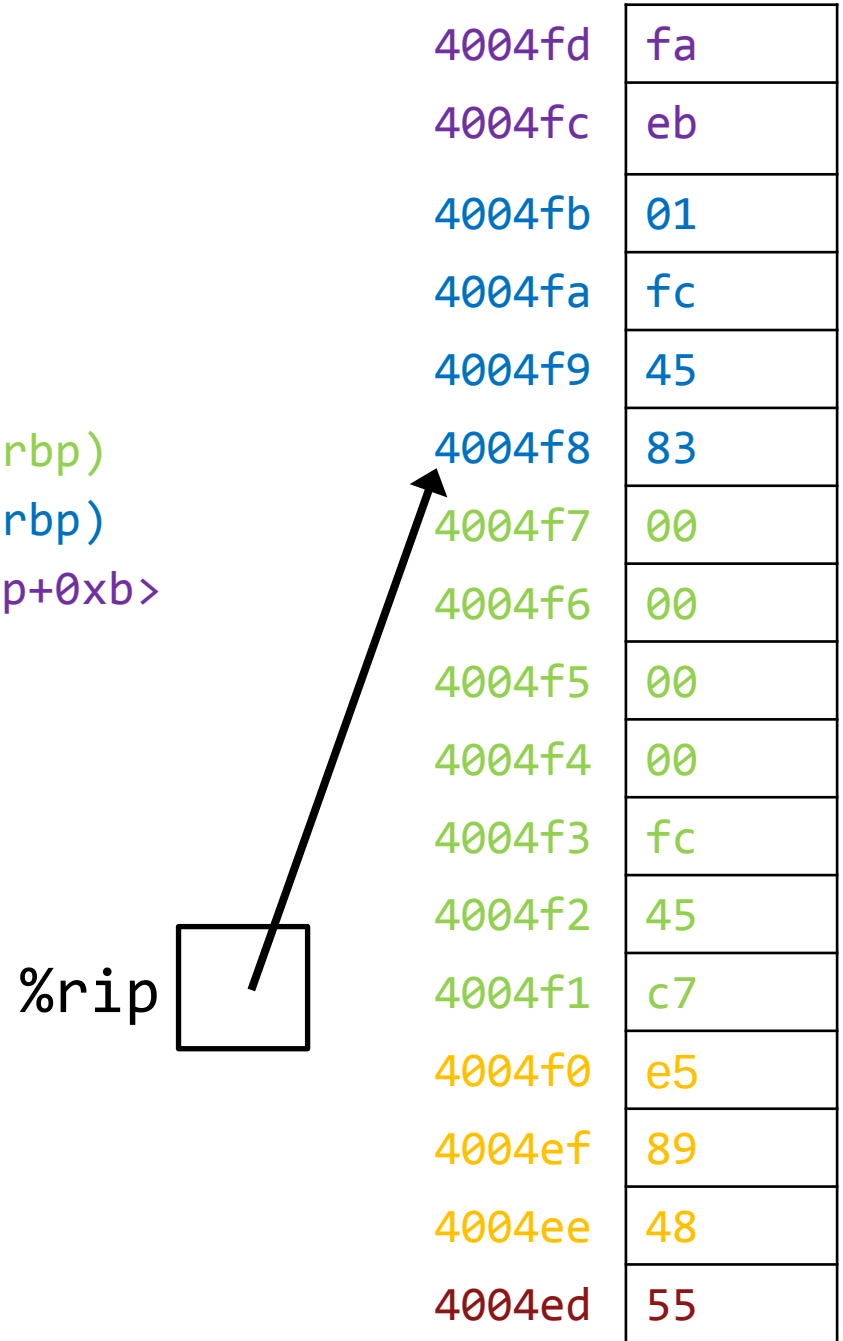
The jmp instruction

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



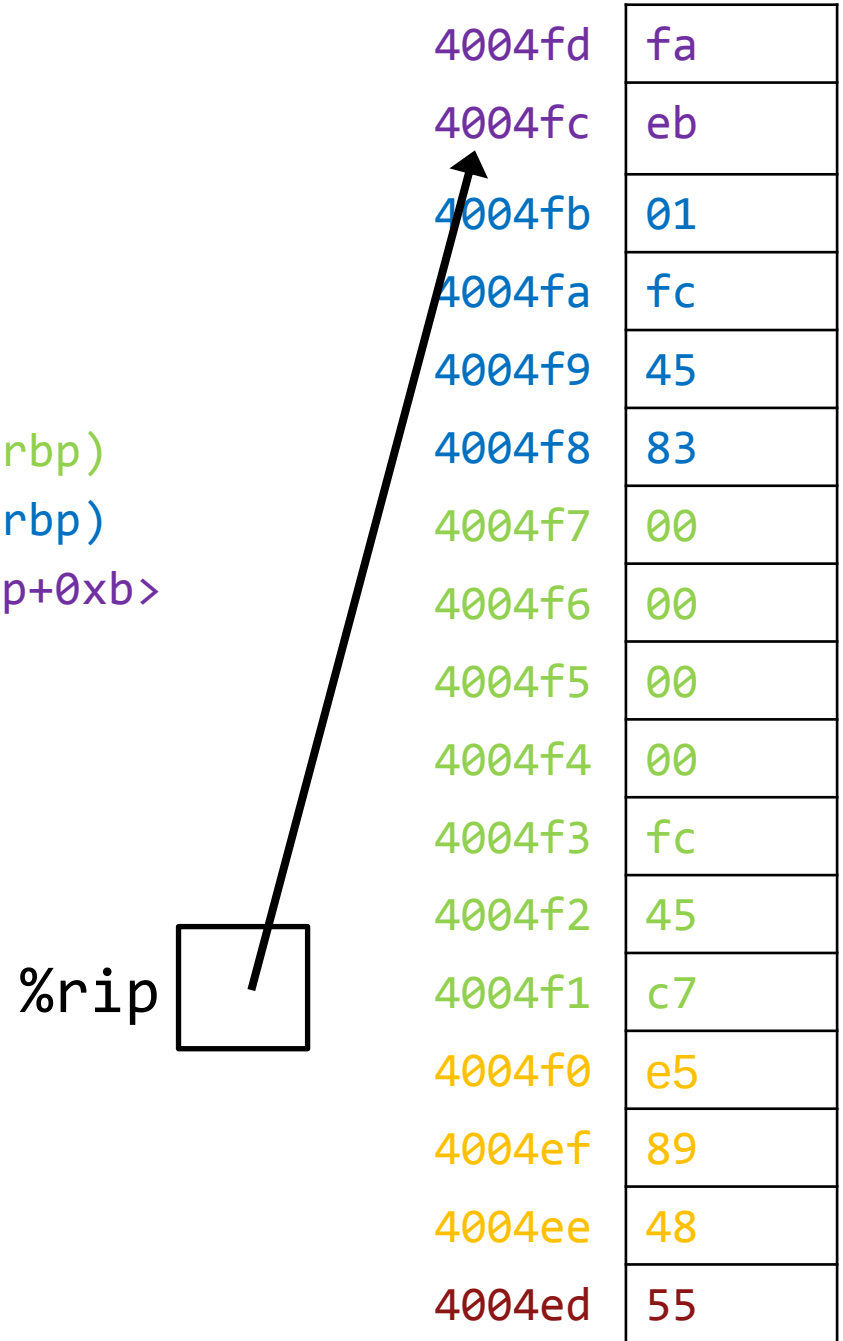
The jmp instruction

