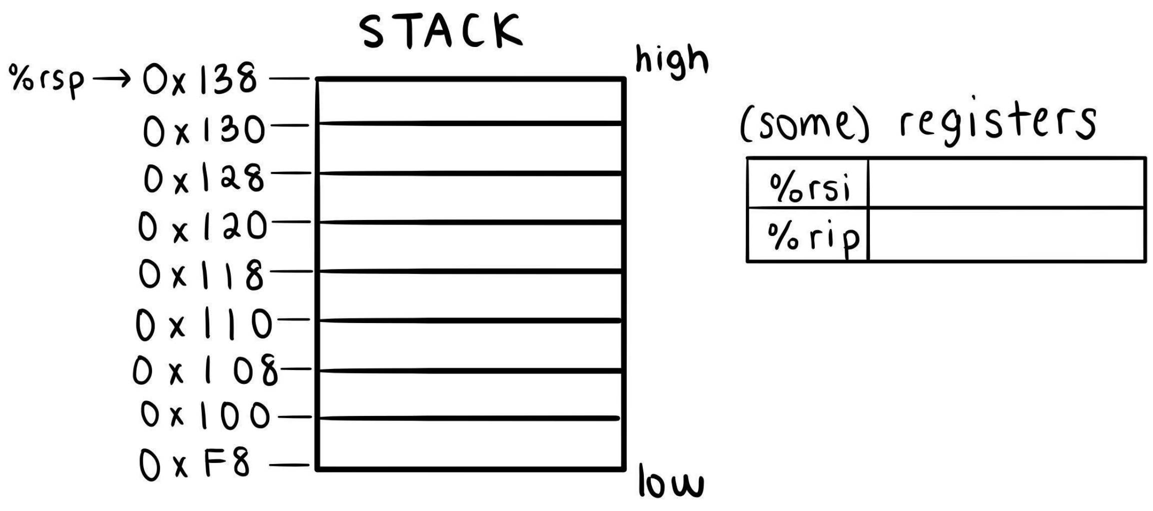
1. Consider the following disassembled function:

```
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)
  …
```

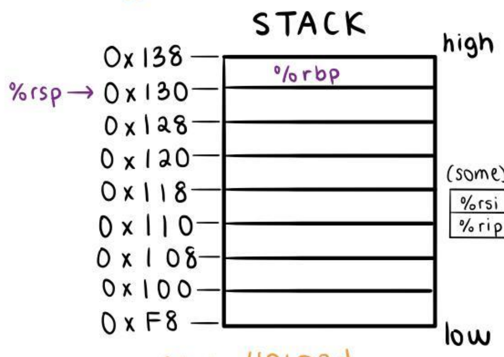
i) Assume `%rsp` initially has a value of 0x138. Draw the stack (see example diagram below) for the execution of `<phase_2>`, updating the stack and register values after each line is executed.



