CSED211: Lab. 3
BombLab

조성준
allencho1222@postech.ac.kr

POSTECH

2023.09.25

What We Have Learned

- Linux commands
- Bit and Byte
- Bitwise operation (~, &, |, ^, <<, >>)
- Date representation

Table of Contents

- GCC
- GDB
- Assembly Language
- Homework

GCC

- GNU compiler collection
  - C, C++, Go, …
  - Standard libraries (e.g., libstdc++)
- How to compile using GCC?
  - Create an executable file from .c (C source file):

```
$ gcc -o execfile source_code.c
```

  - Create an executable file from .s (assembly file)

```
$ gcc -o execfile source_code.s
```

  - How to create an assembly file from .c?

```
$ gcc -S source_code.c -o asm.s
```

GDB

- GNU project debugger
  - Allows you to see what is going on inside a target program while it is being executed

- How to execute the target program over GDB?
  - Verify that GDB (`gdb`) is installed on your machine

    ```
    $ which gdb # output an installed location of GDB
    ```

  - Run the target program (`execfile`) over GDB

    ```
    $ gdb execfile
    ```

  - Finally, you will be prompted

    ```
    (gdb)
    ```

GDB Commands

- Disassemble (`disas`) **a specific function** by name

```
(gdb)disas main
```

- List (`l`) source code lines for a specific function by name

```
(gdb)l main
```

  - Need the `-g` flag when compiling (e.g., `gcc -g -o test test.c`)

GDB Commands

- Set a breakpoint (`b`) at a specific memory address

```
(gdb)b *main
```
(Note: the function name is memory address)

```
(gdb)b *main+11
(gdb)b *0x300
```

- Print (`i`) breakpoints (`b`)

```
(gdb)i b
```

- Clear (`cl`) a breakpoint

```
(gdb)cl *main
```

- Delete (`d`) a breakpoint by `Num`

```
(gdb)d 2
(gdb)d
```
(Delete all breakpoints)

GDB Commands

- Kill (`k`) the current running program

```
(gdb) k
```

- Run (`r`) the program

```
(gdb) r arg1, arg2, …
```

  - Kill the current running program and re-run the program (e.g., restart)
  - If you've set any breakpoints, GDB will stop at the first breakpoint

- Continue (`c`) the program

```
(gdb) c
```

  - Resume program execution from the current point to the next breakpoint or endpoint

GDB Commands

- Step (`s`) instruction (`i`): execute one machine instruction

```
(gdb) si
```

- Next (`n`) instruction (`i`): execute one machine instruction

```
(gdb) ni
```

  - If it is a function call, proceed until the function returns

GDB Commands

- Examine (`x`) contents of memory

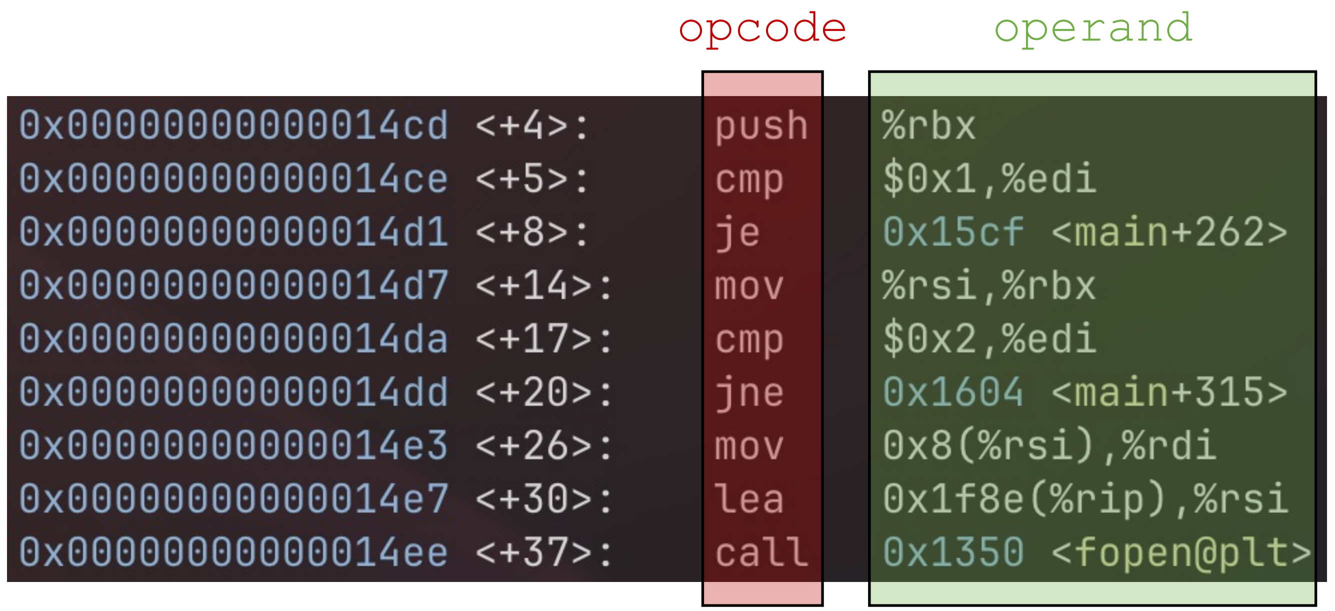
  (gdb) `x/wx addr`

  (gdb) `x/s addr`

  - `wx`: word, hex
  - `s`: string

Assembly Language

- Assembly language is what a machine really sees and runs
  - But, the machine actually executes binary code (usually called machine code)
- Syntax
  - opcode (what an instruction does) + operand (data)



```
0x00000000000014cd <+4>:    push   %rbx
0x00000000000014ce <+5>:    cmp    $0x1,%edi
0x00000000000014d1 <+8>:    je     0x15cf <main+262>
0x00000000000014d7 <+14>:   mov    %rsi,%rbx
0x00000000000014da <+17>:   cmp    $0x2,%edi
0x00000000000014dd <+20>:   jne    0x1604 <main+315>
0x00000000000014e3 <+26>:   mov    0x8(%rsi),%rdi
0x00000000000014e7 <+30>:   lea    0x1f8e(%rip),%rsi
0x00000000000014ee <+37>:   call   0x1350 <fopen@plt>
```

Assembly Language (OPCODE)

- `Opcode` specifies the operation performed by a machine
  - Unary operator (Format: `opcode operand`)
    - `pop, inc, dec, jmp, …`
  - Binary operator (Format: `opcode operand1, operand2`)
    - `mov, add, sub, cmp, …`

Assembly Language (OPERAND)

- `Operand` specific the data used by the operation (`opcode`)
  - Immediate: real values
    - `$2, $6, $-1`
  - Register: register values
    - `%rax, %rsp, %rsi`
  - Memory: values at specific memory location
    - `%-8(%ebx), 12(%ebx), …`

Assembly Language (Registers)

- x86_64 instruction set architecture (ISA) includes 16 general purpose registers
  - `rbp` and `rsp` are used to manage stack frame
  - `rsi` and `rdi` are used for source and destination index, repsectively

    -
      ```
      mov al, [rsi]
      mov [rdi], al
      ```

rax	r8
rbx	r9
rcx	r10
rdx	r11
rsi	r12
rdi	r13
rbp	r14
rsp	r15

Assembly Language (Registers)