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



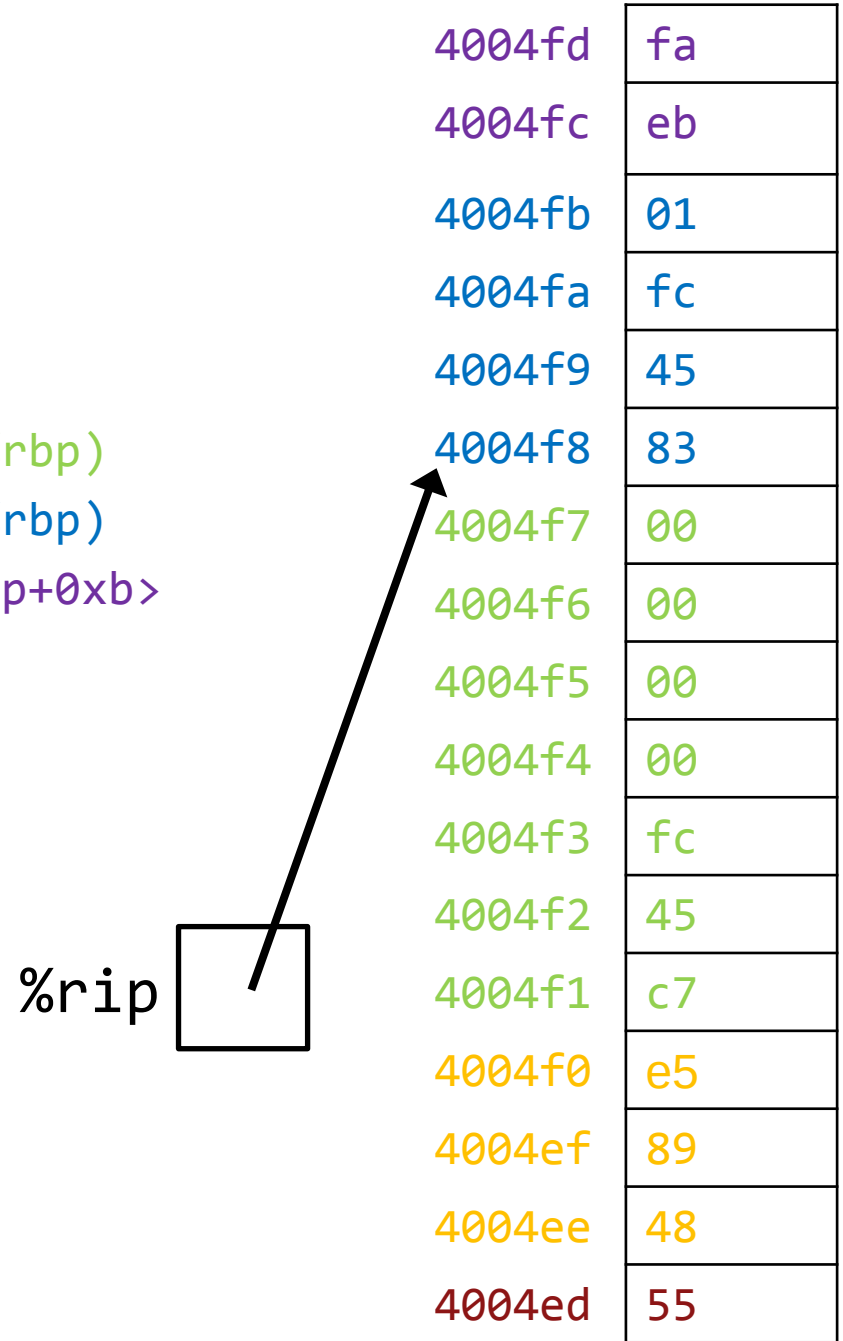
The jmp instruction

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



