1. (20 pts) Design a parallel to serial data converter (16-bit to 8-bit) using the waveform below. Write down the Verilog code for your design.

```verilog
module par2serial (rst, clk, datain, dataout);
  input rst, clk;
  input [15:0] datain;
  output wire [7:0] dataout;
  reg toggle;
  always @(posedge clk)
    toggle<=~toggle;
  assign dataout =(toggle)?datain[15:8]: datain[7:0];
endmodule
```
2. (20 pts) Using the example BlockRAM (BRAM) simulation below, write a Verilog module such that
   i. a synchronous-read-BRAM of 8-bit address and 16-bit data width (4KBit) is inferred. Note that if EN is low, BRAM is disabled if EN is high and WE is low, the content of the BRAM location at ADDR is read out as DOUT. If EN is high and WE is high, DIN is written at ADDR of BRAM.
   ii. Then create a top module. Instantiate the BRAM in your top module. Write controller logic to initialize the content as 16'hAAAA for each BRAM address (from 8'h00 to 8'hFF). Then read out all the BRAM contents (from 8'h00 to 8'hFF). You can use for loop.

   PART i:

   module xilinx_one_port_ram_sync
     #(
       parameter ADDR_WIDTH = 8,
       DATA_WIDTH = 16
     )
     (input wire clk,
      input wire we,
      input wire en,
      input wire [ADDR_WIDTH-1:0] addr,
      input wire [DATA_WIDTH-1:0] din,
      output wire [DATA_WIDTH-1:0] dout
    );

    // signal declaration
    reg [DATA_WIDTH-1:0] ram [2**ADDR_WIDTH-1:0];
    reg [ADDR_WIDTH-1:0] addr_reg:
// body
always @(posedge clk)
begin
    if(en)
        if (we) // write operation
            ram[addr] <= din;
            addr_reg <= addr;
    // read operation
    assign dout = ram[addr_reg];
endmodule

PART ii:
THIS IS A TRICKY QUESTION! IT IS DANGEROUS TO USE “FOR LOOP” IN AN ALWAYS BLOCK!!!
USE A COUNTER INSTEAD OF A FOR LOOP!!! YOU CAN USE AN INITIAL BLOCK TO INIT THE BRAM (NOTE THAT XILINX XST IS OK WITH THIS, BUT INITIAL AND FOR LOOP ARE NOT SYNTHESIZABLE!)
BELOW CODE IS SYNTHESIZABLE, IMPLEMENTED WITH A COUNTER.

module top (clk, rst, dout);
    input clk;
    input rst;
    output wire [15:0] dout;
    reg en, we;
    reg [7:0] i; // Counter
    reg[7:0] addr;
    reg [15:0] din;
    reg[1:0] state=2'b00;

    xilinx_one_port_ram_sync #( .ADDR_WIDTH(8), .DATA_WIDTH(16)) myram (clk, en, we, addr, din, dout);

    always @(posedge clk) begin
        if (rst) begin
            state<= 2'b00;
            en <= 1'b0; we<=1'bX; addr<=8'hXX; din <= 16'hXXXX;
            i <=8'h00;
        end
        else begin
            case (state)
                2'b00: begin
                    en <= 1'b1; we<=1'b1; din <= 16'hAAAA; state <=2'b00;
                    addr<=i;
                end
            endcase
        end
    end
endmodule
i<=i+1;
if (i == 8'hFF) begin
    en <= 1'b1; we<=1'b1; din <= 16'hAAAA;
    addr<=8'hFF; state <=2'b01;
    i<=8'h00;
end

2'b01: begin
    en <= 1'b1; we<=1'b0; din <= 16'hXXXX; state <=2'b01; addr<=i;
    i<=i+1;
    if (i== 8'hFF) begin
        en <= 1'b1; we<=1'b0; din <= 16'hXXXX;
        addr<=i; state <=2'b10;
    end
end

2'b10: begin
    en <= 1'b0; we<=1'bX; addr<=8'hXX; din <= 16'hXXXX; state <=2'b10;
end

default: begin
    en <= 1'b0; we<=1'bX; addr<=8'hXX; din <= 16'hXXXX; state <=2'b10;
end
endcase
end
endmodule
3. (10 pts) Write a Verilog module for a single-buffer bidirectional I/O port.

```
module exam(Data_out, dir, Data_in, bi);
    inout bi;
    input Data_out, dir;
    output Data_in;

    assign bi = (dir) ? Data_out : 1'hZ;
    assign Data_in = bi;
endmodule
```
4. (20 pts) Design a fall-rise edge detector circuit using only NOR2 gates and a flip-flop. Then write the Verilog code for it.