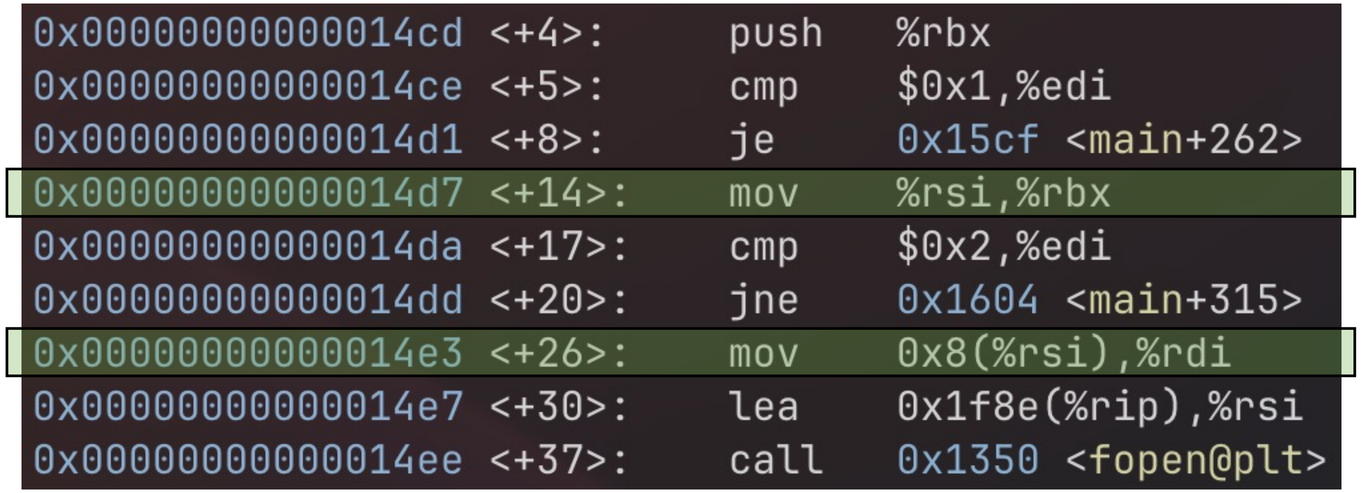
- x86_64 uses different registers to access bits

64-bit	RAX							
32-bit					EAX			
16-bit							AX	
8-bit							AH	AL
Value (hex)	01	23	45	67	89	AB	CD	EF

- You can google them to find more details

Assembly Language (Instructions)

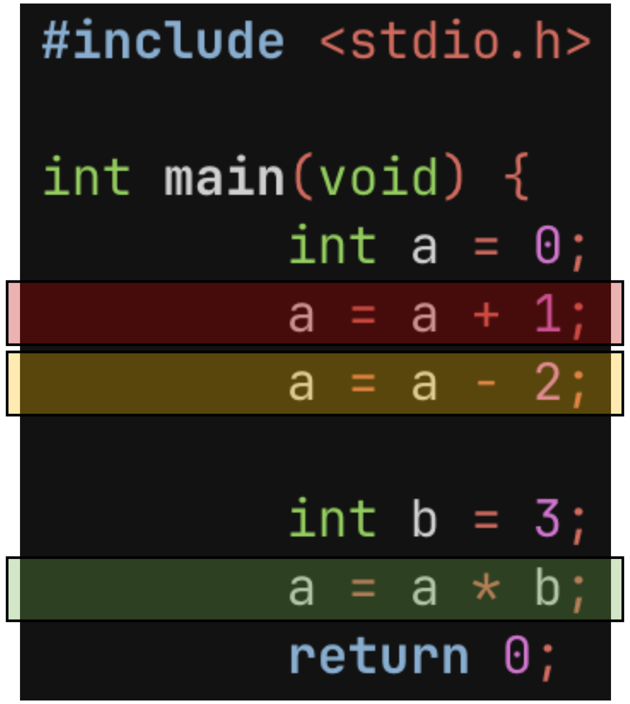
- Date movement instruction
  - `mov src_operand, dest_operand`



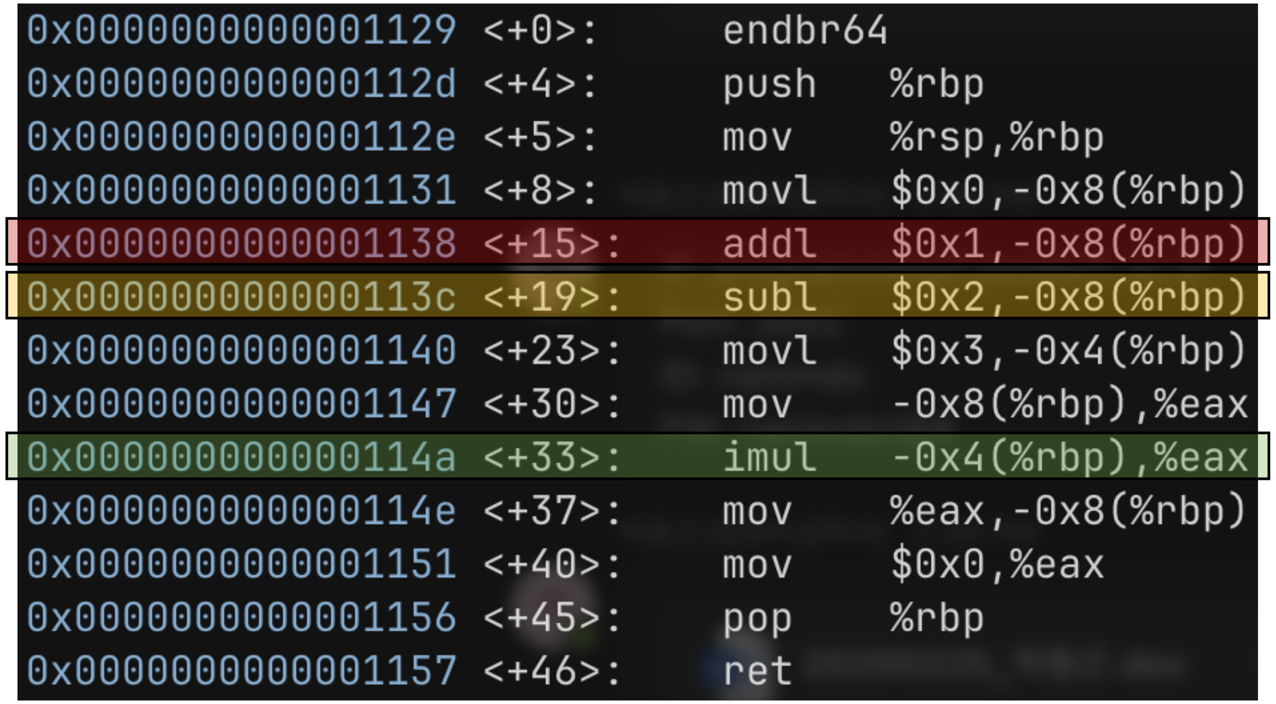
```
0x00000000000014cd <+4>:     push   %rbx
0x00000000000014ce <+5>:     cmp    $0x1,%edi
0x00000000000014d1 <+8>:     je     0x15cf <main+262>
0x00000000000014d7 <+14>:    mov    %rsi,%rbx
0x00000000000014da <+17>:    cmp    $0x2,%edi
0x00000000000014dd <+20>:    jne    0x1604 <main+315>
0x00000000000014e3 <+26>:    mov    0x8(%rsi),%rdi
0x00000000000014e7 <+30>:    lea    0x1f8e(%rip),%rsi
0x00000000000014ee <+37>:    call   0x1350 <fopen@plt>
```

Assembly Language (Instructions)

- Arithmetic instructions
  - `add operand1, operand2`
  - `sub operand1, operand2`
  - `mul operand1, operand2`



