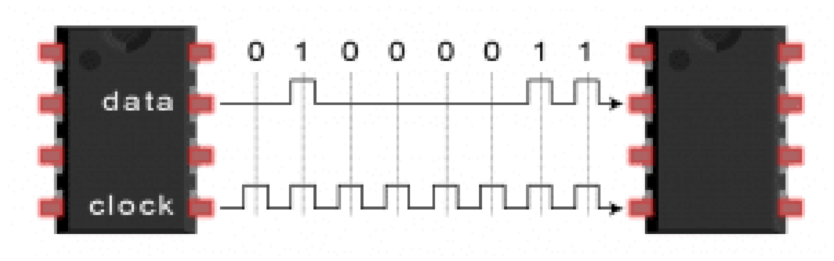
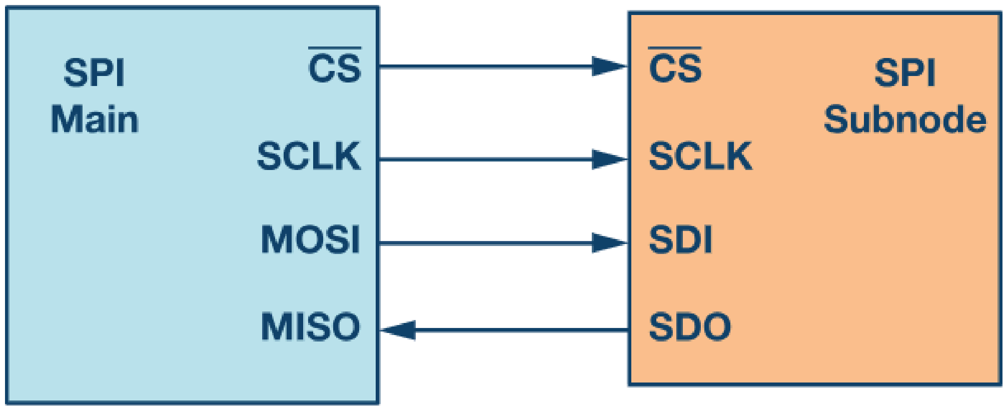
SPI Interfacing (25 Marks)

Serial peripheral interface (SPI) is one of the most widely used interfaces between microcontrollers and peripheral ICs such as sensors, ADCs, DACs, shift registers, SRAM, and others. In SPI, the communication is serial, i.e., the information flows bit by bit.



It consists of two devices - Master & Slave.

SPI devices have 4 signals: MISO (Master In Slave Out), MOSI (Master Out Slave In), SCLK (SPI CLK) and CS/SS (Chip/Slave Select).



In the MISO line, the Master receives a signal from the Slave, while MOSI receives the opposite. SCLK is a reference clock, and CS is used to select the slave. CS is active low.

In this question, you are required to program the master part of the SPI interface.

The start code is follow:

```
module Master(ss, MOSI, done, data_in, clk, rst);
        output reg ss, MOSI, done;
        input data_in, clk, rst;

        …
```

Here, ss has to be high while data is being transferred and low otherwise, MOSI is the output data stream, done is a flag which is 1 when the whole data is transferred. data_in is the input signal stream. When rst is low, ss will be high, and data transfer will begin. When ‘1111’ stream is detected, data transfer ends, and ss becomes low. You can assume all of this happens at posedge of clk.

Conway’s Game of Life   (20 Marks)

Conway’s Game of Life is a two-dimensional cellular automaton.

The "game" is played on a two-dimensional grid of cells, where each cell is either 1 (alive) or 0 (dead). At each time step, each cell changes state depending on how many neighbours it has:

- 0-1 neighbour: Cell becomes 0.
- 2 neighbours: Cell state does not change.
- 3 neighbours: Cell becomes 1.
- 4+ neighbours: Cell becomes 0.

The game is formulated for an infinite grid. In this circuit, we will use a 16x16 grid. To make things more interesting, we will use a 16x16 toroid, where the sides wrap around the other side of the grid. For example, the corner cell (0,0) has 8 neighbours: (15,1), (15,0), (15,15), (0,1), (0,15), (1,1), (1,0), and (1,15). The 16x16 grid is represented by a length 256 vector, where each row of 16 cells is represented by a sub-vector: q[15:0] is row 0, q[31:16] is row 1, etc.

- load: Loads data into q at the next clock edge for loading the initial state.
- q: The 16x16 current state of the game updated every clock cycle.

The game state should advance by one timestep every clock cycle.

