Name: SOLUTIONS	UID: CS 33

1. mov vs lea?

Describe the difference between the following:

```
movq (%rdx), %rax
leaq (%rdx), %rax
```

movq takes the **contents** of what's stored in register %rdx and moves it to %rax. leaq computes the load effective **address** and stores it in %rax. leaq analogous to returning a pointer, whereas movq is analogous to returning a dereferenced pointer.

2. Instruction Junction

a. What would be the corresponding instruction to move 64 bits of data from register %rax to register %rcx?

   movq %rax, %rcx

b. What would be the corresponding instruction to move 64 bits of data from the memory location stored in register %rax to register %rcx?

   movq (%rax), %rcx

3. What was Compiled?

Which of the functions `cool1`, `cool2`, `cool3` would compile into this assembly code?

```
movl %esi, %eax
cmpl %eax, %edi
jge .L4
movl %edi, %eax
.L4:
ret
```

```
int cool1(int a, int b)
{
    if (b < a)
        return b;
    else
        return a;


}
```

```
int cool2(int a, int b)
{
    if (a < b)
        return a;
    else
        return b;


}
```

```
int cool3(int a, int b)
{
    unsigned ub = (
       unsigned) b;
    if (ub < a)
        return a;
    else
        return ub;
}
```

cool2 was compiled

- Arguments passed to a function is stored in the %edi, %esi, etc registers
    - %edi is a and %esi is b
- When comparing, we compare as cmp Two One
    - Thus the instruction jge is checking if %edi is greater than or equal to %eax
    - This is essentially checking if $a \geq b$, which is the else condition
- We can observe that when we do jump, %eax is not updated
    - We return b in the else case
- If we don't jump, we update %eax to %edi
    - We return a in the if case
- Thus, this matches the cool2 function
- This question was inspired by a previous midterm.

4. Operand Form Practice

Assume the following values are stored in the indicated registers and memory addresses.

Address	**Value**	**Register**	**Value**
0x104	0x34	%rax	0x104
0x108	0xCC	%rcx	0x5
0x10C	0x19	%rdx	0x3
0x110	0x42	%rbx	0x4

Fill in the table for the indicated operands:

Operand	**Value**	**Value (if `lea`)**
\$0x110	0x110 (immediate value)	— (illegal syntax: must be memory addressing)
%rax	0x104 (value stored in %rax)	— (illegal syntax: must be memory addressing)
0x110	0x42 (value stored at memory address $0x110$)	— (illegal syntax: must be memory addressing)
(%rax)	0x34 (%rax $= 0x104$, memory$[0x104] = 0x34$)	0x104 (%rax is the base address, thus return $0x104$)
8(%rax)	0x19 ($\%rax = 0x104$, $0x104 + 8 = 0x10C$, memory$[0x10C] = 0x19$)	0x10C ($0x104 + 8 = 0x10C$)

Operand	**Value**	**Value (if `lea`)**
(%rax, %rbx)	0xCC (%rax $= 0x104$, %rbx $= 0x4$, $0x104 + 0x4 = 0x108$, memory$[0x108] = 0xCC$)	0x108 ($0x104 + 0x4 = 0x108$)
3(%rax, %rcx)	0x19 (%rax $= 0x104$, %rcx $= 0x5$, $3{+}0x104{+}0x5 = 0x10C$, memory$[0x10C] = 0x19$)	0x10C ($0x104 + 0x5 + 3 = 0x10C$)
256(, %rbx, 2)	0xCC (%rbx $= 0x4$, $0x100{+}(0x4{\cdot}2) = 0x108$, memory$[0x108] = 0xCC$)	0x108 ($256 + 2 \cdot \%rbx = 0x108$)
(%rax, %rbx, 2)	0x19 (%rax $= 0x104$, %rbx $= 0x4$, $0x104 + (0x4 \cdot 2) = 0x10C$, memory$[0x10C]$ $= 0x19$)	0x10C (%rax $+$ $2 \cdot \%rbx = 0x10C$)