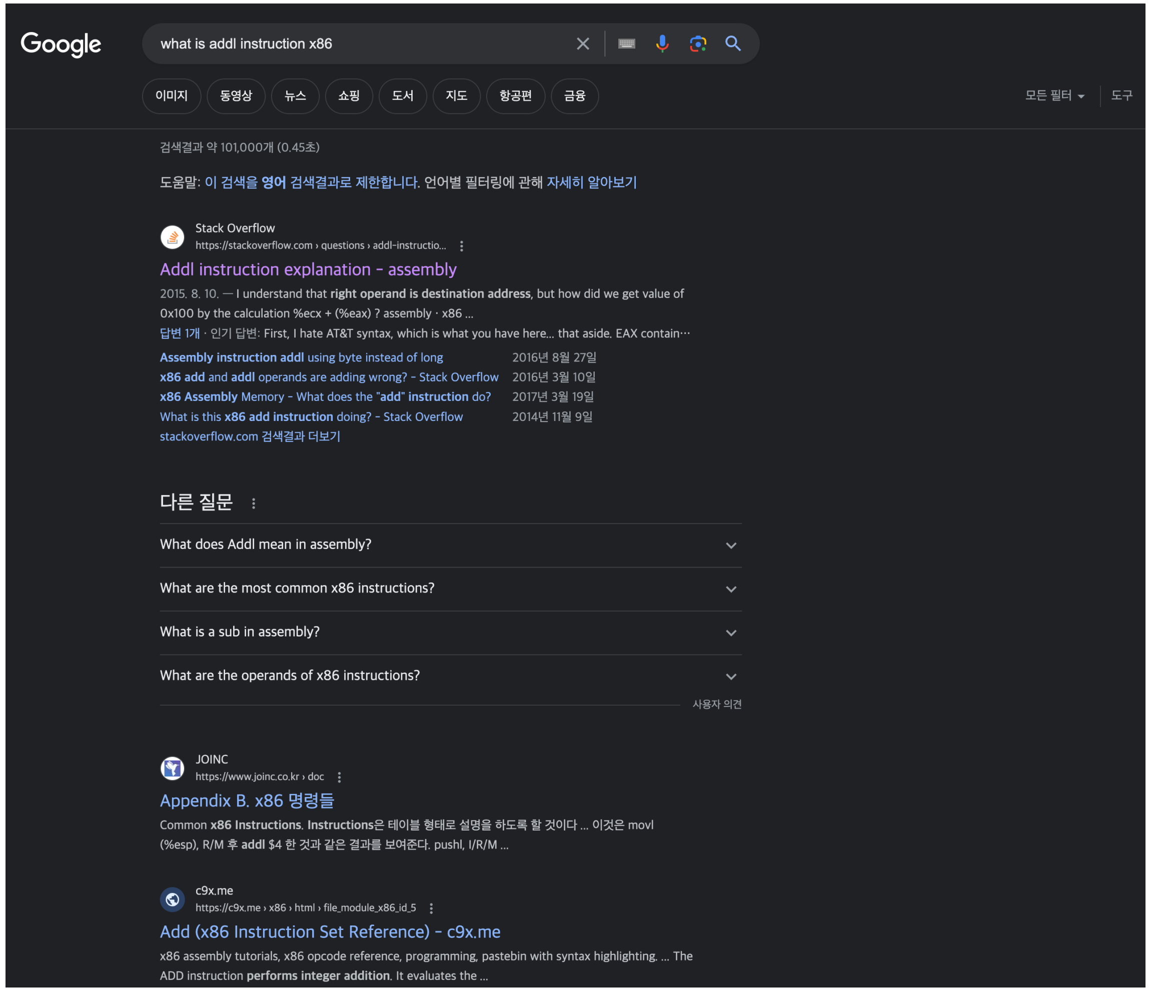
```c
#include <stdio.h>

int main(void) {
        int a = 0;
        a = a + 1;
        a = a - 2;

        int b = 3;
        a = a * b;
        return 0;
```



```
0x0000000000001129 <+0>:     endbr64
0x000000000000112d <+4>:     push   %rbp
0x000000000000112e <+5>:     mov    %rsp,%rbp
0x0000000000001131 <+8>:     movl   $0x0,-0x8(%rbp)
0x0000000000001138 <+15>:    addl   $0x1,-0x8(%rbp)
0x000000000000113c <+19>:    subl   $0x2,-0x8(%rbp)
0x0000000000001140 <+23>:    movl   $0x3,-0x4(%rbp)
0x0000000000001147 <+30>:    mov    -0x8(%rbp),%eax
0x000000000000114a <+33>:    imul   -0x4(%rbp),%eax
0x000000000000114e <+37>:    mov    %eax,-0x8(%rbp)
0x0000000000001151 <+40>:    mov    $0x0,%eax
0x0000000000001156 <+45>:    pop    %rbp
0x0000000000001157 <+46>:    ret
```

Assembly Language (Instructions)

- Not clear? Try to google keywords or use chatgpt



검색결과 약 101,000개 (0.45초)

도움말: 이 검색을 **영어** 검색결과로 제한합니다. 언어별 필터링에 관해 자세히 알아보기

Stack Overflow
https://stackoverflow.com › questions › addl-instructio...

Addl instruction explanation - assembly

2015. 8. 10. — I understand that **right operand is destination address**, but how did we get value of 0x100 by the calculation %ecx + (%eax) ? assembly · x86 ...

답변 1개 · 인기 답변: First, I hate AT&T syntax, which is what you have here... that aside. EAX contain···

Assembly instruction addl using byte instead of long	2016년 8월 27일
x86 add and **addl** operands are adding wrong? - Stack Overflow	2016년 3월 10일
x86 Assembly Memory - What does the **"add" instruction** do?	2017년 3월 19일
What is this **x86 add instruction** doing? - Stack Overflow	2014년 11월 9일

stackoverflow.com 검색결과 더보기

다른 질문

What does Addl mean in assembly?

What are the most common x86 instructions?

What is a sub in assembly?

What are the operands of x86 instructions?

사용자 의견

JOINC
https://www.joinc.co.kr › doc

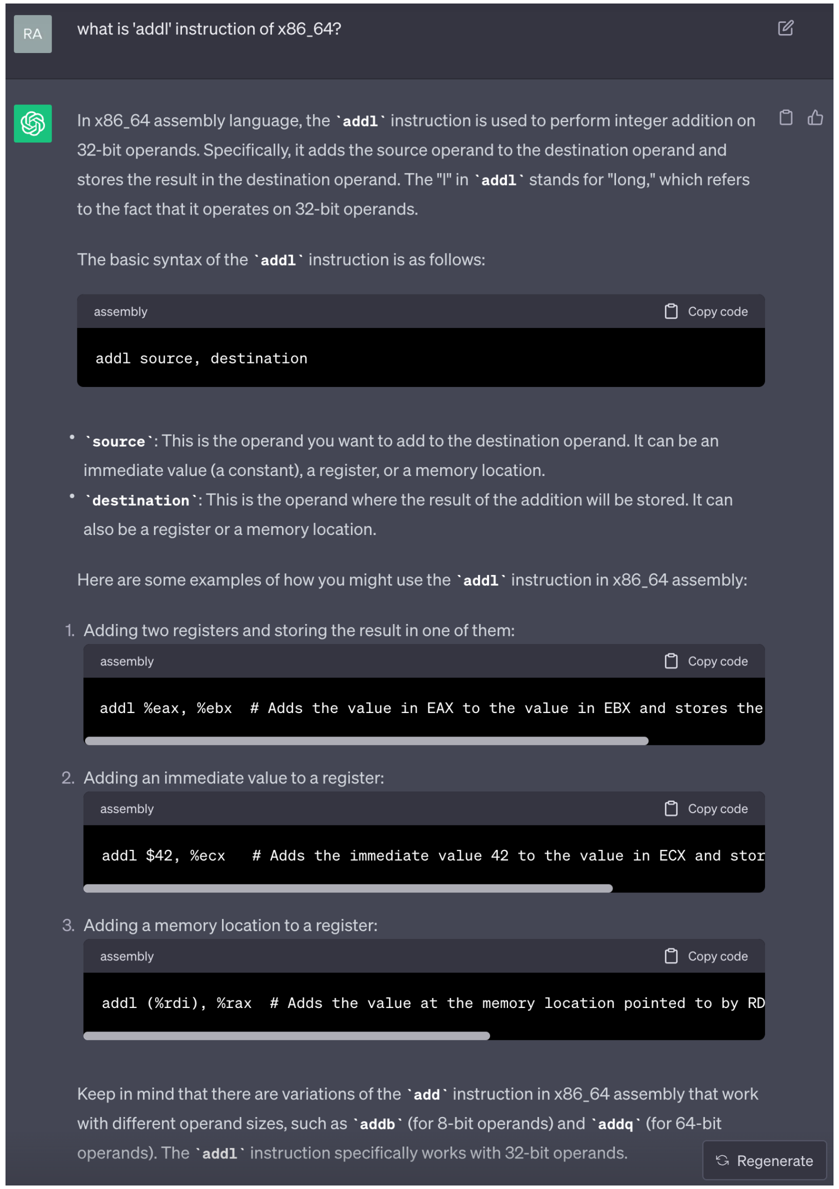
Appendix B. x86 명령들

Common **x86 Instructions**. **Instructions**은 테이블 형태로 설명을 하도록 할 것이다 ... 이것은 movl (%esp), R/M 후 **addl** $4 한 것과 같은 결과를 보여준다. pushl, I/R/M ...

c9x.me
https://c9x.me › x86 › html › file_module_x86_id_5

Add (x86 Instruction Set Reference) - c9x.me

x86 assembly tutorials, x86 opcode reference, programming, pastebin with syntax highlighting. ... The ADD instruction **performs integer addition**. It evaluates the ...



what is 'addl' instruction of x86_64?

