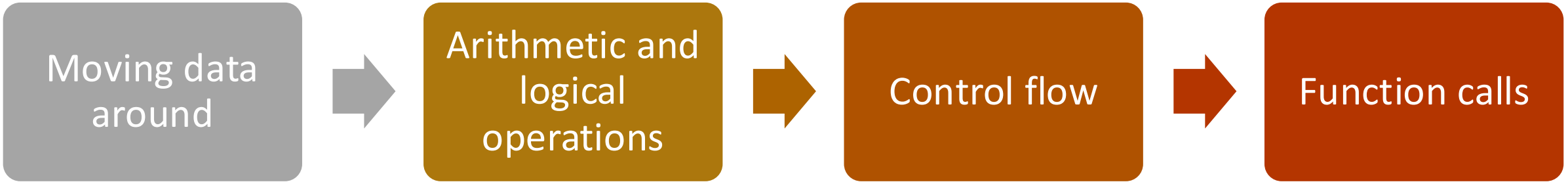
CS107, Lecture 19
Assembly: Control Flow, Continued

Reading: B&O 3.6

Learning Assembly



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
- Get practice using GDB to step through assembly and examine register contents.

Lecture Plan

- **Recap and continuing:** Control Flow Mechanics and **if** statements
- **test**, and more about Condition Codes
- Loops
  - While loops
  - For loops
- Other Instructions That Depend On Condition Codes

Lecture Plan

- **Recap and continuing: Control Flow Mechanics and if Statements**
- **test**, and more about Condition Codes
- Loops
  - While loops
  - For loops
- Other Instructions That Depend On Condition Codes

Control Flow Review

- %rip is a special register that stores the address of the next instruction to execute
- We can “interfere” with %rip to change the flow of our program’s execution
- **jmp** is one way to do this – it always jumps to the specified instruction location
- Conditional jumps are another way to do this – it jumps when its condition is true
- Conditional jumps work by relying on the *condition codes*; a set of 1-bit values that are updated by the result of the most recent arithmetic or logical operation.
- **cmp** is commonly paired with a conditional jump to check a condition

Control Flow Review

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

```
// Jump if %edi == 4:
// calculates %edi – 4,
// jump if result = 0 (ZF=1)
cmp $4, %edi
je [target]

// Jump if %edi != 3:
// calculates %edi – 3,
// jump if result != 0 (ZF=0)
cmp $3, %edi
jne [target]
```

```
// Jump if %edi > 2:
// calculates %edi – 2,
// jump if result != 0 (ZF=0) AND either
// (nonnegative and no signed overflow)
// OR (negative and signed overflow)
// (SF=OF)
cmp $2, %edi
jg [target]

// Jump if %edi <= 1:
// calculates %edi – 1,
// jump if result = 0 (ZF=1) OR (negative
// and no signed overflow) OR
// (nonnegative and signed overflow)
// (SF!=OF)
cmp $1, %edi
jle [target]
```

Practice: Fill In The Blank

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

  return param1 * 2;
}
```

```
0000000000401126 <if_then>:
  401126:   cmp    $0x5,%edi
  401129:   jg     40112e
  40112b:   add    $0x1,%edi
  40112e:   lea    (%rdi,%rdi,1),%eax
  401131:   ret
```

Alternative Construction

```
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:   ret
  40112f:   add    $0x1,%edi
  401132:   jmp    40112b
```

The structure may vary – to identify the **if** statement condition, identify the case where you do *extra operations.*

Common If-Else Construction

If-Else In C