The sample code is as follows:

```
module top_module(
    input clk,
    input load,
    input [255:0] data,
    output [255:0] q );
```

Divider (10 Marks)

Design a Verilog module to divide two 8-bit unsigned numbers (dividend and divisor) and provide the quotient (quotient) as output. Additionally, include an output flag (divisor_zero_flag) set high if the divisor is zero. You **cannot** use the built-in division operator (/) in Verilog. Test your Verilog code with the following example input values and expected outputs:

Example Test Cases:

Input: dividend = 100 (decimal), divisor = 10 (decimal)
Expected Output: quotient = 10 (decimal), divisor_zero_flag = 0

Input: dividend = 255 (decimal), divisor = 16 (decimal)
Expected Output: quotient = 15 (decimal), divisor_zero_flag = 0

Input: dividend = 200 (decimal), divisor = 25 (decimal)
Expected Output: quotient = 8 (decimal), divisor_zero_flag = 0

Input: dividend = 75 (decimal), divisor = 8 (decimal)
Expected Output: quotient = 9 (decimal), divisor_zero_flag = 0

Input: dividend = 128 (decimal), divisor = 0 (decimal)
Expected Output: quotient = 0 (decimal), divisor_zero_flag = 1

Implement the Verilog module considering the divisor can be zero and verify its functionality using these example test cases

Start Code:

```
module (input [7:0] dividend, divisor, output [7:0] quotient, remainder, output
divisor_zero_flag);
```

Convolution (15 Marks)

Develop a Verilog module to perform the convolution of two signals (signal_a and signal_b), where each individual element of the signals is limited to 8 bits and the number of elements to 16. Provide the result (convolved_signal) as output. Additionally, include an output flag (invalid_input_flag) set high if input signals are invalid (e.g. if either signal_a or signal_b is all zeros). Test your Verilog code with the following example input values and expected outputs:

Example Test Cases:
1. Input: signal_a = {1, 2, 3, 4, 5, 6, 7, 8} (8 elements, each element is 8 bits), signal_b = {1, 0, 1, 0, 1, 0} (6 elements, each element is 8 bits)
   Expected Output: convolved_signal = {1, 2, 4, 6, 9, 12, 16, 14, 12, 9, 6, 4, 2} (13 elements, each element is 8 bits), invalid_input_flag = 0

2. Input: signal_a = {0, 0, 0, 0, 0, 0, 0, 0} (8 elements, each element is 8 bits), signal_b = {1, 2, 1, 0, 1, 0} (6 elements, each element is 8 bits)
   Expected Output: convolved_signal = {0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0} (13 elements, each element is 8 bits), invalid_input_flag = 1

3. Input: signal_a = {3, 1, 2, 0, 0, 0} (6 elements, each element is 8 bits), signal_b = {0, 0, 0, 0, 0, 0} (6 elements, each element is 8 bits)
   Expected Output: convolved_signal = {0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0} (13 elements, each element is 8 bits), invalid_input_flag = 1

4. Input: signal_a = {2, 3, 4, 5, 6, 7} (6 elements, each element is 8 bits), signal_b = {1, 1, 1, 1, 1, 1, 1, 1} (8 elements, each element is 8 bits)
   Expected Output: convolved_signal = {2, 5, 9, 14, 20, 26, 27, 25, 20, 14, 9, 5, 2} (13 elements, each element is 8 bits), invalid_input_flag = 0

5. Input: signal_a = {4, 2, 3, 1, 0, 0, 0, 0} (8 elements, each element is 8 bits), signal_b = {1, 0, 1, 0, 1, 0, 1, 0} (8 elements, each element is 8 bits)
   Expected Output: convolved_signal = {4, 2, 7, 5, 4, 3, 1, 0, 0, 0, 0, 0, 0} (13 elements, each element is 8 bits), invalid_input_flag = 0

Implement the Verilog module considering the limitation of individual element sizes to 8 bits and the possibility of invalid input signals. Verify its functionality using these example test cases.

Start Code:

```
module(input [7:0] signal_a, signal_b, output [7:0] convolved_signal,
          output  invalid_input_flag)
```

Roots of a Mathematical Function   (30 Marks)

"Write a Verilog module to find the number of roots of the equation of the given function g(x) using Taylor series expansions to approximate the functions tan(x) and e^x. The module should use a binary search algorithm to find the roots and output the number of roots found."

This question outlines the specific requirements for the Verilog module, including the functions to be approximated using Taylor series expansions, the binary search algorithm, and the output format for reporting the number of roots found.

$$g(x) = e^{(\sin x)} - \int_0^x f(t)\,dt$$

$$f(x) = \tan(x+2)$$

<u>NOTE</u>: Only for this question, submit a write-up explaining your approach. If your logic is correct, there are partial marks for logic (50%).