In x86_64 assembly language, the `addl` instruction is used to perform integer addition on 32-bit operands. Specifically, it adds the source operand to the destination operand and stores the result in the destination operand. The "l" in `addl` stands for "long," which refers to the fact that it operates on 32-bit operands.

The basic syntax of the `addl` instruction is as follows:

```assembly
addl source, destination
```

- `source`: This is the operand you want to add to the destination operand. It can be an immediate value (a constant), a register, or a memory location.
- `destination`: This is the operand where the result of the addition will be stored. It can also be a register or a memory location.

Here are some examples of how you might use the `addl` instruction in x86_64 assembly:

1. Adding two registers and storing the result in one of them:

```assembly
addl %eax, %ebx  # Adds the value in EAX to the value in EBX and stores the
```

2. Adding an immediate value to a register:

```assembly
addl $42, %ecx   # Adds the immediate value 42 to the value in ECX and stor
```

3. Adding a memory location to a register:

```assembly
addl (%rdi), %rax  # Adds the value at the memory location pointed to by RD
```

Keep in mind that there are variations of the `add` instruction in x86_64 assembly that work with different operand sizes, such as `addb` (for 8-bit operands) and `addq` (for 64-bit operands). The `addl` instruction specifically works with 32-bit operands.

Assembly Language (Instructions)

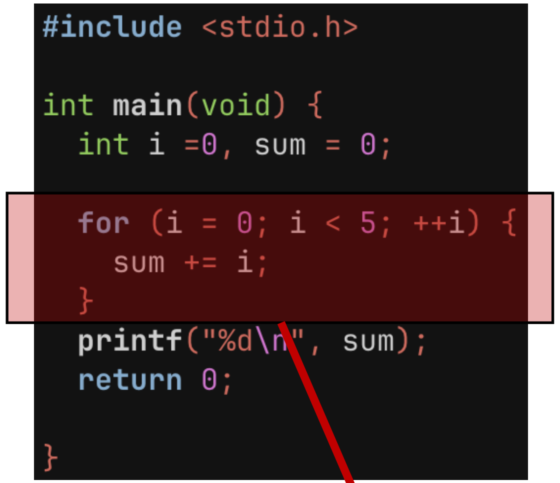
- Compare instruction (Set status flags depending on the result)
  - `cmp operand1, operand2`
    - `R = operand2 – operand1`

- Status flags
  - Carry flag (CF)
    - set if `(unsigned) R < (unsigned) operand1`
  - Zero flag (ZF)
    - set if `R == 0`
  - Sign flag (SF)
    - set if `R < 0`
  - Overflow flag (OF)
    - set if `(operand1 < 0 == operand2 < 0) && (R < 0 != operand1 < 0)`

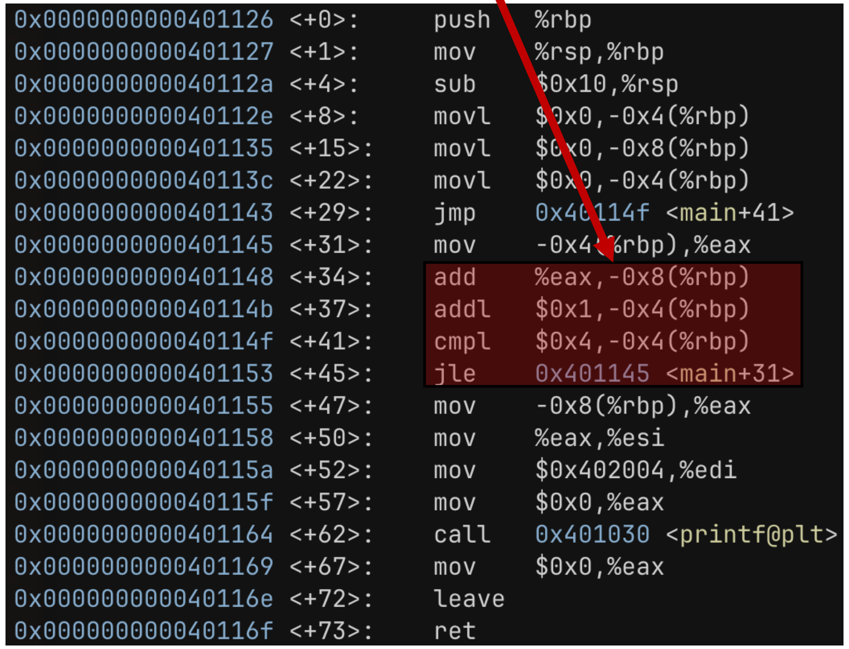
Assembly Language (Instructions)

- Jump instructions
  - Can be used to implement conditional branch

Instruction	Jump condition	Description
`jmp Label`	1	Direct jump
`jmp *operand`	1	Indirect jump
`je Label`	ZF	Equal / zero
`jne Label`	~ZF	Not equal / not zero
`jg Label`	~(SF^OF)&~ZF	Greater (>)
`jge Label`	~(SF^OF)	Greater or equal (>=)
`jl Label`	SF^OF	Less (<)
`jle Label`	(SF^OF)|ZF	Less or equal (<=)



```c
#include <stdio.h>

int main(void) {
  int i =0, sum = 0;

  for (i = 0; i < 5; ++i) {
    sum += i;
  }
  printf("%d\n", sum);
  return 0;

}
```



```
0x0000000000401126 <+0>:     push   %rbp
0x0000000000401127 <+1>:     mov    %rsp,%rbp
0x000000000040112a <+4>:     sub    $0x10,%rsp
0x000000000040112e <+8>:     movl   $0x0,-0x4(%rbp)
0x0000000000401135 <+15>:    movl   $0x0,-0x8(%rbp)
0x000000000040113c <+22>:    movl   $0x0,-0x4(%rbp)
0x0000000000401143 <+29>:    jmp    0x40114f <main+41>
0x0000000000401145 <+31>:    mov    -0x4(%rbp),%eax
0x0000000000401148 <+34>:    add    %eax,-0x8(%rbp)
0x000000000040114b <+37>:    addl   $0x1,-0x4(%rbp)
0x000000000040114f <+41>:    cmpl   $0x4,-0x4(%rbp)
0x0000000000401153 <+45>:    jle    0x401145 <main+31>
0x0000000000401155 <+47>:    mov    -0x8(%rbp),%eax
0x0000000000401158 <+50>:    mov    %eax,%esi
0x000000000040115a <+52>:    mov    $0x402004,%edi
0x000000000040115f <+57>:    mov    $0x0,%eax
0x0000000000401164 <+62>:    call   0x401030 <printf@plt>
0x0000000000401169 <+67>:    mov    $0x0,%eax
0x000000000040116e <+72>:    leave
0x000000000040116f <+73>:    ret
```

Assembly Language (Instructions)

- Other instructions
  - `lea` (Load Effective Address)
    - `lea 7(%rdx), %rdi` => `%rdi = 7 + %rdx`
    - Note that, `mov` stores value, while `lea` stores address (e.g., pointer)

Homework (Bomb Lab)

- Make sure that you enable local forwarding to access bomb server
  - See [CSED211]SSH.pdf for more details
- To download your bomb, go to http://127.0.0.1:15213
  - Enter your information, student ID and school email (we have done this during the lab session)
  - Transfer your bomb{#}.tar from the local machine to the programming2.postech.ac.kr
    - See [CSED211]SSH.pdf for more details
  - You must not re-download bomb from the server after lab session (10% degrade)
- Your goal is to defuse bomb by solving 6 phases
  - phase_1, phase_2, ..., phase_6
  - Do not cause bomb explosion frequently to avoid heavy load on the programming server
- Your score (corresponds to bomb #) will be automatically uploaded at
  - http://127.0.0.1:15214/scoreboard
  - Bomb can be defused only on programming2.postech.ac.kr
  - The score is not updated if you work on the local machine