```
int if_else(int param1) {
    if (param1 != 6) {
        param1++;
    } else {
        param1--;
    }

    return param1 * 2;
}
```

If-Else In Assembly pseudocode

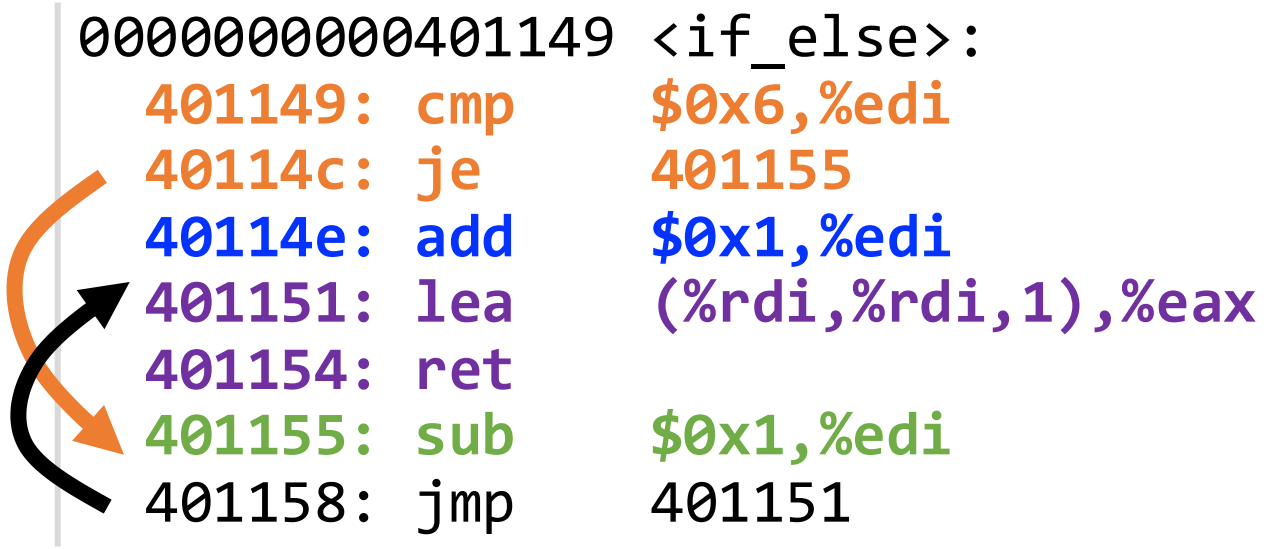
```
Check opposite of code condition
Jump to else-body if test passes
If-body
Past if body
Else-body
Jump up to past if body
```

Common If-Else Construction

If-Else In C

```
int if_else(int param1) {
    if (param1 != 6) {
        param1++;
    } else {
        param1--;
    }

    return param1 * 2;
}
```



If-Else In Assembly pseudocode

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

If-Else Construction Variations

C Code

```
int if_else_2(int param1) {
    if (param1 == 6) {
        param1++;
    } else {
        param1--;
    }

    return param1 * 2;
}
```

Assembly

```
0000000000401 15a <if_else_2>:
  40115a: cmp    $0x6,%edi
  40115d: je     401166
  40115f: sub    $0x1,%edi
  401162: lea    (%rdi,%rdi,1),%eax
  401165: ret
  401166: add    $0x1,%edi
  401169: jmp    401162
```

The structure may vary – key idea is to identify what operation(s) go with what check.

Lecture Plan

- **Recap and continuing:** Control Flow Mechanics and **if** statements
- ***test*, and more about Condition Codes**
- Loops
  - While loops
  - For loops
- Other Instructions That Depend On Condition Codes

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

The test Instruction

- TEST S1, S2 is S2 & S1

```
test %edi, %edi
jns  …
```



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

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!

Exercise 1: Conditional jump

```
je target              jump if ZF is 1
```

Let %edi store 0x20. Will we jump in the following cases?

%edi `0x20`

1.
```
cmp   $0x20,%edi
jge   40056f
add   $0x1,%edi
```

2.
```
test $0x20,%edi
je   40056f
add  $0x1,%edi
```

Respond on PollEv:
pollev.com/cs107

L19. Will we jump in the listed cases?

Exercise 1: Conditional jump