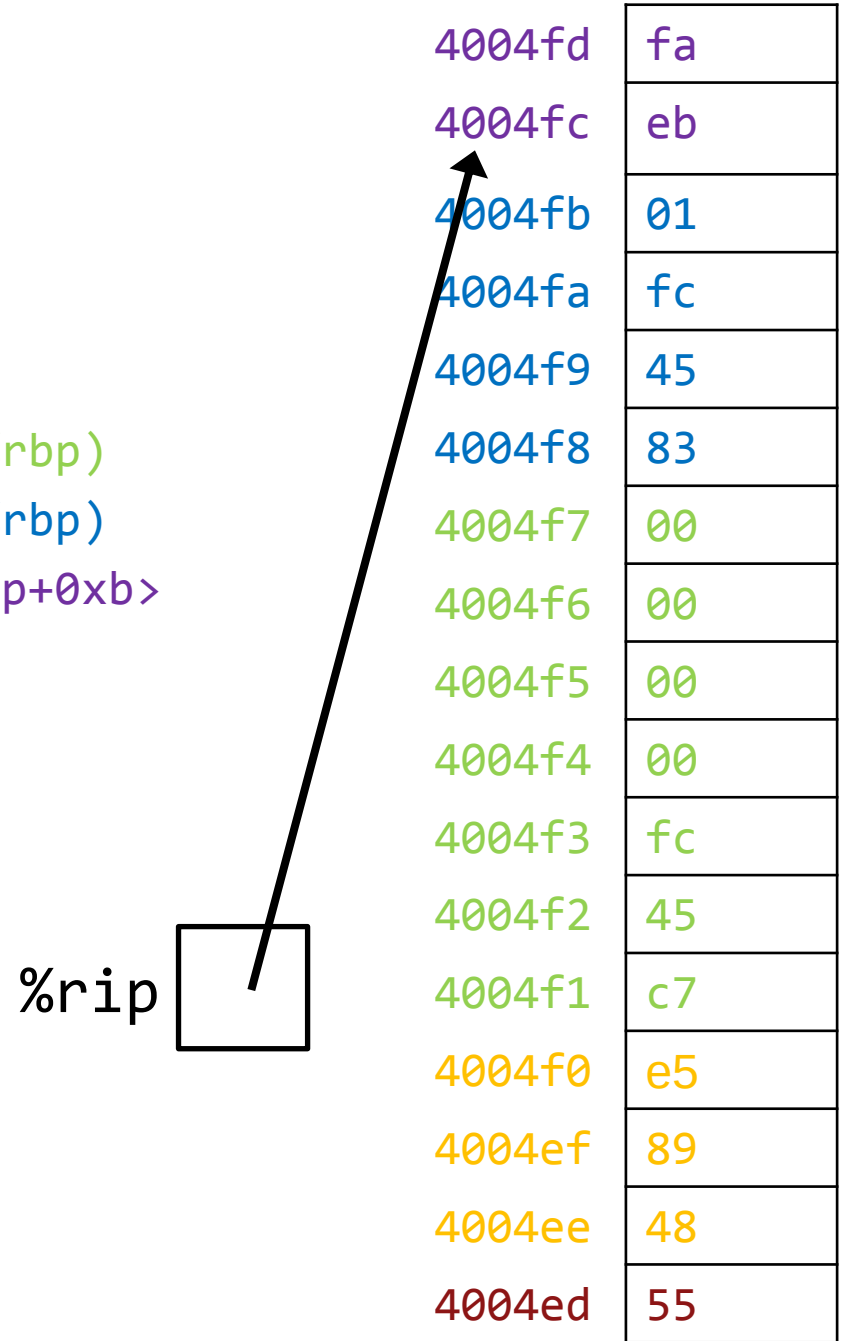
The jmp instruction

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



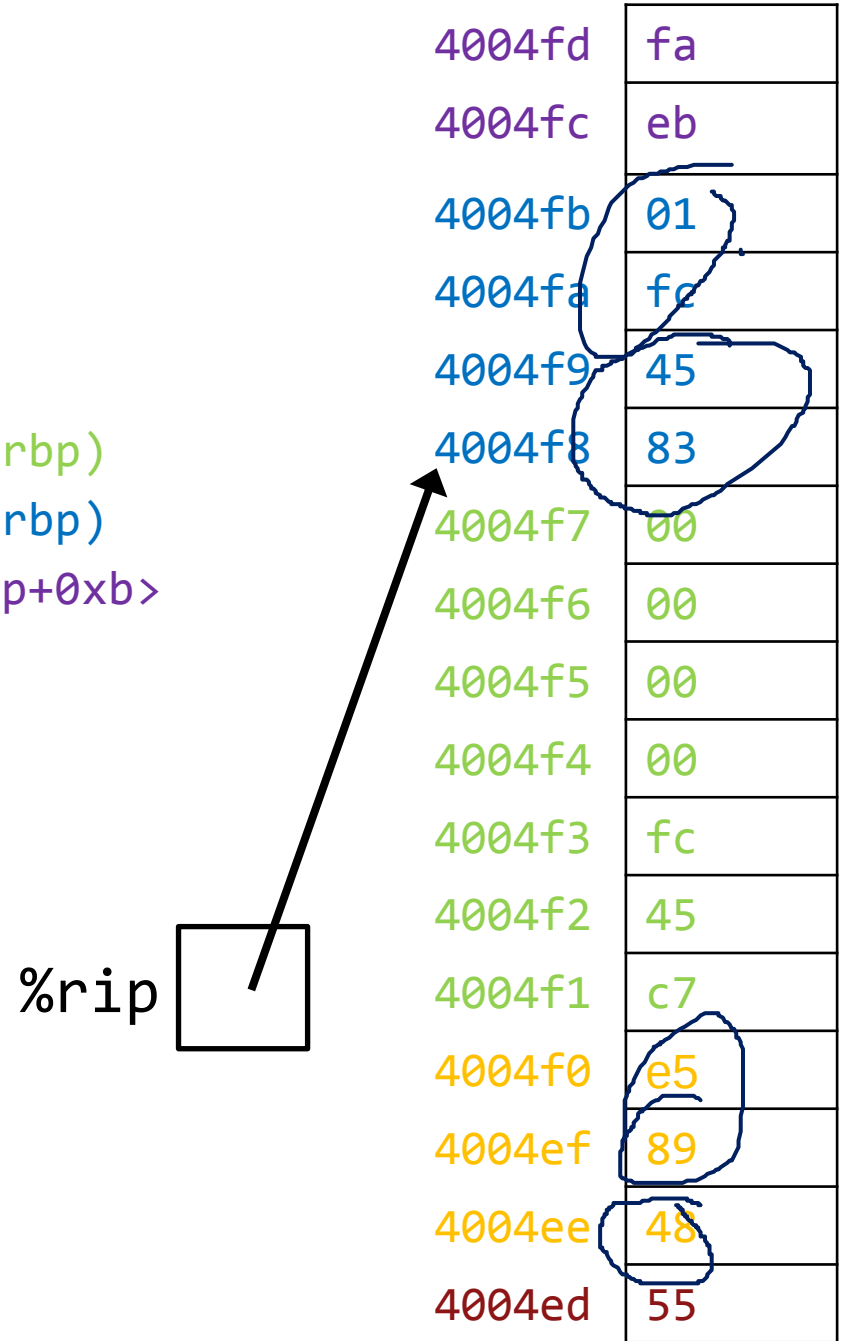
The jmp instruction