Homework (Bomb Lab)

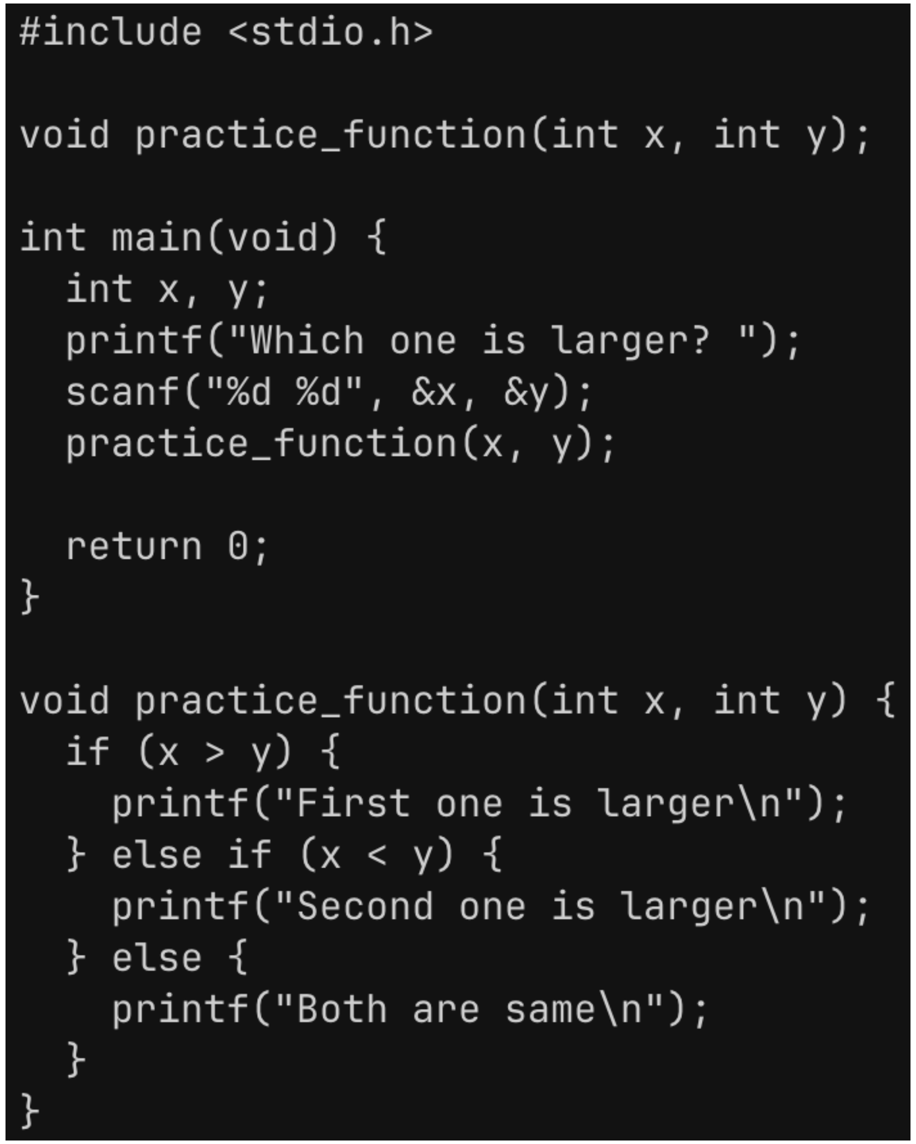
- Bomb server will close at
  - 10/16 (Wed) 23:59 (midnight)
- You can find more details in `writeup_lab3.pdf`

Homework (Report)

- Deadline: 10/16 (Wed) 23:59 (midnight)
- You need to
  - Explain how you defuse bomb in the report
  - Follow the file name format, [student #].pdf.
    - For example, 2020XXXX.pdf (No square brackets in the filename)
    - **No doc, No zip!**

Practice

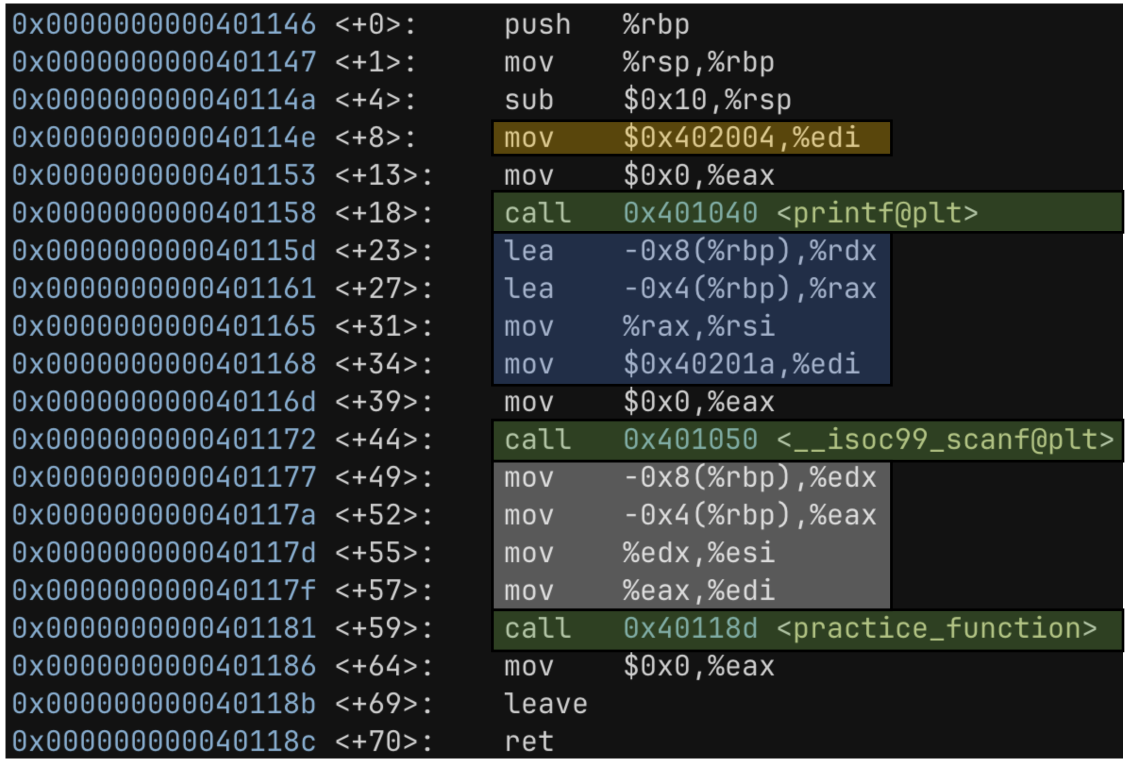
- Write the C program
  - `vi test.c`
- Compile the source code
  - `gcc -g -o test test.c`
- Run GDB over the program
  - `gdb test`



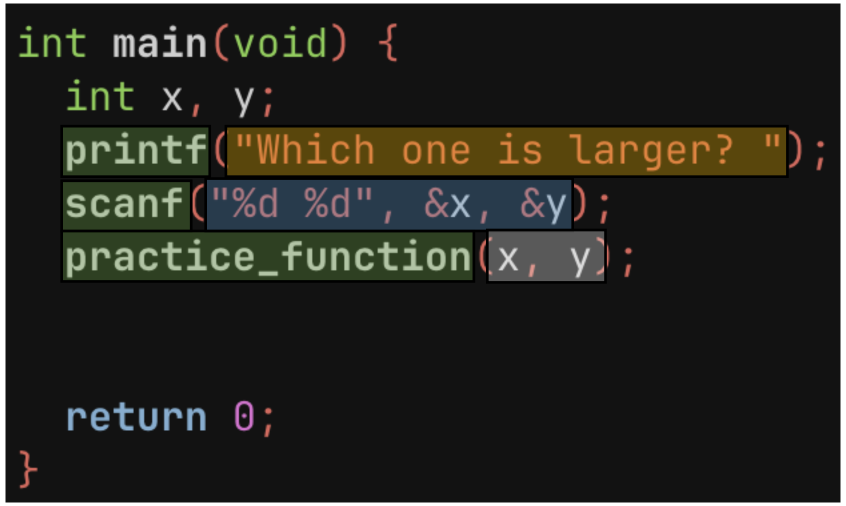
```c
#include <stdio.h>

void practice_function(int x, int y);

int main(void) {
  int x, y;
  printf("Which one is larger? ");
  scanf("%d %d", &x, &y);
  practice_function(x, y);

  return 0;
}

void practice_function(int x, int y) {
  if (x > y) {
    printf("First one is larger\n");
  } else if (x < y) {
    printf("Second one is larger\n");
  } else {
    printf("Both are same\n");
  }
}
```

Practice

- Disassemble the `main` function
  - Which are called in the main function?
  - How are arguments passed to functions?



