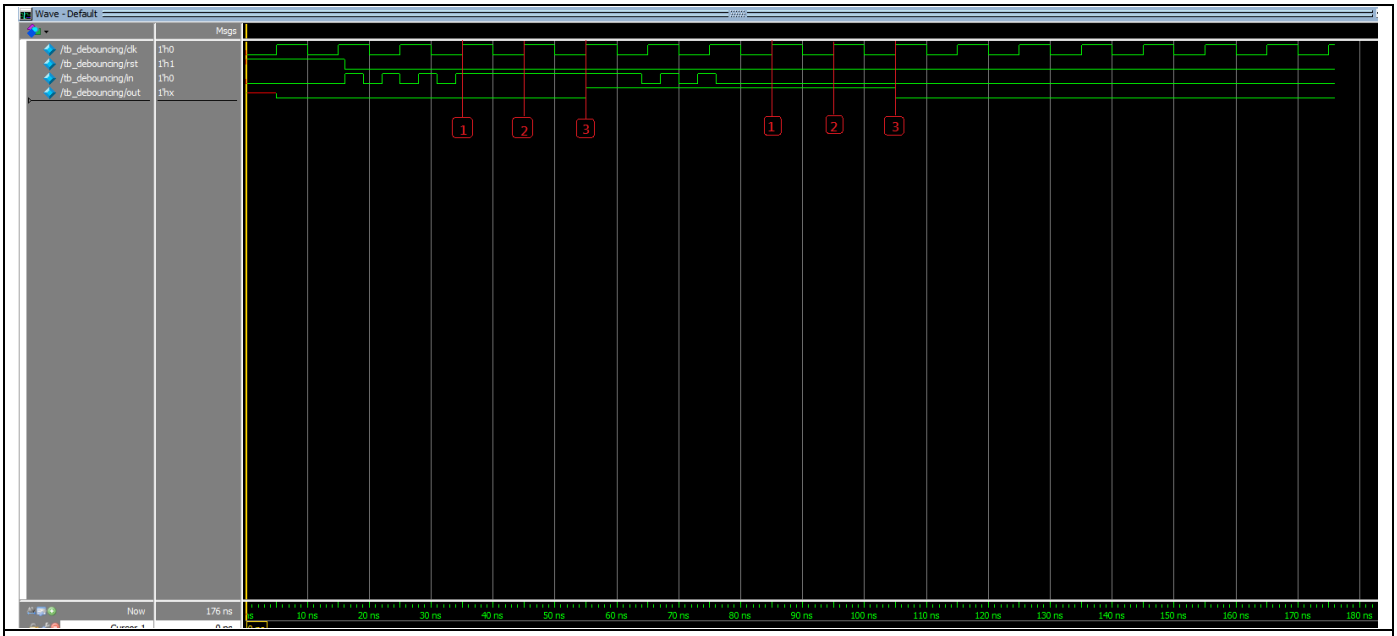
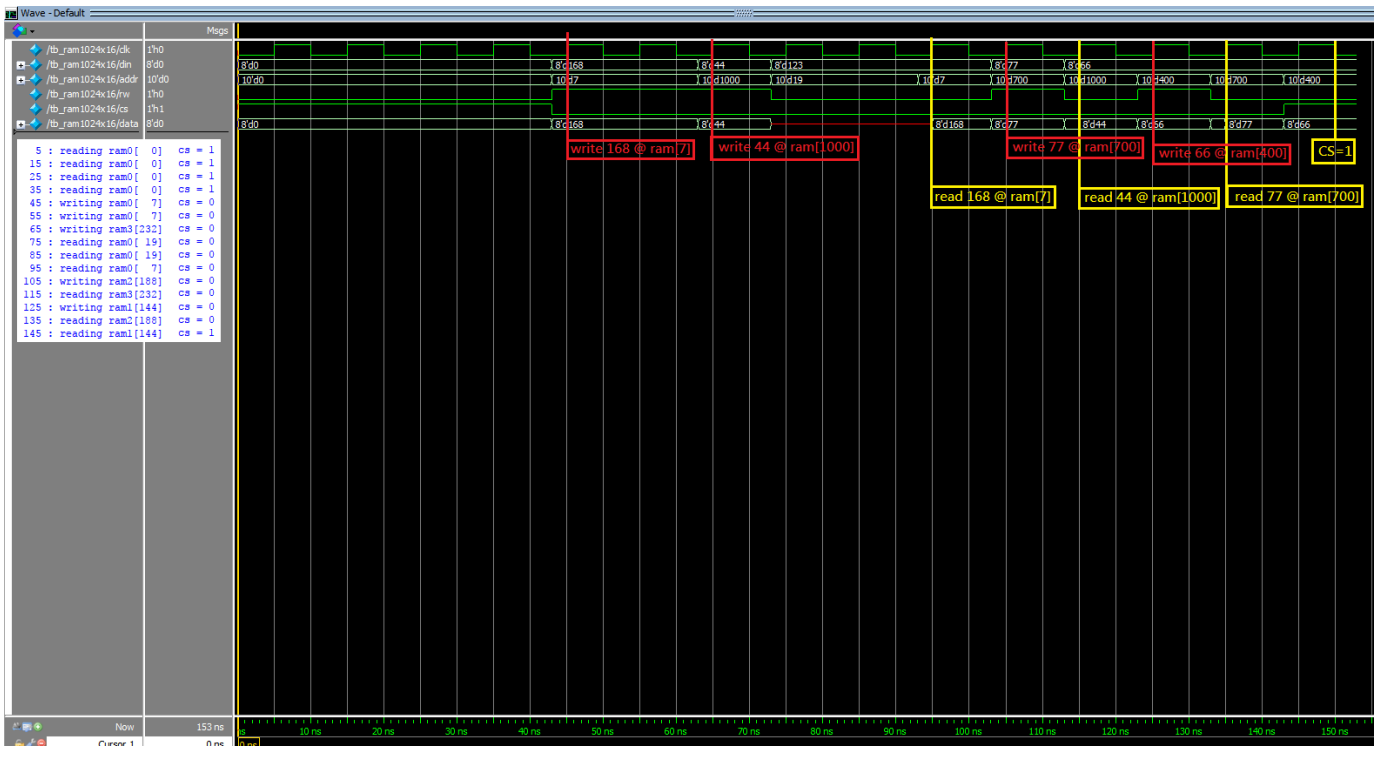
# Q1




```
module debouncing(clk, rst, in, out);
input clk, rst, in;
output reg out;

reg [2:0] state, next_state;
parameter WAIT = 3'b000,
          h1   = 3'b001,
          h2   = 3'b010,
          h3   = 3'b011,
          l1   = 3'b100,
          l2   = 3'b101;

always@(posedge clk) begin
    if(rst) state <= WAIT;
    else    state <= next_state;
end

always@(state or in) begin //excitation logic
    case(state)
        WAIT: next_state = (in) ? h1 : WAIT;
        h1  : next_state = (in) ? h2 : WAIT;
        h2  : next_state = (in) ? h3 : WAIT;
        h3  : next_state = (in) ? h3 : l1  ;
        l1  : next_state = (in) ? h3 : l2  ;
        l2  : next_state = (in) ? h3 : WAIT;
```

```
        default: next_state = WAIT;
