1) Review the following C code segments. For each segment, discuss and write down “what is the state of these programs?” How/when is the state changing?

a)

```
int variable1 = 3;
int variable2 = 8;
variable2 = variable2 + variable1;
int variable3 = variable2 + 5;
```

The state variable declarations are creating state. The state assignment operations are storing state information into the state that we created. The addition operation is manipulating our state information. The assignment operator stores this manipulation each clock cycle. The state in this segment is pretty much exclusively state as data.

b)

```
if (x >= 0) {
    x = x + 5;
}
x |= 1;
```

The variable x is state information and the result of comparing x with 0 also results in the creation of some invisible state information. x is generall treated as state as data, but is used to generate state as control information inside the if condition. The addition of 5 and the OR operation are two different state manipulations.

```
void init_array_of_length_12(int array[12]) {
    for (int i = 0 ; i <= 12 ; i ++) {
        array[i] = 0;
    }
}
```

The array and the iterator i are our two primary state variables. Like x in the if statement above, the variable i is also used to create invisible state as control information inside the for loop’s condition. Elsewhere, i is primarily treated as an address (a form of state as indirection). i is telling us where the data we want to use is in the array. You will learn later in the course that incrementing i actually becomes a form of pointer arithmetic. Consequently, we can again see that our state information can switch between being treated as different forms of state information. Our array is always treated solely as state as data. We never know what is in the array, nor do we care.
For the following expression:

\[ g(x, y) = (x' + y)' \]

Draw a circuit for \( g(x, y) \):

![Circuit Diagram]

Fill out a truth table for \( g(x, y) \). (We’ve provided extra columns to compute \( x' \) and \( x' + y \).)

<table>
<thead>
<tr>
<th>( x )</th>
<th>( y )</th>
<th>( x' )</th>
<th>( x' + y )</th>
<th>( g(x, y) )</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>
Verilog Modules

module sc_block(s, c, a, b);
  output s, c;
  input a, b;
  wire w1, w2, not_a, not_b;

  // the "c" output is just the AND of the two inputs
  and a1(c, a, b);

  // the "s" output is 1 only when exactly one of the inputs is 1
  not n1(not_a, a);
  not n2(not_b, b);
  and a2(w1, a, not_b);
  and a3(w2, b, not_a);
  or o1(s, w1, w2);
endmodule // sc_block

Draw a circuit diagram for the Verilog module above:

![Circuit Diagram](image)

Figure 1. AND, OR, and NOT gates.
Hierarchical design: building modules from modules

**Figure 2.** The sc2_block, which is implemented using sc_block, and its partially completed truth table.

Complete the truth table and write the Verilog for the sc2_block module:

*Will be released after Lab 1’s late deadline has passed*