```
je target
```

jump if ZF is 1

Let %edi store 0x20. Will we jump in the following cases?

%edi | 0x20

1.
```
cmp  $0x20,%edi
jge  40056f
add  $0x1,%edi
```

%edi >= $0x20 (ZF = 1, one flag that jge checks), so jump.

2.
```
test $0x20,%edi
je   40056f
add  $0x1,%edi
```

S2 & S1 != 0, so don't jump

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

- **Recap and continuing:** Control Flow Mechanics and **if** statements
- **test**, and more about Condition Codes
- **Loops**
  - **While loops**
  - For loops
- Other Instructions That Depend On Condition Codes

GCC Possible While Loop Construction

C

```
while (test) {
      body
}
```

Assembly

```
Jump to check
Body
Check code condition
Jump to body if test passes
```

Loops and Control Flow

```
void loop() {
    int i = 0;
    while (i < 100) {
        i++;
    }
}
```

```
0x0000000000401106 <+0>:     mov    $0x0,%eax
0x000000000040110b <+5>:     jmp    0x401110 <loop+10>
0x000000000040110d <+7>:     add    $0x1,%eax
0x0000000000401110 <+10>:    cmp    $0x63,%eax
0x0000000000401113 <+13>:    jle    0x40110d <loop+7>
0x0000000000401115 <+15>:    ret
```

Jump to check
Body
Check code condition
Jump to body if test passes

Loops and Control Flow

```
void loop() {
    int i = 0;
    while (i < 100) {
        i++;
    }
}
```

```
0x0000000000401106 <+0>:    mov    $0x0,%eax
0x000000000040110b <+5>:    jmp    0x401110 <loop+10>
0x000000000040110d <+7>:    add    $0x1,%eax
0x0000000000401110 <+10>:   cmp    $0x63,%eax
0x0000000000401113 <+13>:   jle    0x40110d <loop+7>
0x0000000000401115 <+15>:   ret
```

Set %eax (i) to 0.

Jump to check
Body
Check code condition
Jump to body if test passes

Loops and Control Flow

```
void loop() {
    int i = 0;
    while (i < 100) {
        i++;
    }
}
```

```
0x0000000000401106 <+0>:    mov    $0x0,%eax
0x000000000040110b <+5>:    jmp    0x401110 <loop+10>
0x000000000040110d <+7>:    add    $0x1,%eax
0x0000000000401110 <+10>:   cmp    $0x63,%eax
0x0000000000401113 <+13>:   jle    0x40110d <loop+7>
0x0000000000401115 <+15>:   ret
```

Jump to check.

```
Jump to check
Body
Check code condition
Jump to body if test passes
```

Loops and Control Flow

```
void loop() {
    int i = 0;
    while (i < 100) {
        i++;
    }
}
```

```
0x0000000000401106 <+0>:    mov    $0x0,%eax
0x000000000040110b <+5>:    jmp    0x401110 <loop+10>
0x000000000040110d <+7>:    add    $0x1,%eax
0x0000000000401110 <+10>:   cmp    $0x63,%eax
0x0000000000401113 <+13>:   jle    0x40110d <loop+7>
0x0000000000401115 <+15>:   ret
```

Check code condition - jump if %eax is less than or equal to 0x63. It does this by calculating %eax – 0x63 and then **jle** checks the resulting condition codes.

```
Jump to check
Body
Check code condition
Jump to body if test passes
```

Loops and Control Flow

```
void loop() {
    int i = 0;
    while (i < 100) {
        i++;
    }
}
```

```
0x0000000000401106 <+0>:    mov    $0x0,%eax
0x000000000040110b <+5>:    jmp    0x401110 <loop+10>
0x000000000040110d <+7>:    add    $0x1,%eax
0x0000000000401110 <+10>:   cmp    $0x63,%eax
0x0000000000401113 <+13>:   jle    0x40110d <loop+7>
0x0000000000401115 <+15>:   ret
```

Add 1 to %eax (i).

