We used the iLab machine vi.cs.rutgers.edu using the -m32 tag.

1. The content of the stack is that when we entered the signal handler, two stack frames are added to the top of the code in main, sigreturn and the signal handler. After going through the signal handler, the stack would pop that out and go to sigreturn before it would go back to main again. The code would run the signal handler and then run sigreturn. Therefore since the only place that would be going back to main would be in sigreturn, the instruction pointer would be stored in that stack frame. In addition, the return address and previous stack pointer in the signal handler was the location of signum, since that code triggered the signal handler. The stack and base pointer would go from main, push sigreturn , go to that frame, push the signal handler, go to that frame, execute that frame, pop the signal handler, return to sigreturn where signum is stored as the return address, execute that frame, pop that frame, and then return to main where it would find the instruction pointer and then resume from there. The return address is also pushed before the base pointer.

2. To find this, we first ran `backtrace` to see that two stack frames were added on top of main (sigreturn and the signal handler) after entering signal handler. There we ran `info frame` when entering the signal handler. In eax was the value of signum (11), and in the command `info frame`, it also gave the previous stack pointer’s location (the previous stack pointer of sigreturn). After printing that out ie.(`x/d 0xfffcbf0) and realizing it also had 11, we realized it was the location of signum since it was also the previous stack pointer and the return address of the signal handler. From there we ran the command `info frame 1` which gives us the stack frame of sigreturn. Since sigreturn would pop to the main, we realized a saved instruction pointer in here would have the return instruction pointer in main. Running `info frame`, we found a saved eip and then we ran ie( `x/i \*0xffffcc2c` )where we dereferenced the pointer and saw it had the next instruction in main and found the location of the instruction pointer in main. We dereferenced the return instruction to make sure it was pointing at the line that caused the segmentation fault. The next step would be to change that return instruction pointer to skip to the next instruction to avoid a segmentation fault.

3. First, we get the address of signum and dereferenced it to get a pointer to the stack frame. Next, we subtracted the two memory addresses (return instruction pointer-signum location) and found it was 60. However, since we used int pointers for signum, we realized it would take up 4 bytes per each memory address and to find the offset, we needed to find how many memory addresses away are in 60 bytes. To do that, we divided by 4(60/4) and got 15. Next, the changes to get the desired result was to add 15 to the memory address of signum pointer so it points to where the instruction pointer is. Next, we dereference it to manipulate the instruction pointer and added the length of the instruction that caused the segmentation fault (5) to the dereferenced pointer so that the pointer now points to the next line of code. We found the length of the instruction pointer by seeing that to skip the instruction it would be +5 by using the `disas` command and seeing the length of skipping the two lines of instructions.