RTL-level the code looks like below.

```verbatim
module rise-fall(clk, in, detect);
  input clk, in;
  output detect;
  reg in_d;
  always@(posedge clk)
    in_d<=in;
  assign detect= in_d^in;
endmodule
```

To implement an XOR gate with NOR2 gates, first write the equation. Verilog supports basic logic Gates as predefined primitives.

With nor Gates the code is as follows:

```verbatim
module rise-fall(clk, in, detect);
  input clk, in;
  output detect;
  reg in_d;
  always@(posedge clk)
    in_d<=in;
  nor i0 (n1, in_d, in_d); // 2-input nor becomes an inverter.
  nor i1 (n2, n1, in);
  nor i2 (n3, in, in); // 2-input nor becomes an inverter.
  nor i3 (n4, n3, in_d);
  nor i4 (n5, n2, n4);
  nor i5 (detect, n5, n5);
endmodule
```
5. (30 pts) We are supposed to sum up 10 numbers. An adder takes 10ps to sum 2 numbers. (Neglect flop and mux delays.)

a. If we pipeline the computation so that every cycle a new computation can be started, how many adders do we need? Draw the DFG. Come up with a schedule.

b. Based on your answer in part a, design the circuit and write down the Verilog code.

c. Suppose you are given only 3 adders. Now do both pipelining and resource sharing. Come up with a schedule such that the throughput is 3 cycles and the latency is 5 cycles. Assume cycle time is 10ps.

a. Since there are 9 additions then we need 9 adders to compute sum of 10 numbers every cycle.

<table>
<thead>
<tr>
<th>A0</th>
<th>A1</th>
<th>A2</th>
<th>A3</th>
<th>A4</th>
<th>A5</th>
<th>A6</th>
<th>A7</th>
<th>A8</th>
<th>Cycle</th>
</tr>
</thead>
<tbody>
<tr>
<td>S0_1</td>
<td>S1_1</td>
<td>S2_1</td>
<td>S3_1</td>
<td>S4_1</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>1</td>
</tr>
<tr>
<td>S0_2</td>
<td>S1_2</td>
<td>S2_2</td>
<td>S3_2</td>
<td>S4_2</td>
<td>S5_1</td>
<td>S6_1</td>
<td>-</td>
<td>-</td>
<td>2</td>
</tr>
<tr>
<td>S0_3</td>
<td>S1_3</td>
<td>S2_3</td>
<td>S3_3</td>
<td>S4_3</td>
<td>S5_2</td>
<td>S6_2</td>
<td>S7_1</td>
<td>-</td>
<td>3</td>
</tr>
<tr>
<td>S0_4</td>
<td>S1_4</td>
<td>S2_4</td>
<td>S3_4</td>
<td>S4_4</td>
<td>S5_3</td>
<td>S6_3</td>
<td>S7_2</td>
<td>S8_1</td>
<td>4</td>
</tr>
<tr>
<td>S0_5</td>
<td>S1_5</td>
<td>S2_5</td>
<td>S3_5</td>
<td>S4_5</td>
<td>S5_4</td>
<td>S6_4</td>
<td>S7_3</td>
<td>S8_2</td>
<td>5</td>
</tr>
<tr>
<td>S0_6</td>
<td>S1_6</td>
<td>S2_6</td>
<td>S3_6</td>
<td>S4_6</td>
<td>S5_5</td>
<td>S6_5</td>
<td>S7_4</td>
<td>S8_3</td>
<td>6</td>
</tr>
</tbody>
</table>

Latency = 4 cycles Thruput= 1 cycle
module tensum(a,b,c,d,e,f,g,h,i,j, sum, clk);
    input [9:0] a,b,c,d,e,f,g,h,i,j;
    input clk;
    output [9:0] sum;
    always @(posedge clk) begin
        S0<=a +b;
        S1<=c+d;
        S2<=e+f;
        S3<=g+h;
        S4<=i+j;
        S5<=S0+S1;
        S6<=S2+S3;
        S7<=S5+S6;
        S8<=S7+S4;
    end
endmodule

c. Resource sharing & pipelining: only 3 adders and 3 cycle throughput, 5 cycle latency.