```
Jump to check
Body
Check code condition
Jump to body if test passes
```

Loops and Control Flow

```
void loop() {
    int i = 0;
    while (i < 100) {
        i++;
    }
}
```

```
0x0000000000401106 <+0>:    mov    $0x0,%eax
0x000000000040110b <+5>:    jmp    0x401110 <loop+10>
0x000000000040110d <+7>:    add    $0x1,%eax
0x0000000000401110 <+10>:   cmp    $0x63,%eax
0x0000000000401113 <+13>:   jle    0x40110d <loop+7>
0x0000000000401115 <+15>:   ret
```

When this test is false, we do not jump back to the body; instead, we continue.

```
Jump to check
Body
Check code condition
Jump to body if test passes
```

Loops and Control Flow

```
void loop() {
    int i = 0;
    while (i < 100) {
        i++;
    }
}
```

```
0x0000000000401106 <+0>:     mov    $0x0,%eax
0x000000000040110b <+5>:     jmp    0x401110 <loop+10>
0x000000000040110d <+7>:     add    $0x1,%eax
0x0000000000401110 <+10>:    cmp    $0x63,%eax
0x0000000000401113 <+13>:    jle    0x40110d <loop+7>
0x0000000000401115 <+15>:    ret
```

Then, we return from the function.

```
Jump to check
Body
Check code condition
Jump to body if test passes
```

GCC Other While Loop Construction

C

```
while (test) {
      body
}
```

Assembly

```
Check opposite of code condition
Skip loop if test passes
Body
Jump back to test
```

```
0x000000000040115c <+0>:    mov    $0x0,%eax
0x0000000000401161 <+5>:    cmp    $0x63,%eax
0x0000000000401164 <+8>:    jg     0x40116b <loop+15>
0x0000000000401166 <+10>:   add    $0x1,%eax
0x0000000000401169 <+13>:   jmp    0x401161 <loop+5>
0x000000000040116b <+15>:   retq
```

The structure may vary – key idea is to identify if a code block can be repeated more than once (vs. at most once, meaning an **if** statement) and what the repeating condition is.

Lecture Plan

- **Recap and continuing:** Control Flow Mechanics and **if** statements
- **test**, and more about Condition Codes
- **Loops**
  - While loops
  - **For loops**
- Other Instructions That Depend On Condition Codes

For Loop Construction

C For loop

```
for (init; test; update) {
      body
}
```

C Equivalent While Loop

```
init
while(test) {
     body
     update
}
```

Assembly pseudocode

```
Init
Jump to test
Body
Update
Check code condition
Jump back to body if test passes
```

For loops and while loops are treated (essentially) the same when compiled down to assembly.

GCC For Loop Output

Possible GCC For Loop Output

```
Initialization
Test
Jump past loop if success
Body
Update
Jump to test
```

Other Possible GCC For Loop Output

```
Initialization
Jump to test
Body
Update
Test
Jump to body if success
```

The structure may vary – key idea is to identify if a code block can be repeated more than once (vs. at most once, meaning an **if** statement) and what the repeating condition is.

Back to Our First Assembly

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