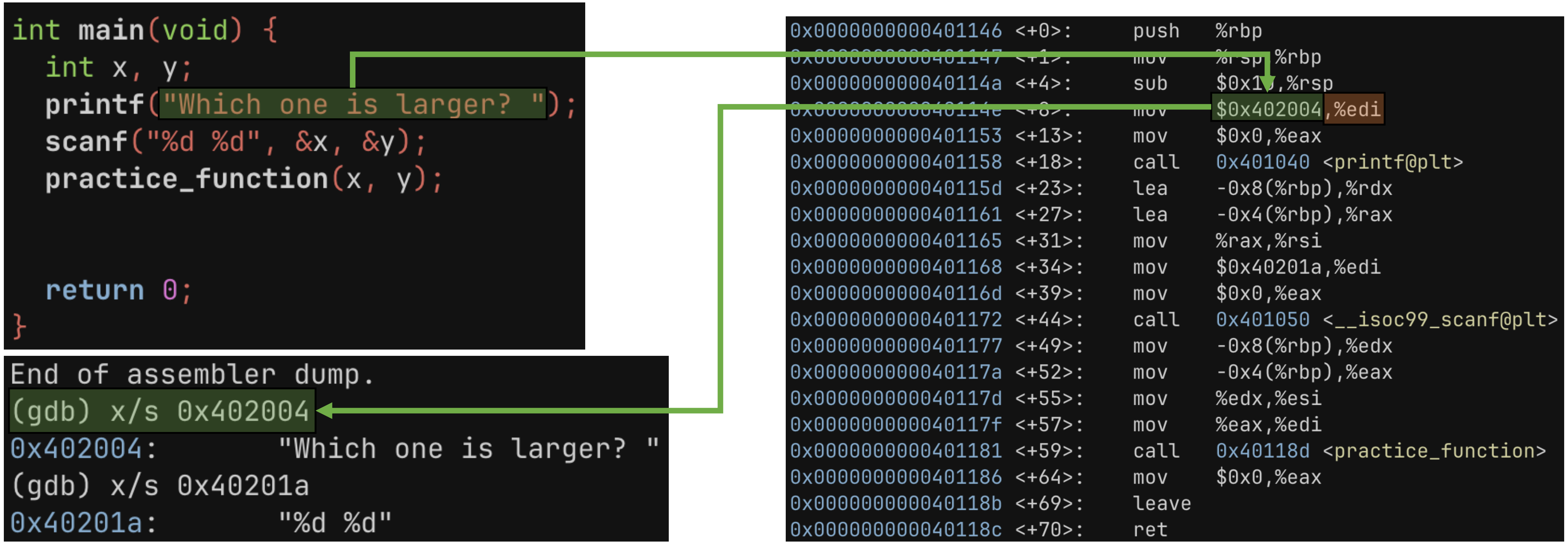
```
0x0000000000401146 <+0>:     push   %rbp
0x0000000000401147 <+1>:     mov    %rsp,%rbp
0x000000000040114a <+4>:     sub    $0x10,%rsp
0x000000000040114e <+8>:     mov    $0x402004,%edi
0x0000000000401153 <+13>:    mov    $0x0,%eax
0x0000000000401158 <+18>:    call   0x401040 <printf@plt>
0x000000000040115d <+23>:    lea    -0x8(%rbp),%rdx
0x0000000000401161 <+27>:    lea    -0x4(%rbp),%rax
0x0000000000401165 <+31>:    mov    %rax,%rsi
0x0000000000401168 <+34>:    mov    $0x40201a,%edi
0x000000000040116d <+39>:    mov    $0x0,%eax
0x0000000000401172 <+44>:    call   0x401050 <__isoc99_scanf@plt>
0x0000000000401177 <+49>:    mov    -0x8(%rbp),%edx
0x000000000040117a <+52>:    mov    -0x4(%rbp),%eax
0x000000000040117d <+55>:    mov    %edx,%esi
0x000000000040117f <+57>:    mov    %eax,%edi
0x0000000000401181 <+59>:    call   0x40118d <practice_function>
0x0000000000401186 <+64>:    mov    $0x0,%eax
0x000000000040118b <+69>:    leave
0x000000000040118c <+70>:    ret
```



```c
int main(void) {
   int x, y;
   printf("Which one is larger? ");
   scanf("%d %d", &x, &y);
   practice_function(x, y);


   return 0;
}
```

Practice

- How to inspect arguments of `printf`?
  - `printf` takes the first argument as string
  - The first argument is stored in `%edi`



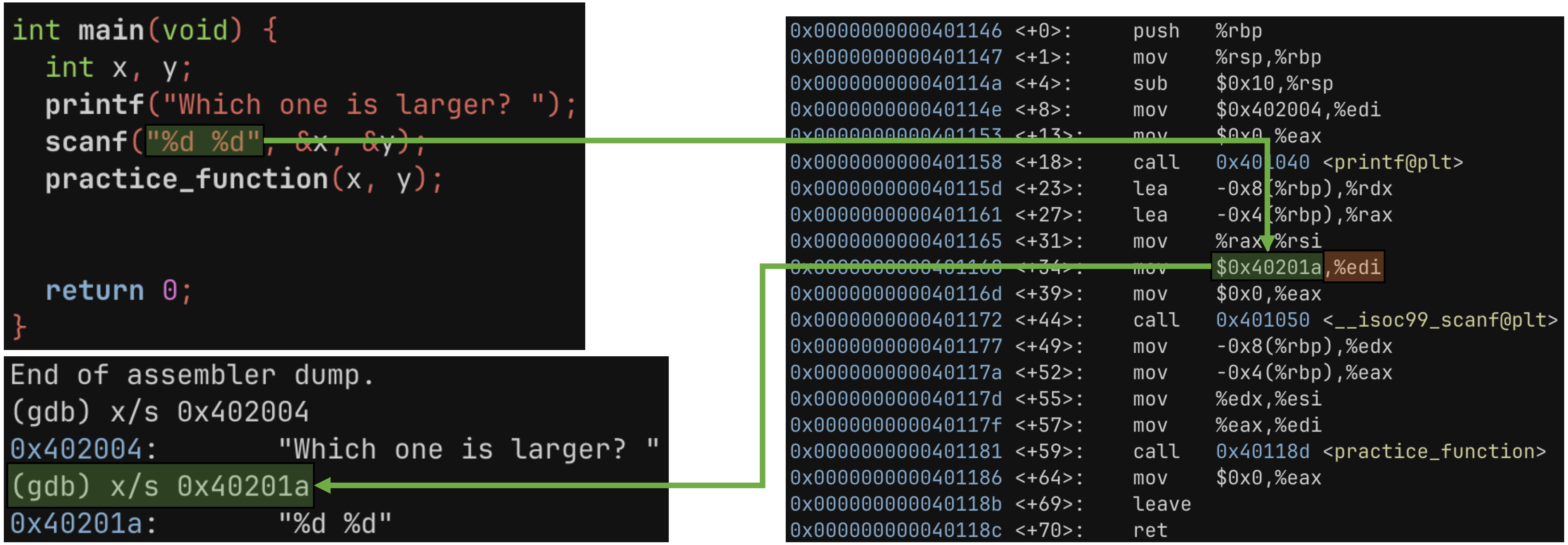
```c
int main(void) {
  int x, y;
  printf("Which one is larger? ");
  scanf("%d %d", &x, &y);
  practice_function(x, y);


  return 0;
}
```

```
End of assembler dump.
(gdb) x/s 0x402004
0x402004:       "Which one is larger? "
(gdb) x/s 0x40201a
0x40201a:       "%d %d"
```

```
0x0000000000401146 <+0>:     push   %rbp
0x0000000000401147 <+1>:     mov    %rsp,%rbp
0x000000000040114a <+4>:     sub    $0x10,%rsp
0x000000000040114e <+8>:     mov    $0x402004,%edi
0x0000000000401153 <+13>:    mov    $0x0,%eax
0x0000000000401158 <+18>:    call   0x401040 <printf@plt>
0x000000000040115d <+23>:    lea    -0x8(%rbp),%rdx
0x0000000000401161 <+27>:    lea    -0x4(%rbp),%rax
0x0000000000401165 <+31>:    mov    %rax,%rsi
0x0000000000401168 <+34>:    mov    $0x40201a,%edi
0x000000000040116d <+39>:    mov    $0x0,%eax
0x0000000000401172 <+44>:    call   0x401050 <__isoc99_scanf@plt>
0x0000000000401177 <+49>:    mov    -0x8(%rbp),%edx
0x000000000040117a <+52>:    mov    -0x4(%rbp),%eax
0x000000000040117d <+55>:    mov    %edx,%esi
0x000000000040117f <+57>:    mov    %eax,%edi
0x0000000000401181 <+59>:    call   0x40118d <practice_function>
0x0000000000401186 <+64>:    mov    $0x0,%eax
0x000000000040118b <+69>:    leave
0x000000000040118c <+70>:    ret
```