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



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>
```



C code for our example

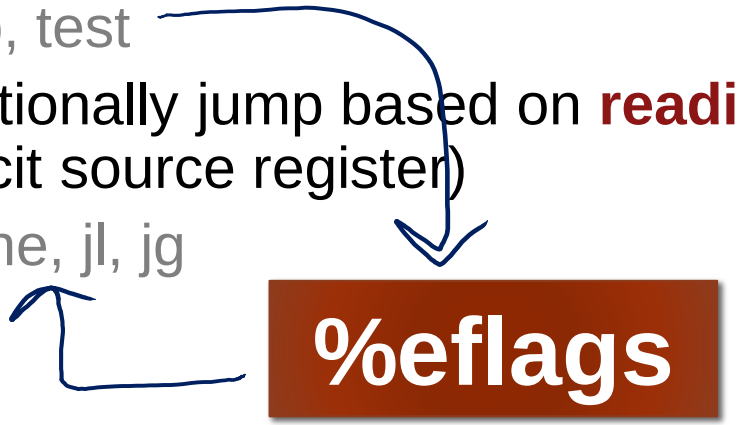
```
0000000004004ed <loop>:
  4004ed:       push   %rbp
  4004ee:       mov    %rsp,%rbp
  4004f1:       movl   $0x0,-0x4(%rbp)
  4004f8:       addl   $0x1,-0x4(%rbp)
  4004fc:       jmp    4004f8 <loop+0xb>
```