```
0000000000401136 <sum_array>:
  401136 <+0>:     mov    $0x0,%eax
  40113b <+5>:     mov    $0x0,%edx
  401140 <+10>:    jmp    0x40114b <sum_array+21>
  401142 <+12>:    movslq %eax,%rcx
  401145 <+15>:    add    (%rdi,%rcx,4),%edx
  401148 <+18>:    add    $0x1,%eax
  40114b <+21>:    cmp    %esi,%eax
  40114d <+23>:    jl     0x401142 <sum_array+12>
  40114f <+25>:    mov    %edx,%eax
  401151 <+27>:    ret
```

1. Which register is C code's `sum`?
2. Which register is C code's `i`?
3. Which assembly instruction is C code's `sum += arr[i]`?
4. What are the **`cmp`** and **`jl`** instructions doing?
   (**`jl`**: signed jump less than)

Let's see a GDB demo!

Demo: GDB and Assembly

`sum_array.c`

gdb tips

`layout split`	`(ctrl-x a: exit, ctrl-l: resize, refresh, focus next)`	View C, assembly, and gdb (lab5)
`info reg`		Print all registers
`p $eax`		Print register value
`p $eflags`		Print all condition codes currently set
`b *0x400546`		Set breakpoint at assembly instruction
`b *0x400550 if $eax > 98`		Set **conditional breakpoint**
`ni`		Next assembly instruction
`si`		Step into assembly instruction (will step into function calls)

gdb tips

`p/x $rdi`	Print register value in hex
`p/t $rsi`	Print register value in binary
`x $rdi`	Examine the byte stored at this address
`x/4bx $rdi`	Examine 4 bytes starting at this address
`x/4wx $rdi`	Examine 4 ints starting at this address

Lecture Plan

- **Recap and continuing:** Control Flow Mechanics and **if** statements
- **test**, and more about Condition Codes
- Loops
  - While loops
  - For loops
- **Other Instructions That Depend On Condition Codes**

Condition Code-Dependent Instructions

There are three common instruction types that use condition codes:

- **jmp** instructions conditionally jump to a different next instruction
- **set** instructions conditionally set a byte to 0 or 1
- new versions of **mov** instructions conditionally move data

`set`: Read condition codes

`set` instructions conditionally set a byte to 0 or 1. (numeric value 1, not all 1s).

- Reads current state of flags
- Destination is a single-byte register (e.g., %al) or single-byte memory location
- Does not perturb other bytes of register
- Typically followed by movzbl to zero those bytes

```
int small(int x) {
    return x < 16;
}
```

```
# if %edi <= 0xf, set %al to 1.
# otherwise, set %al to 0.
cmp $0xf,%edi
setle %al
# zero-extend %al to fill %eax
movzbl %al, %eax
retq
```

`set`: Read condition codes

Instruction	Synonym	Set Condition (1 if true, 0 if false)
`sete D`	`setz`	Equal / zero
`setne D`	`setnz`	Not equal / not zero
`sets D`		Negative
`setns D`		Nonnegative
`setg D`	`setnle`	Greater (signed >)
`setge D`	`setnl`	Greater or equal (signed >=)
`setl D`	`setnge`	Less (signed <)
`setle D`	`setng`	Less or equal (signed <=)
`seta D`	`setnbe`	Above (unsigned >)
`setae D`	`setnb`	Above or equal (unsigned >=)
`setb D`	`setnae`	Below (unsigned <)
`setbe D`	`setna`	Below or equal (unsigned <=)

cmov: Conditional move

`cmovx src,dst` conditionally moves data in `src` to data in `dst`.

- Mov `src` to `dst` if condition `x` holds; no change otherwise
- `src` is memory address/register, `dst` is register
- May be more efficient than branch (i.e., jump)
- Often seen with C ternary operator: `result = test ? then: else;`

```
int max(int x, int y) {
    return x > y ? x : y;
}
```

```
cmp     %edi,%esi
mov     %edi, %eax
# If %esi >= %edi, mov.
# otherwise, do nothing.
cmovge %esi, %eax
retq
```

cmov: Conditional move

Instruction	Synonym	Move Condition
cmove S,R	cmovz	Equal / zero (ZF = 1)
cmovne S,R	cmovnz	Not equal / not zero (ZF = 0)
cmovs S,R		Negative (SF = 1)
cmovns S,R		Nonnegative (SF = 0)
cmovg S,R	cmovnle	Greater (signed >) (SF = 0 and SF = OF)
cmovge S,R	cmovnl	Greater or equal (signed >=) (SF = OF)
cmovl S,R	cmovnge	Less (signed <) (SF != OF)
cmovle S,R	cmovng	Less or equal (signed <=) (ZF = 1 or SF! = OF)
cmova S,R	cmovnbe	Above (unsigned >) (CF = 0 and ZF = 0)
cmovae S,R	cmovnb	Above or equal (unsigned >=) (CF = 0)
cmovb S,R	cmovnae	Below (unsigned <) (CF = 1)
cmovbe S,R	cmovna	Below or equal (unsigned <=) (CF = 1 or ZF = 1)

Last Lab: Conditional Move