Practice

- How to inspect arguments of `scanf`?
  - `scanf` takes the first argument as string
  - The first argument is stored in `%edi`



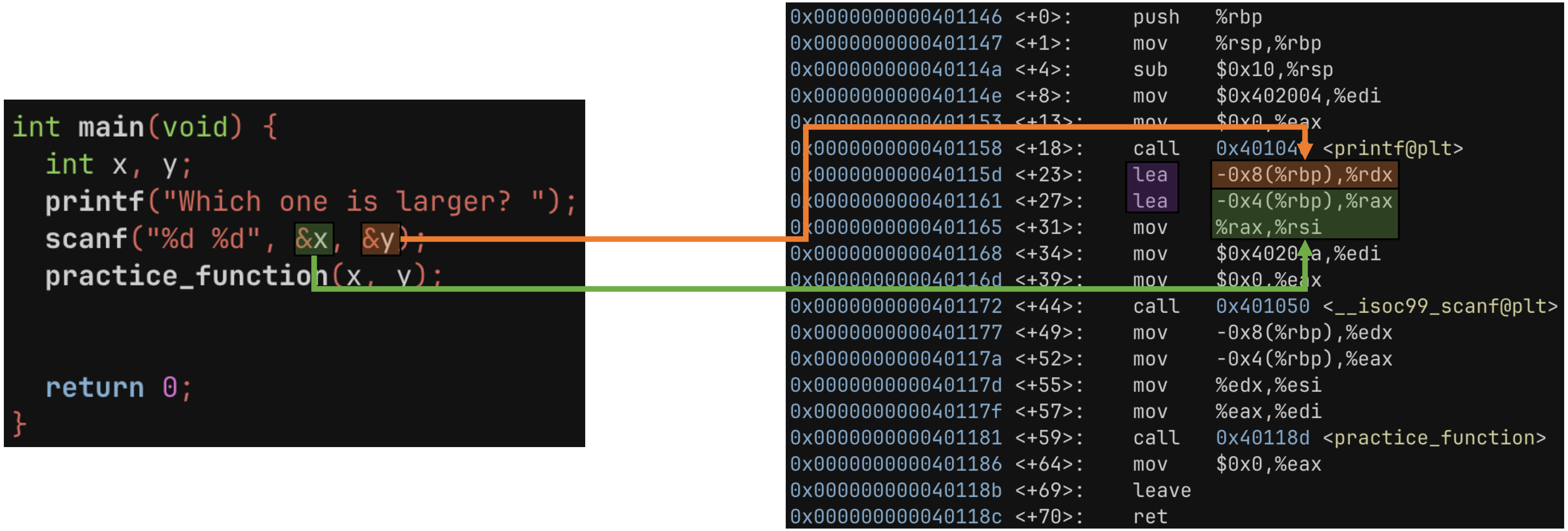
```
int main(void) {
  int x, y;
  printf("Which one is larger? ");
  scanf("%d %d", &x, &y);
  practice_function(x, y);


  return 0;
}
```

```
End of assembler dump.
(gdb) x/s 0x402004
0x402004:       "Which one is larger? "
(gdb) x/s 0x40201a
0x40201a:       "%d %d"
```

```
0x0000000000401146 <+0>:     push   %rbp
0x0000000000401147 <+1>:     mov    %rsp,%rbp
0x000000000040114a <+4>:     sub    $0x10,%rsp
0x000000000040114e <+8>:     mov    $0x402004,%edi
0x0000000000401153 <+13>:    mov    $0x0,%eax
0x0000000000401158 <+18>:    call   0x401040 <printf@plt>
0x000000000040115d <+23>:    lea    -0x8(%rbp),%rdx
0x0000000000401161 <+27>:    lea    -0x4(%rbp),%rax
0x0000000000401165 <+31>:    mov    %rax,%rsi
0x0000000000401168 <+34>:    mov    $0x40201a,%edi
0x000000000040116d <+39>:    mov    $0x0,%eax
0x0000000000401172 <+44>:    call   0x401050 <__isoc99_scanf@plt>
0x0000000000401177 <+49>:    mov    -0x8(%rbp),%edx
0x000000000040117a <+52>:    mov    -0x4(%rbp),%eax
0x000000000040117d <+55>:    mov    %edx,%esi
0x000000000040117f <+57>:    mov    %eax,%edi
0x0000000000401181 <+59>:    call   0x40118d <practice_function>
0x0000000000401186 <+64>:    mov    $0x0,%eax
0x000000000040118b <+69>:    leave
0x000000000040118c <+70>:    ret
```

Practice

- How to inspect arguments of `scanf`?
  - 2nd and 3rd arguments are passed to `rsi` and `rdi`, respectively
  - Why use `lea` instructions?
    - Because `&x` and `&y` are pointers (e.g., pointing to addresses of x and `y`)

