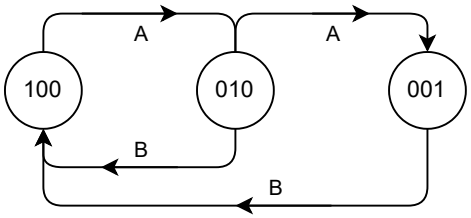
Solutions to Midterm 2

Question 1

A sequence detector detects input sequences consisting of a 1, followed by any number of zeros (including none), followed by a 1.

There are two inputs named **reset** and **in**. There is one output named **out**. When **reset** is asserted the output is set to 0. When the desired sequence is detected, the output is set to 1. The output remains at 1 until **reset** is asserted again.

The state transition diagram and state transition table are shown ~~at right~~ below. The state does not change for other input conditions. A second table shows the value of **out** for each state. Fill in the missing code in the following Verilog module so as to implement the sequence detector state machine described above.



A: in == 1 && reset == 0
B: reset == 1

state	reset	in	next state
X	1	X	100
100	0	1	010
010	0	1	001

state	out
100	0
010	0
001	1

```
module detector
  ( input logic reset, clk, in,
    output logic out ) ;

   logic [2:0] state ;
...
endmodule
```

Solution

Two possible **detector** modules and a testbench are given below. The second solution takes advantage of the one-hot encoding of the states.

```
// simple solution

module detector
  ( input logic reset, clk, in,
    output logic out ) ;

   logic [2:0] state ;

   always_ff @(posedge clk)
     state <= reset == '1 ? 3'b100 :
              state == 3'b100 && in == '1 ? 3'b010 :
              state == 3'b010 && in == '1 ? 3'b001 :
              state ;

   assign out = state == 3'b001 ? '1 : '0 ;

endmodule

// taking advantage of the one-hot encoding

module detector_
  ( input logic reset, clk, in,
    output logic out ) ;

   logic [2:0] state ;

   always_ff @(posedge clk)
     state <= reset ? 3'b100 :
              state[2] && in ? 3'b010 :
              state[1] && in ? 3'b001 :
              state ;

   assign out = state[0] ;

endmodule

module detector_tb ;

   logic reset, clk, in ;
   logic out ;

   detector d0 (.*) ;

   logic [0:15] tv[2]='{ 16'b1000_0010_0000_0000,
                         16'b0001_1000_0100_1000 } ;

   initial begin
      $dumpfile("detector.vcd") ;
      $dumpvars ;

      {reset, clk, in} = 3'b100 ;

      for ( int i=0 ; i<16 ; i++ ) begin
         #1 clk = '0 ;
         {reset,in}={tv[0][i],tv[1][i]} ;
         #1 clk = '1 ;
      end
   end

endmodule
```

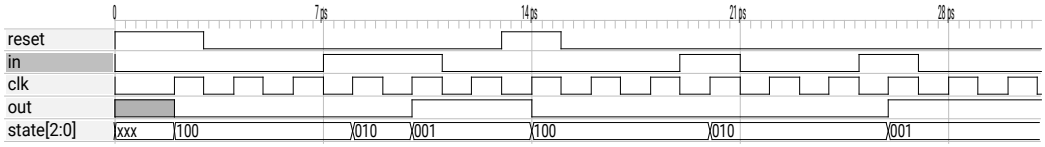


Figure 1: Simulation results for Question 1

The simulation waveforms showing a reset, detection of the sequence **11**, a second reset, and detection of the sequence **1001** is shown in Figure 1:

Question 2

(a) Should an input named $\overline{\texttt{RESET}}$ (or **RESET**) be set high or low to cause a reset?

(b) Should an output named **INTERRUPT** (or $\overline{\texttt{INTERRUPT}}$) be set high or low to cause an interrupt?