```
int signed_division(int x) {
    return x / 4;
}
```

```
signed_division:
  leal 3(%rdi), %eax
  testl %edi, %edi
  cmovns %edi, %eax
  sarl $2, %eax
  ret
```

Put x + 3 into %eax

Check the sign of x

If x is nonnegative, put x into %eax

Divide %eax by 4

Recap

- **Recap and continuing:** Control Flow Mechanics and **if** statements
- **test**, and more about Condition Codes
- Loops
  - While loops
  - For loops
- Other Instructions That Depend On Condition Codes

Lecture 19 takeaway: Loops and conditionals commonly use cmp or test along with jumps to conditionally skip over or repeat assembly instructions. We can use GDB to step through individual assembly instructions and view register contents.

Next time: functions and the stack

Practice: Fill in the blanks

```
long loop(long a, long b) {
    long result = ___(1)___;
    while (___(2)___) {
      result = ____(3)_____;
      a = ___(4)___;
    }
    return result;
}
```

```
<+0>:   mov    $0x1,%eax
<+5>:   cmp    %rsi,%rdi
<+8>:   jge    0x1151 <loop+24>
<+10>:  lea    (%rdi,%rsi,1),%rdx
<+14>:  imul   %rdx,%rax
<+18>:  add    $0x1,%rdi
<+22>:  jmp    0x113e <loop+5>
<+24>:  retq
```

GCC common while loop construction:

Test

Jump past loop if fails

Body

Jump to test

https://godbolt.org/z/zrW6c5MGa

Practice: Fill in the blanks

```
long loop(long a, long b) {
    long result = ___(1)___;
    while (___(2)___) {
      result = ____(3)_____;
      a = ___(4)___;
    }
    return result;
}
```

```
<+0>:   mov    $0x1,%eax
<+5>:   cmp    %rsi,%rdi
<+8>:   jge    0x1151 <loop+24>
<+10>:  lea    (%rdi,%rsi,1),%rdx
<+14>:  imul   %rdx,%rax
<+18>:  add    $0x1,%rdi
<+22>:  jmp    0x113e <loop+5>
<+24>:  retq
```

GCC common while loop construction:

Test
Jump past loop if fails
Body
Jump to test

Practice: Fill in the blanks

```
long loop(long a, long b) {
    long result = _________;
    while (_________) {
      result = ___________;
      a = _________;
    }
    return result;
}
```

```
<+0>:   mov    $0x1,%eax

<+5>:   cmp    %rsi,%rdi
<+8>:   jge    0x1151 <loop+24>

<+10>:  lea    (%rdi,%rsi,1),%rdx
<+14>:  imul   %rdx,%rax
<+18>:  add    $0x1,%rdi

<+22>:  jmp    0x113e <loop+5>

<+24>:  retq
```

Practice: Fill in the blanks

```
long loop(long a, long b) {
    long result = ____1____;
    while (__a < b__) {
      result = result*(a+b);
      a = __a + 1__;
    }
    return result;
}
```

```
<+0>:   mov    $0x1,%eax

<+5>:   cmp    %rsi,%rdi
<+8>:   jge    0x1151 <loop+24>

<+10>:  lea    (%rdi,%rsi,1),%rdx
<+14>:  imul   %rdx,%rax
<+18>:  add    $0x1,%rdi

<+22>:  jmp    0x113e <loop+5>

<+24>:  retq
```

`test` practice: What's the C code?