- $ denotes immediates
- Note: any numbers starting with "0x" are hexadecimal numbers!!
- All of the operands can be evaluated using the specific formulas on page 181 in the textbook
- More generally, whenever you see an address of the form $D(r_b, r_i, s)$, where D is an number, r_b and r_i are registers, and s is either 1,2,4, or 8, you can use the following formula:

  $D + R[rb] + R[ri] * s$

  If D is missing, assume $D == 0$
  If r_b is missing, assume $r_b == 0$
  If r_s is missing, assume $r_s == 0$
  If s is missing, assume $s == 0$

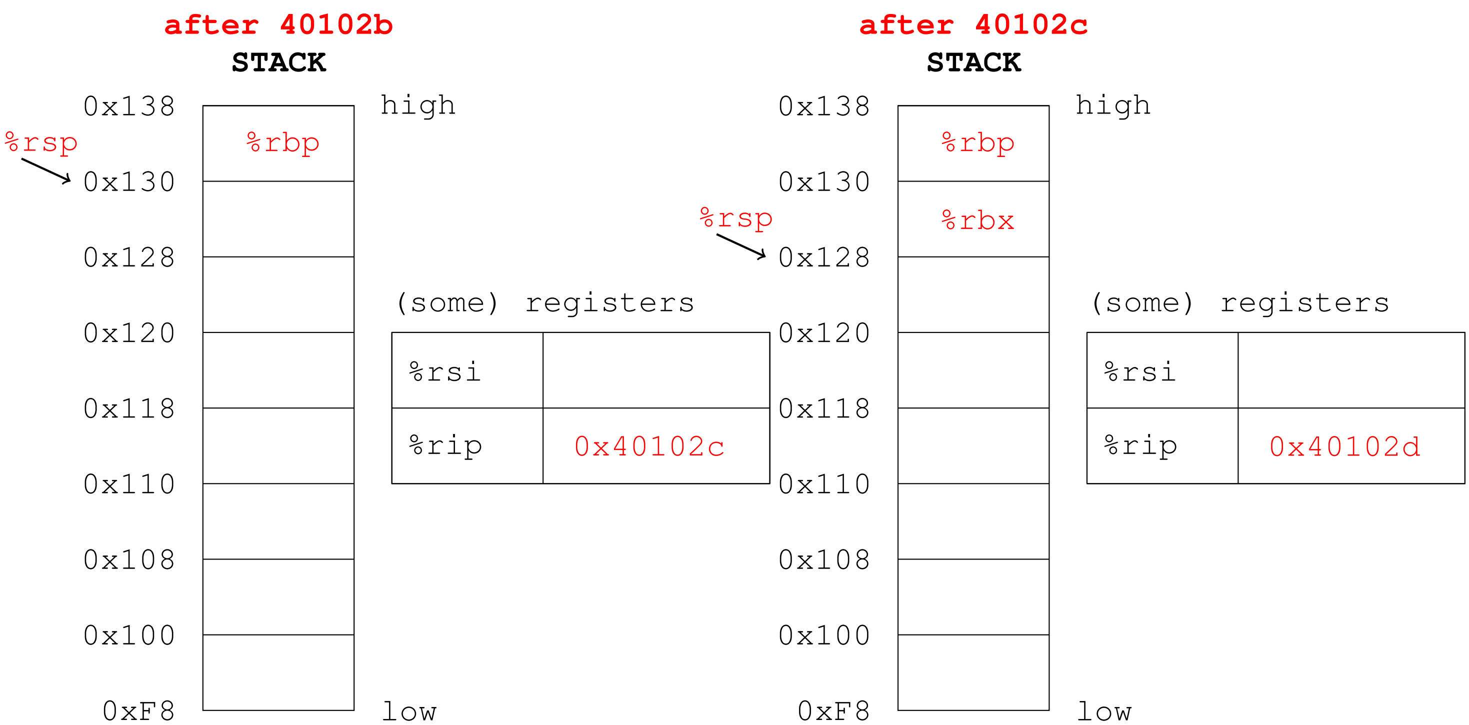
- For more practice, try practice problem 3.1 on page 182 of the textbook