```
void loop()
{
    int i = 0;
again:
    ++i;
    goto again;
}
```

Conditional jumps

Typical 2-step control flow

1. Compare two values to **write** the condition codes (implicit destination register)
   - cmp, test
2. Conditionally jump based on **reading** the condition codes (implicit source register)
   - je, jne, jl, jg



- There is also a 1-step unconditional jump
- Doesn't look at condition code, just goes no matter what
  - jmp [target]

STEP 1 of control flow: cmp, test

Op	Source1	Source2	Dest Comments
`cmp`	`op2`	`op1`	op1 – op2, sets condition codes
`test`	`op2`	`op1`	op1 & op2, sets condition codes

- op1 and op2 can be any of the complex addressing modes we've seen
- Implicit destination %eflags contains condition codes
  - Sequence of Boolean values packed into one register
  - **t** is the result of the cmp or test operation above
    - 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"

What is the value of %eflags after this code?

```
0000000004004fe <if_then>:
  4004fe:       push   %rbp
  4004ff:       mov    %rsp,%rbp
  400502:       mov    %edi,-0x4(%rbp)
  400505:       cmpl   $0x6,-0x4(%rbp)
    …
```

Which of these bits (flags) are 1 (set) if we pass **argument 107** to this function?

- 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)

STEP 2 of control flow: jump

Op	Target	Remarks
`jmp`	`target`	`Unconditional jump`
`je`	`target`	`Jump if ZF is 1, in other words op1-op2=0 in previous cmp, in other words op1=op2`

- Target is a memory address: the address of the instruction where we should jump
- Implicit source %eflags contains condition codes
  - Sequence of Boolean values packed into one register
    - ZF = zero flag
    - SF = sign flag
    - CF = carry flag
    - OF = overflow flag
    - …

Control operations

```
cmpl op2, op1     # result = op1 - op2, discards result, sets condition
test op2, op1     # result = op1 & op2, discards result, sets condition

jmp target        # unconditional jump
je  target        # jump equal, synonym jz jump zero (ZF=1)
jne target        # jump not equal, synonym jnz (ZF=0)
jl  target        # jump less than, synonym jnge (SF!=OF)
jle target        # jump less or equal, synonym jng (ZF=1 or SF!=OF)
jg  target        # jump greater than, synonym jnle (ZF=0 and SF=OF)
jge target        # jump greater or equal, synonym jnl (SF=OF)
ja  target        # jump above, synonym jnbe (CF=0 and ZF=0)
jb  target        # jump below, synonym jnae (CF=1)
js  target        # jump signed (SF=1)
jns target        # jump not signed (SF=0)
```

What is the value of %rip after the jne?

```
0000000004004fe <if_then>:
  4004fe:       push   %rbp
  4004ff:       mov    %rsp,%rbp
  400502:       mov    %edi,-0x4(%rbp)
  400505:       cmpl   $0x6,-0x4(%rbp)
  400509:       jne    40050f
  40050b:       addl   $0x1,-0x4(%rbp)
…
```

What is param1 at the marked location when the input was 3?

a) 400509

b) 40050b

c) 40050f

d) Something else

Target instruction

Op	Target	Remarks
`jmp`	`target`	`Unconditional jump`
`je`	`target`	`Jump if ZF is 1, in other words op1-op2=0 in previous cmp, in other words op1=op2`

- Reminder: everything is bits/bytes to a computer!
  - Instructions are just 1s and 0s that we interpret in a certain way
  - Those bits/bytes are in a memory location, just like pointers, ints, floats, doge pictures, cat videos, and other data
  - So “target” is an address (or offset from current address) to write to the PC (program counter)

- char = 1 byte
- int = 4 bytes
- Double, void* = 8 bytes
- instruction = ?? bytes # **LETS LOOK IN GDB TO FIND OUT**

C translation examples

44

If statement construction