```
0x400546 <test_func>    test   %edi,%edi
0x400548 <test_func+2>  jns    0x400550 <test_func+10>
0x40054a <test_func+4>  mov    $0xfeed,%eax
0x40054f <test_func+9>  retq
0x400550 <test_func+10> mov    $0xaabbccdd,%eax
0x400555 <test_func+15> retq
```

`test` practice: What's the C code?

```
0x400546 <test_func>     test   %edi,%edi
0x400548 <test_func+2>   jns    0x400550 <test_func+10>
0x40054a <test_func+4>   mov    $0xfeed,%eax
0x40054f <test_func+9>   retq
0x400550 <test_func+10>  mov    $0xaabbccdd,%eax
0x400555 <test_func+15>  retq
```

```
int test_func(int x) {
    if (x < 0) {
        return 0xfeed;
    }
    return 0xaabbccdd;
}
```

(or anything like this)

Practice: "Escape Room"

```
<escape_room+0>     lea    (%rdi,%rdi,1),%eax
<escape_room+3>     cmp    $0x5,%eax
<escape_room+6>     jg     0x114c <escape_room+19>
<escape_room+8>     cmp    $0x1,%edi
<escape_room+11>    je     0x1152 <escape_room+25>
<escape_room+13>    mov    $0x0,%eax
<escape_room+18>    retq
<escape_room+19>    mov    $0x1,%eax
<escape_room+24>    retq
<escape_room+25>    mov    $0x1,%eax
<escape_room+30>    retq
```

What must be passed to the escapeRoom function such that it returns true (1) and not false (0)?

You don't have to reverse-engineer C code exactly!

Just figure out the big picture!

Practice: “Escape Room”

```
<escape_room+0>     lea    (%rdi,%rdi,1),%eax
<escape_room+3>     cmp    $0x5,%eax
<escape_room+6>     jg     0x114c <escape_room+19>
<escape_room+8>     cmp    $0x1,%edi
<escape_room+11>    je     0x1152 <escape_room+25>
<escape_room+13>    mov    $0x0,%eax
<escape_room+18>    retq
<escape_room+19>    mov    $0x1,%eax
<escape_room+24>    retq
<escape_room+25>    mov    $0x1,%eax
<escape_room+30>    retq
```

What must be passed to the escapeRoom function such that it returns true (1) and not false (0)?

First param > 2 or == 1.

Exercise 2: Conditional jump

```
0000000004004d6 <if_then>:
  4004d6:   83 ff 06    cmp    $0x6,%edi
  4004d9:   75 03       jne    4004de <if_then+0x8>
  400rdb:   83 c7 01    add    $0x1,%edi
  4004de:   8d 04 3f    lea    (%rdi,%rdi,1),%eax
  4004e1:   c3          retq
```

%edi `0x5`

1. What is the value of `%rip` after executing the `jne` instruction?
   A. `4004d9`
   B. `4004db`
   C. `4004de`
   D. Other

2. What is the value of `%eax` when we hit the `retq` instruction?
   A. `4004e1`
   B. `0x2`
   C. `0xa`
   D. `0xc`
   E. Other

Exercise 2: Conditional jump

```
0000000004004d6 <if_then>:
  4004d6:   83 ff 06    cmp    $0x6,%edi
  4004d9:   75 03       jne    4004de <if_then+0x8>
  400rdb:   83 c7 01    add    $0x1,%edi
  4004de:   8d 04 3f    lea    (%rdi,%rdi,1),%eax
  4004e1:   c3          retq
```

%edi **0x5**

1. What is the value of `%rip` after executing the `jne` instruction?
   - A. `4004d9`
   - B. `4004db`
   - C. `4004de`
   - D. Other

2. What is the value of `%eax` when we hit the `retq` instruction?
   - A. `4004e1`
   - B. `0x2`
   - C. `0xa`
   - D. `0xc`
   - E. Other