Solution

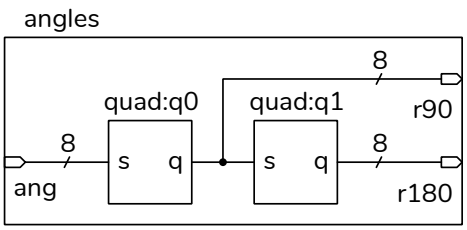
Active-high inputs or outputs such as **RESET** or **INTERRUPT** are true (asserted) when they are high. Thus to cause a reset or interrupt **RESET** or **INTERRUPT** should be high.

Active-low inputs or outputs such as $\overline{\texttt{RESET}}$ or $\overline{\texttt{INTERRUPT}}$ are true (asserted) when they are low. Thus to cause a reset or interrupt $\overline{\texttt{RESET}}$ or $\overline{\texttt{INTERRUPT}}$ should be low.

Question 3

The following Verilog shows the declaration of a module named **quad**. The diagram shows how two of these are connected within a module named **angles**:

```
module quad
  ( input  logic [7:0] s,
    output logic [7:0] q ) ;
   // ...
endmodule
```



The numbers above the lines show the signal (bus) widths. The identifiers above the boxes show the module and instance names. The identifiers below the ports show the **angles** module port names.

Write a System Verilog module named **angles** that implements the diagram above. Declare any signals required to implement the **angles** module. Do not write the **quad** module. Follow the course coding conventions

Solution

Concise and verbose versions of an **angle** module are shown below. Note that the **quad** module does not have parameters and specifying them would have no effect.

```
module quad
  ( input logic [7:0] s,
    output logic [7:0] q ) ;
   // ...
endmodule

// concise solution

module angle_
  ( input logic [7:0] ang,
    output logic [7:0] r90, r180 ) ;

   quad q0 (ang, r90) ;
   quad q1 (r90, r180) ;

endmodule

// verbose solution

module angle
  #(N=8)
  ( input logic [N-1:0] ang,
    output logic [N-1:0] r90, r180 ) ;

   logic [N-1:0] t ;

   quad q0 (.s(ang), .q(t)) ;
   quad q1 (.s(t),   .q(r180)) ;

endmodule
```

Question 4

Fill in the following testbench module with code that does the following:

(a) sets the value of **n** to **0** in an **initial** block,

(b) terminates the simulation with **$stop()** if the value of **n** is equal to **8'd255**,

(c) adds 1 to **n** every 5 µs, and

(d) uses **$display()** to print the value of **n** every 20 µs.

```
module midterm_tb ;

   logic [7:0] n ;
...
endmodule
```

Solution

Various solutions are possible. A **midterm_tb** module meeting the above requirements and written using the Verilog language features described in the lecture notes (**wait**, **$stop**, **$display**, **always** and *#delay*) is shown below.

```
module midterm_tb ;

   logic [7:0] n ;

   initial begin
      n = '0 ;
      wait ( n == 8'd255 ) ;
      $stop() ;
   end

   always  #5us n = n + 1 ;

   always #20us $display(n) ;

endmodule
```

The start and end of the logged (**$display**) output is:

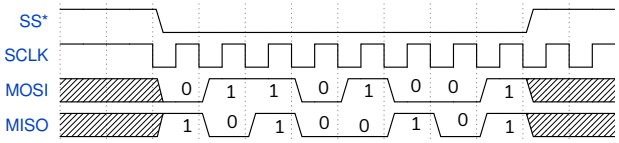
```
# run -all
#   3
#   7
#  11
...

# 243
# 247
# 251
# ** Note: $stop    : midterm2sol-3.sv(12)
#    Time: 1275 us  Iteration: 1  Instance: /midterm_tb
# Break in Module midterm_tb at midterm2sol-3.sv line 1
```

Question 5

The following waveform shows the signals on an SPI interface. What value was transmitted from the master to the slave (or slave to the master) assuming the bits were transferred most-significant-bit first? Give you answer as a hexadecimal number. Show your work.

Solution



Data is transferred on the rising edge of SCLK while SS* is asserted (low). The values of the bits on MOSI and MISO are shown in the diagram below.

From master to slave on MOSI the bits transferred were: 8'b0110_1001 = 8'h69.

From slave to master on MISO the bits transferred were: 8'b1010_0101 = 8'ha5.