```
/* if-stmt */
void if_then(int param1)
{
    if (param1 == 6)
        param1++;
    param1 *= 2;
}
```

```
# gcc output
0000000004004fe <if_then>:
  4004fe:       push   %rbp
  4004ff:       mov    %rsp,%rbp
  400502:       mov    %edi,-0x4(%rbp)
  400505:       cmpl   $0x6,-0x4(%rbp)
  400509:       jne    40050f
  40050b:       addl   $0x1,-0x4(%rbp)
  40050f:       shll   -0x4(%rbp)
  400512:       pop    %rbp
  400513:       retq
```

45

If-else construction: Find the bug!

```
/* if-else */
void if_else(int param1)
{
    if (param1 < 5)
        param1++;
    else
        param1--;
    param1 = -param1;
    //what is param1 here?
}
```

```
# BUGGY (not actual gcc output, but close)
0000000000400514 <if_else>:
  400514:       push   %rbp
  400515:       mov    %rsp,%rbp
  400518:       mov    %edi,-0x4(%rbp)
  40051b:       cmpl   $0x4,-0x4(%rbp)
  40051f:       jg     400527
  400521:       addl   $0x1,-0x4(%rbp)
  400527:       subl   $0x1,-0x4(%rbp)
  40052b:       negl   -0x4(%rbp)
  40052e:       pop    %rbp
  40052f:       retq
```

What is param1 at the marked location when the input was 3?

(a) 2 (b) 3 (c) 4 (d) something else

46

If-else construction

```
/* if-else */
if (num > 3) {
    x = 10;
} else {
    x = 7;
}

/* equivalent if-else */
if (num <= 3) GOTO L2
x = 10;
GOTO L3
L2: x = 7;
L3: …
```

```
# equivalent AMD64 pseudocode
Test
Branch OVER if-body if test fails
If-body
jmp past else-body
Else-body
[PAST ELSE-BODY]
```

47

For loop construction

```
/* for loop */
for (int i=0; i<n; i++) {
    /* body */
}
```

```
# equivalent AMD64 pseudocode
Initialization
Test
Branch past loop if fails
Body
Increment
jmp to Test
```

```
/* equivalent while loop */
int i=0;
while (i<n) {
    /* body */
    i++;
}
```

48

For loop construction

```
# equivalent AMD64
Initialization
Test
Branch past loop if fails
Body
Increment
jmp to Test
```

```
# AMD64 gcc actually emits
Initialization
jmp to Test
Body
Increment
Test
Branch to Body if succeeds
```

- Same number of instructions! Why does gcc use the format on the right?

Say for loop "`for (int i=0; i<n; i++)`" and **`n=0`**, **`n=1000`**

Compare the instructions executed in the left and right

A. LEFT and RIGHT have same number of instructions
B. LEFT has more instructions (bad for left)
C. RIGHT has more instructions (bad for right)
D. Other/help

Computer Architecture BIG IDEA:
Code with Smaller Static Instruction Count != Code with Smaller Dynamic Instruction Count

- Our two codes had the same number of instructions
  - **Same** static instruction count
- If loop never executes, right had **higher** dynamic instruction count (bad for right)
- If loop executes many times, left had **higher** dynamic instruction count (bad for left)
- **This lack of correlation is very common!**
  - There are even cases where the compiler emits a static instruction count that is *several times* longer than an alternative, yet still more efficient assuming loops execute many times (e.g. loop unrolling)

Discussion question:

- Does the compiler **know** that the loop will execute many times?
  - In general, no!
- So…what if our code has loops that always execute a small number of times? Did gcc make a bad decision?

(take EE108, EE180, CS316 for more)