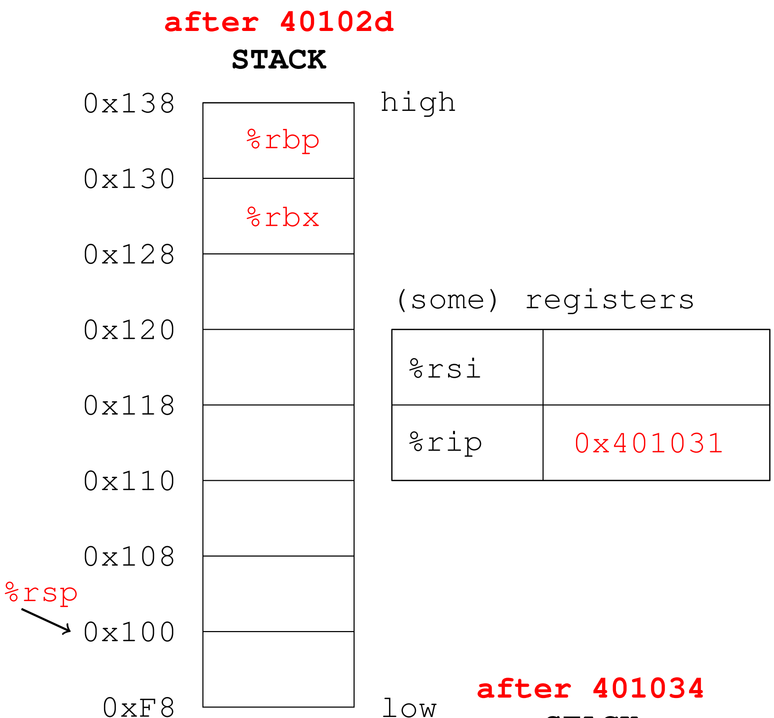
5. Disassembled Function

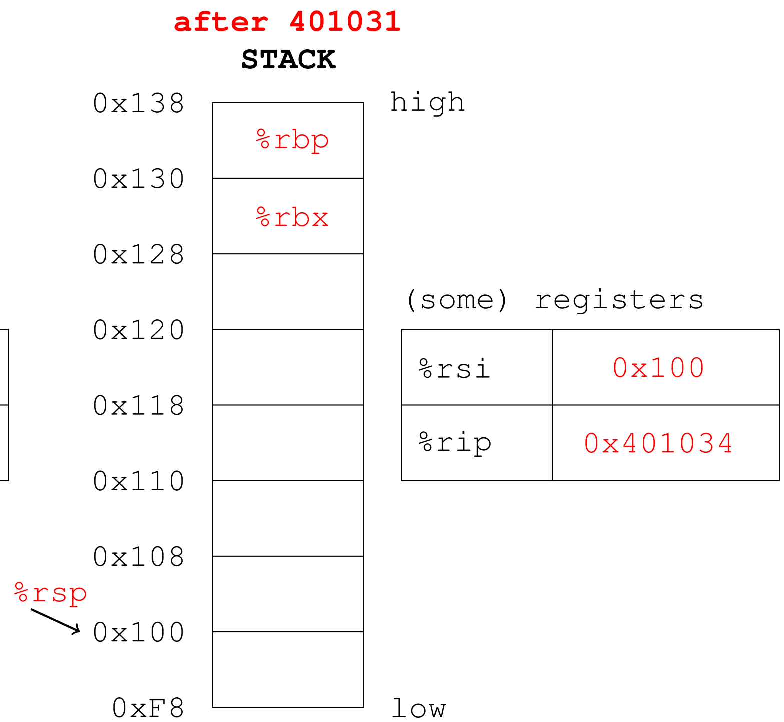
Consider the following disassembled function:

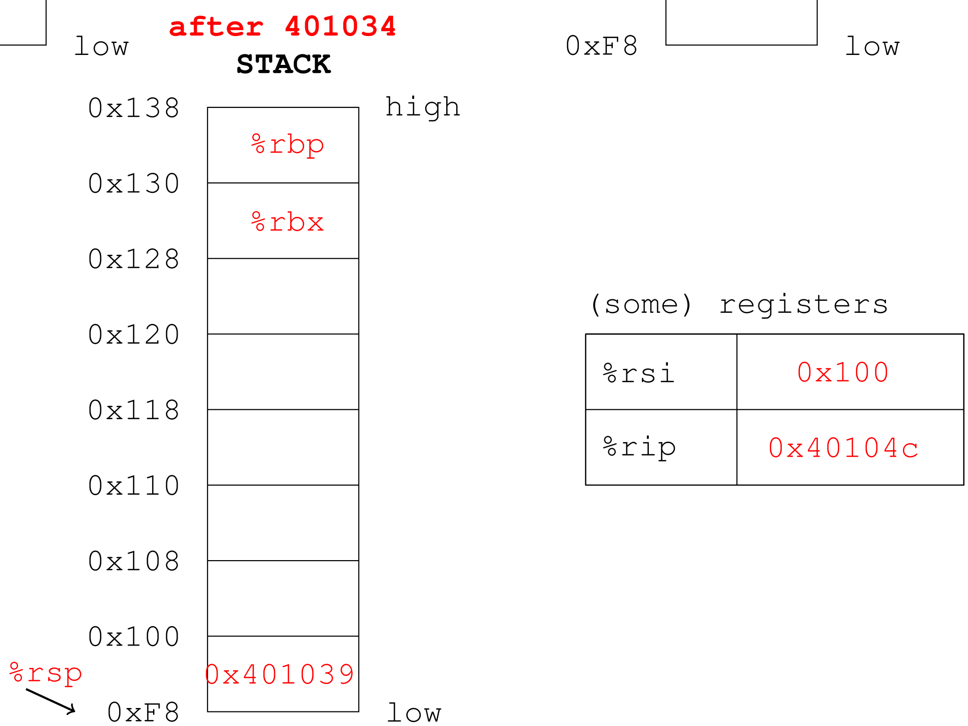
```
000000000040102b <phase_2>:
40102b: 55                    push %rbp
40102c: 53                    push %rbx
40102d: 48 83 ec 28           sub $0x28,%rsp
401031: 48 89 e6              mov %rsp,%rsi
401034: e8 e3 03 00 00        callq 40141c <read_six_numbers>
401039: 83 3c 24 01           cmpl $0x1,(%rsp)
```

(a) Assume %rsp initially has a value of 0x138. Draw the stack (see example diagram below) for the execution of <phase_2>, updating the stack and register values as necessary after each line.









(b) Right after the `callq` instruction has been executed, what are the values of %rsp, %rsi, and %rip?

- Recall: pushing onto the stack DECREMENTS %rsp
- after 401034 (callq):
  - the return address (401039) gets pushed
  - %rip gets set to the callq address, %rip = 40141c
- Overall, after the callq insn the values are as follows:
  - $\%rsp = 0xF8$, $\%rip = 40141c$, $\%rsi = 0x100$

Bonus: Invalid mov Instructions

Explain why these instructions would not be found in an assembly program.

a. `movl %eax, %rdx`

destination operand has the incorrect size

b. `movb %di, 8(%rdx)`

mismatch between instruction suffix (b, 1 byte) and size of register %di (2 bytes)

c. `movq (%rsi), 8(%rbp)`

source and destination for mov cannot both be memory references, (i.e., cannot read and write to memory in the same instruction)

d. `movw $0xFF, (%eax)`

%eax (only 32 bits) cannot be used as an address register in x86-64

Bonus: Condition Codes and Jumps

Assume the addresses and registers are in the same state as in Problem 4. Does the following code result in a jump to `.L2`?

```
leaq (%rax, %rbx), %rdi
cmpq $0x100, %rdi
jg .L2
```

Yes.

- First line will put $0x104 + 0x4$ into %rdi
- Second line sets condition codes according to $0x108$–$0x100$, which sets no condition codes
- Since jg is evaluated as $(\sim (SF\,\hat{}\,OF))\&(\sim ZF)$ which in this case is evaluates to 1, we will jump