1.4.1) What are the contents in the stack? Feel free to describe your understanding

Upon execution of the code, the stack contains various elements, including local variables, function parameters, and return addresses. In particular, the stack contains the following:

- Local variables from the ‘main’ function, including an integer variable labeled r2.
- A return address, which is the address to resume the execution after the main function has completed.
- Once the segmentation fault happens and the signal handler is invoked, relevant information, including the signal number, has been stored into the stack.

1.4.2) Where is the program counter, and how did you use GDB to locate the PC?

The Program Counter is a register that is holding the address of the next instruction to be executed, so when the signal handler is invoked, it has been pushed onto the stack. In order to determine its location utilizing the GDB, the following has been done:

- The code has been compiled first.
  - gcc -o stack stack.c -m32
- Start the GDB.
  - gdb ./stack
- Set a breakpoint at the signal handler.
  - break signal_handle
- Run the code.
  - run
- Fetch the information of the stack frame.
  - info frame
  - From this step, we get information about the saved registers, including the register “eip” which stands for Extended Information Pointer.
  - In x86, this register is considered the Program Counter. Thus, we successfully located the Program Counter from the stack frame utilizing GDB.

1.4.3) What were the changes to get the desired result?

To achieve the desired outcome, it is first needed to be known both the location of the erroneous input and the program counter. The signal handler's input parameter, signalno, served as the key to identifying the address of the faulty instruction. By setting an integer pointer variable to the address of signalno in the signal handler, the specific signal type causing the segmentation fault was determined:

  - int* address = &signalno;

The program counter's location was already established in the previous step. However, within the stack frame, another register known as "ebp" or the frame pointer is saved onto the frame, which mirrors the address of the flawed input. To redirect the pointer variable to the program counter's location, it needed to be adjusted by the offset between the frame pointer and the program counter. This offset, calculated to be 14, required an additional increment of one to encompass the starting point of the program counter:

- address = address + 15;

Now, with the pointer associated with the program counter's address, achieving the desired result means augmenting the value at that address by the length of the offending instruction. As the offending instruction was an integer variable, occupying 4 bits, the addition of 4 bits, along with an increment of 1 to signify the start of the next instruction, leads to the following line of instruction:

- *address = *address + 5;

With that, the code can be executed again, leading to the desired input.

2.2) In your report, describe how you implemented the bit operations

Function 1: Extracting the top order bits.

To accomplish this task, I employed a for loop that iterates over the specified number of bits, starting from the rightmost end of the top-order bits.

To determine the value of a bit at a given position, the following expression is executed to generate a bitmask with only the i-th bit set:

- 1 << (sizeof(unsigned int) * 8 - 1 - i)

Subsequently, a bitwise AND operation is executed to ascertain whether that specific bit is set or not:

- value & (1 << (sizeof(unsigned int) * 8 - 1 - i))

If the bit is set, it increments an integer variable, 'sum,' accordingly, capturing the binary value associated with that bit. Following the completion of the for loop, the function concludes by returning the cumulative value of the top-order bits.

Function 2: Setting a bit at an index.

To execute this operation, I determine the byte index corresponding to the bit specified for setting. This is achieved by right-shifting the index by 3 bits, essentially dividing it by 8, as a byte comprises 8 bits. Subsequently, I construct a bitmask by isolating the bit position with index & 7 (equivalent to finding index mod 8 or the bit's location). Finally, I employ a bitwise OR assignment to set the bit.

- bitmap[index >> 3] |= (1 << (index & 7))

Function 3: Getting a bit at an index

To achieve this, I determine the byte index associated with the bit to be set by right-shifting the index by 3 bits (effectively dividing the index by 8, given that a byte comprises 8 bits). Subsequently, I create a bitmask using the bit's position obtained from
index & 7 (equivalent to index mod 8 index mod 8 or the bit's location). Following this, I retrieve the bit value through a bitwise AND operation:

- (bitmap[index >> 3] & (1 << (index & 7)))

I then evaluate whether this value is non-zero; if true, it indicates that the bit is set. Conversely, if it evaluates to false, the bit is not set.