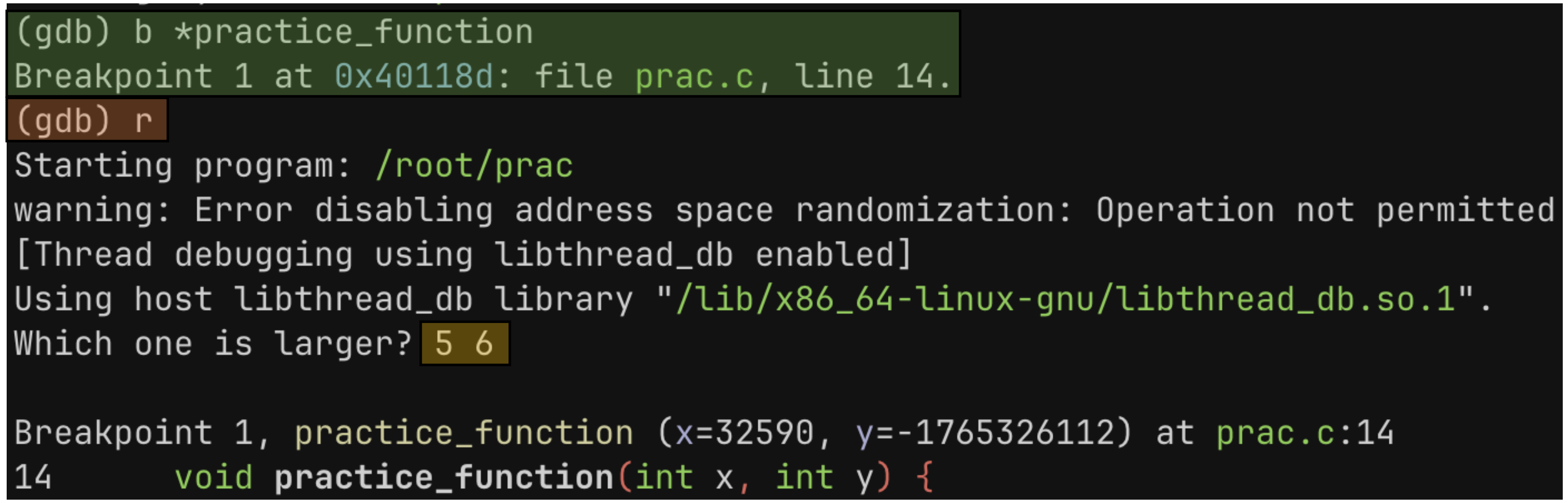


```c
int main(void) {
  int x, y;
  printf("Which one is larger? ");
  scanf("%d %d", &x, &y);
  practice_function(x, y);


  return 0;
}
```

```
0x0000000000401146 <+0>:     push   %rbp
0x0000000000401147 <+1>:     mov    %rsp,%rbp
0x000000000040114a <+4>:     sub    $0x10,%rsp
0x000000000040114e <+8>:     mov    $0x402004,%edi
0x0000000000401153 <+13>:    mov    $0x0,%eax
0x0000000000401158 <+18>:    call   0x401040 <printf@plt>
0x000000000040115d <+23>:    lea    -0x8(%rbp),%rdx
0x0000000000401161 <+27>:    lea    -0x4(%rbp),%rax
0x0000000000401165 <+31>:    mov    %rax,%rsi
0x0000000000401168 <+34>:    mov    $0x40200a,%edi
0x000000000040116d <+39>:    mov    $0x0,%eax
0x0000000000401172 <+44>:    call   0x401050 <__isoc99_scanf@plt>
0x0000000000401177 <+49>:    mov    -0x8(%rbp),%edx
0x000000000040117a <+52>:    mov    -0x4(%rbp),%eax
0x000000000040117d <+55>:    mov    %edx,%esi
0x000000000040117f <+57>:    mov    %eax,%edi
0x0000000000401181 <+59>:    call   0x40118d <practice_function>
0x0000000000401186 <+64>:    mov    $0x0,%eax
0x000000000040118b <+69>:    leave
0x000000000040118c <+70>:    ret
```

Practice

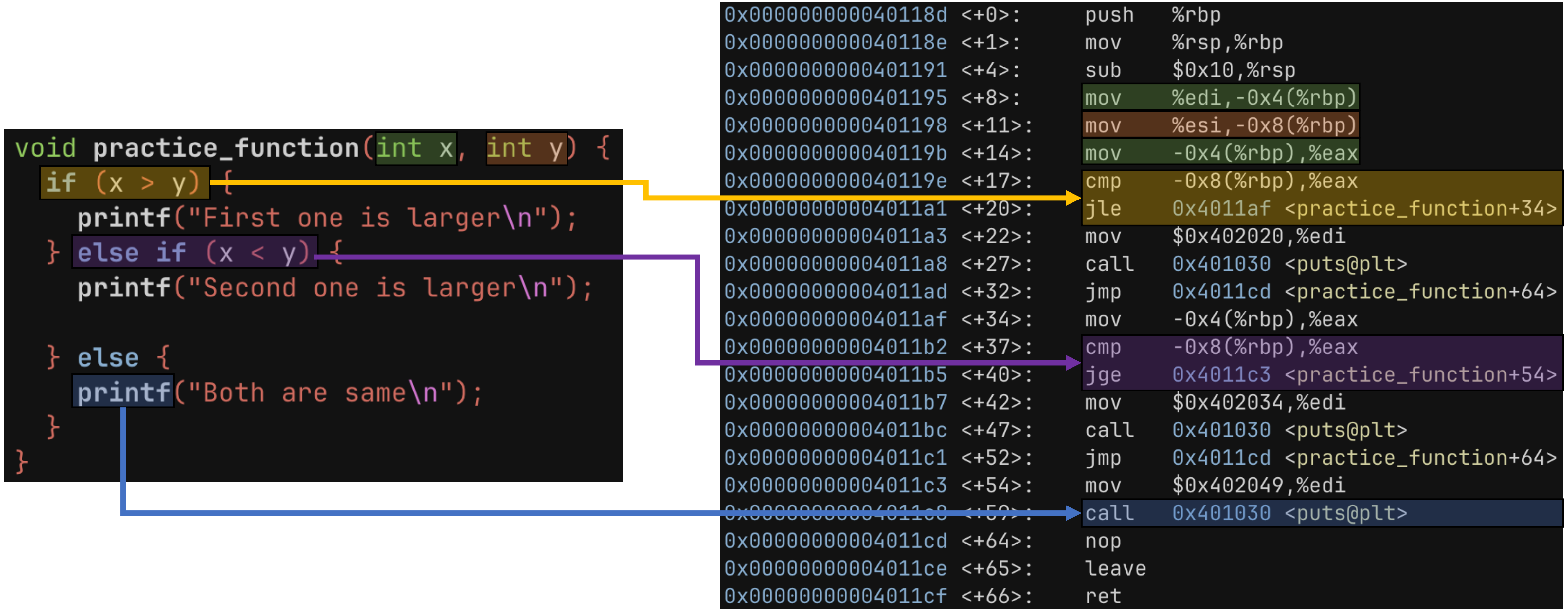
- Before exploring `practice_function`, make breakpoint at `practice_function`
- Run program using `r`
  - Provide the program with two integer arguments
- Now, we can inspect register values written right before calling `practice_function`



```
(gdb) b *practice_function
Breakpoint 1 at 0x40118d: file prac.c, line 14.
(gdb) r
Starting program: /root/prac
warning: Error disabling address space randomization: Operation not permitted
[Thread debugging using libthread_db enabled]
Using host libthread_db library "/lib/x86_64-linux-gnu/libthread_db.so.1".
Which one is larger? 5 6

Breakpoint 1, practice_function (x=32590, y=-1765326112) at prac.c:14
14        void practice_function(int x, int y) {
```

Practice

- How to inspect branch statements?
  - x: %edi -> -0x4(%rbp), y: %esi -> -0x8(%rbp)



```c
void practice_function(int x, int y) {
    if (x > y) {
        printf("First one is larger\n");
    } else if (x < y) {
        printf("Second one is larger\n");

    } else {
        printf("Both are same\n");
    }
}
```

```
0x000000000040118d <+0>:     push   %rbp
0x000000000040118e <+1>:     mov    %rsp,%rbp
0x0000000000401191 <+4>:     sub    $0x10,%rsp
0x0000000000401195 <+8>:     mov    %edi,-0x4(%rbp)
0x0000000000401198 <+11>:    mov    %esi,-0x8(%rbp)
0x000000000040119b <+14>:    mov    -0x4(%rbp),%eax
0x000000000040119e <+17>:    cmp    -0x8(%rbp),%eax
0x00000000004011a1 <+20>:    jle    0x4011af <practice_function+34>
0x00000000004011a3 <+22>:    mov    $0x402020,%edi
0x00000000004011a8 <+27>:    call   0x401030 <puts@plt>
0x00000000004011ad <+32>:    jmp    0x4011cd <practice_function+64>
0x00000000004011af <+34>:    mov    -0x4(%rbp),%eax
0x00000000004011b2 <+37>:    cmp    -0x8(%rbp),%eax
0x00000000004011b5 <+40>:    jge    0x4011c3 <practice_function+54>
0x00000000004011b7 <+42>:    mov    $0x402034,%edi
0x00000000004011bc <+47>:    call   0x401030 <puts@plt>
0x00000000004011c1 <+52>:    jmp    0x4011cd <practice_function+64>
0x00000000004011c3 <+54>:    mov    $0x402049,%edi
0x00000000004011c8 <+59>:    call   0x401030 <puts@plt>
0x00000000004011cd <+64>:    nop
0x00000000004011ce <+65>:    leave
0x00000000004011cf <+66>:    ret
```

Quiz