ii) 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

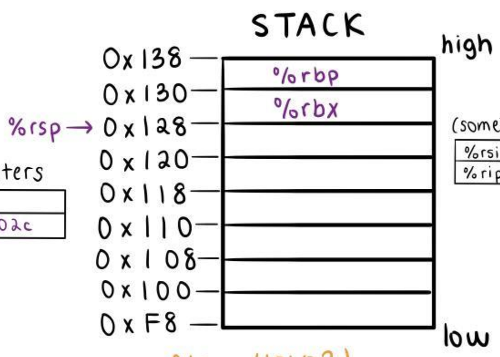
after 40102b



(some) registers

%rsi	
%rip	0x40102c

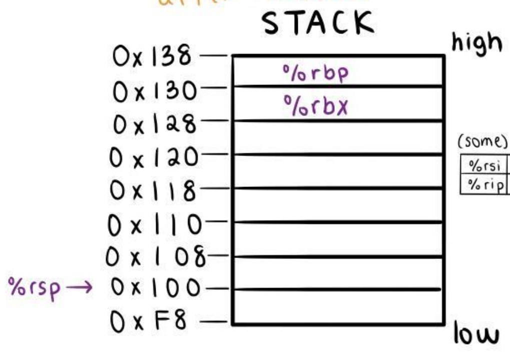
after 40102c



(some) registers

%rsi	
%rip	0x40102d

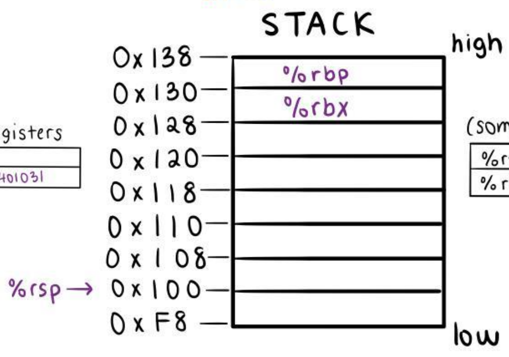
after 40102d



(some) registers

%rsi	
%rip	0x401031

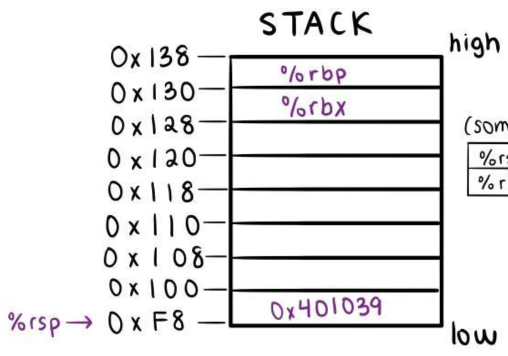
after 401031



(some) registers

%rsi	0x100
%rip	0x401034

after 401034



(some) registers

%rsi	0x100
%rip	0x40141c

2. What will the following print out?

```
typedef struct {
      char shookie;
      int tata;
      char cookie;
      double chimmy;
} bt;

void main(int argc, char** argv){
      bt band[7];
      printf( "%d\n", (int)sizeof(band));
}

(1 + (3) + 4 + 1 + (7) + 8) * 7 = 168
Due to alignment, we need to add the numbers in parentheses
```

3. What is the best* ordering of the following variables if you want to have a struct that uses all of them? What would be the optimal size? Assume a 64-bit architecture with 4-byte ints.

** the ordering that will result in the optimal usage of space.*

```
char tully;
long stark;
float* lannister;
double targaryen;
int greyjoy;
float arryn;     // hint: floats are 4 bytes

// want to order from largest size to smallest, as structs are
// x-aligned, where x is the size of the largest data type in
// the struct

struct Westeros{
      float* lannister;      // ALL pointers are 8 bytes
      double targaryen;      // doubles are 8 bytes
      long stark;            // longs are 8 bytes
      float arryn;           // floats are 4 bytes
      int greyjoy;           // ints are 4 bytes
      char tully;            // chars are 1 byte

      // Note: this is one possible ordering, but there are many
others that work as well!
};
```

4. 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)
  …
```

Right after the callq instruction has been executed (i.e., your current execution address is 40141c), what address will be at the top of the stack?

401039.

- When executing a call instruction, you push the return address onto the stack
    - The instruction pointer (%rip) points to the next instruction to execute
    - In this case, 401039
- When you reach the ret instruction in read_six_numbers, you will pop this address off the stack so control will return to the next instruction in phase_2.

5. Consider the following C code:

```
typedef struct {
      char first;
      int second;
      short third;
      int* fourth;
} stuff;

stuff array[5];

int func0(int index, int pos, long dist) {
      char* ptr = (char*) &(array[index].first);
      ptr += pos;
      *ptr = index + dist;

      return *ptr;
}

int func1() {
      int x = func0(1, 4, 12);
      return x;
}
```

Clearly some code is missing - your job is to fill in the blanks! Note that the size of the blanks is not significant. The two functions will be compiled using the following assembly code:

```
0000000000400492 <func0>:
  400492:  8d 04 17                lea    (%rdi,%rdx,1),%eax
  400495:  48 63 ff                movslq %edi,%rdi
  400498:  48 63 f6                movslq %esi,%rsi
  40049b:  48 8d 14 7f             lea    (%rdi,%rdi,2),%rdx
  40049f:  88 84 d6 60 10 60 00    mov    %al,0x601060(%rsi,%rdx,8)
  4004a6:  0f be c0                movsbl %al,%eax
  4004a9:  c3                      retq

00000000004004aa <func1>:
  4004aa:  c6 05 cb 0b 20 00 0d    movb   $0xd,0x200bcb(%rip)
                                               # 60107c <array+0x1c>
  4004b1:  b8 0d 00 00 00          mov    $0xd,%eax
  4004b6:  c3                      retq
```

```
The answer can be derived by tackling func0 first, then func1
```
func0

- From instruction 400492, we can see that the return value is set to %rdi + %rdx, where %rdi is index and %rdx is dist
    - %rdi is set to the first parameter, %rsi to the second parameter, %rdx to the third
    - %eax is unchanged, until instruction 4004a6 with %al
        - This makes sense, since we're returning the value from dereferencing a pointer to a char, aka a single byte (%al is a single byte)
    - Thus we know ***ptr = index + dist**
- From instruction 40049b:
    - %rdx is set to 3 * %rdi
    - %rdx is thus 3 * index
- From instruction 40049f:
    - 0x601060 is presumably the start of the array
        - This is confirmed in instruction 4004aa, where 60107c is shown to be <array+0x1c>
    - The destination of instruction 40049f is thus:
        - (Start of the array) +
          8 * (3 * %rdi) +
          pos
        - = (start of array) + (24 * index) + pos
    - Each object of type stuff is 24 bytes (alignment)
    - ptr from func0 is thus pointing to **array[index].first**
        - The "+ pos" comes from the second line of func0

func1

- **(note) there is no call to func0, as this code was produced from gcc -O**
    - Optimization has not been covered yet, but in the spirit of the problem, we needed the parameters passed to func0 to be hidden but the return value to be known. The non-optimization generated assembly would have done the opposite.
    - From Week3 Lecture slides "data_examples.pdf", students should understand that 0x200bcb(%rip) from instruction 4004aa is location <array + 0x1c>
    - 0x1c = 28
    - Since each object of type stuff is 24 bytes, we know the second parameter (pos) was called with value 4
        - array[1].first would be at byte 24
        - ptr += 4 would bring us to 28
        - Thus we know **pos = 28 - 24 = 4**
- 0xd = 13
    - Thus we know that the **third parameter (dist) was called with value 12**