    endcase
end

always@(state) begin //output logic
    case(state)
        WAIT: out = 1'b0;
        h1  : out = 1'b0;
        h2  : out = 1'b0;
        h3  : out = 1'b1;
        l1  : out = 1'b1;
        l2  : out = 1'b1;
        default: out = 1'b0;
    endcase
end

endmodule
```

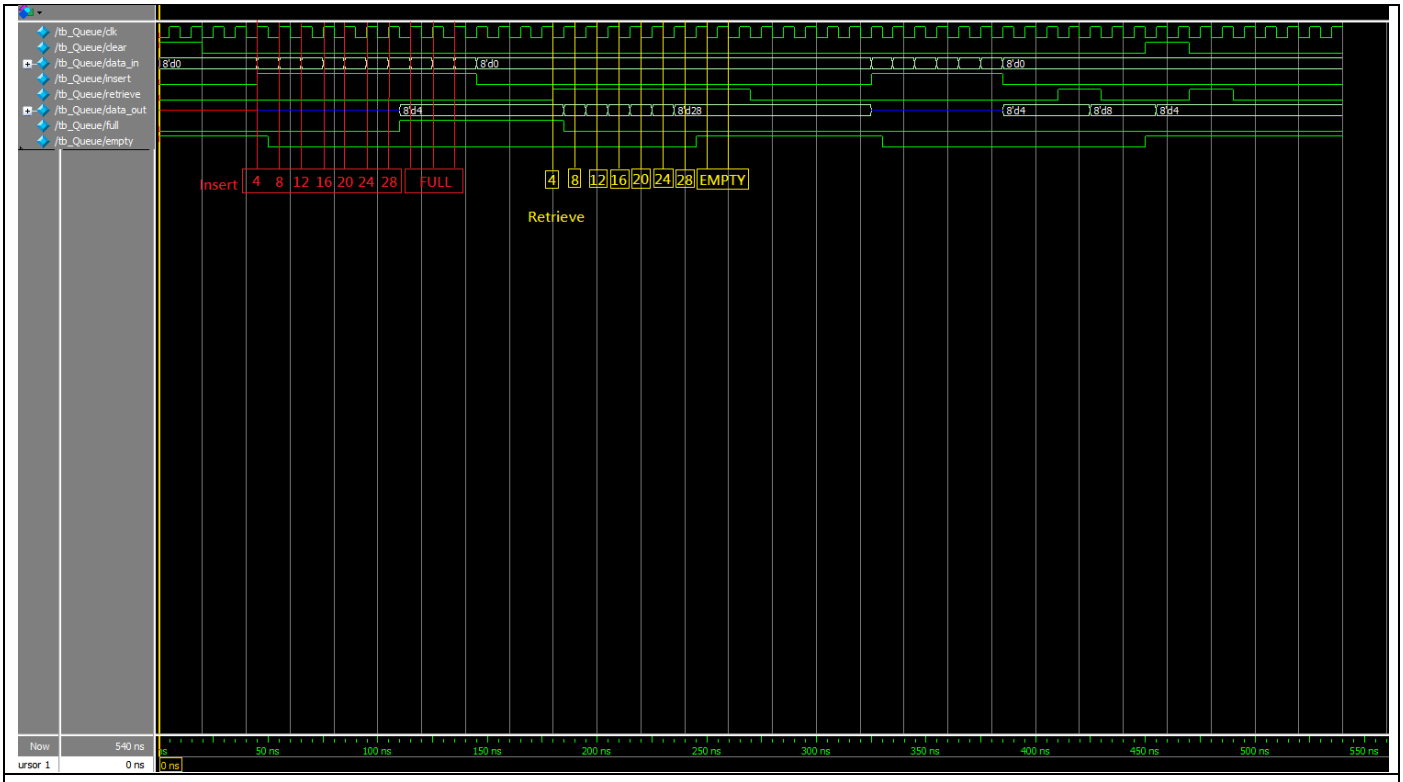
# Q2




```
module ram1024X16(clk, addr, data, rw, cs);
input clk, rw, cs;
input [9:0] addr;
```

```
inout [15:0] data;

reg [15:0] ram0 [0:1023];
reg [15:0] ram1 [0:1023];
reg [15:0] ram2 [0:1023];
reg [15:0] ram3 [0:1023];
reg [15:0] dout;

//dataflow
assign data = (!rw)&&(!cs) ? dout : 16'bzzzz_zzzz_zzzz_zzzz;

// read
always@(posedge clk) begin
    // read
    if(!rw) begin
        case(addr[9:8])
            2'b00 : dout <= ram0[addr[7:0]];
            2'b01 : dout <= ram1[addr[7:0]];
            2'b10 : dout <= ram2[addr[7:0]];
            2'b11 : dout <= ram3[addr[7:0]];
            default : $display("error");
        endcase
        $display("%t : reading ram%d[%d] = %d, cs = %d", $time, addr[9:8], addr[7:0],
dout, cs);
    end
    //write
    else begin
        case(addr[9:8])
            2'b00 : ram0[addr[7:0]] <= data;
            2'b01 : ram1[addr[7:0]] <= data;
            2'b10 : ram2[addr[7:0]] <= data;
            2'b11 : ram3[addr[7:0]] <= data;
            default : $display("error");
        endcase
        $display("%t : writing ram%d[%d] <= %d, cs = %d", $time,  addr[9:8],
addr[7:0], data, cs);
    end
end
endmodule
```

# Q3




```
/*3bit counter head pointer*/
module head_ptr(
    input clk,
    input en,
    output reg [2:0] Q,
    input rst
);
always@(posedge clk) begin
    if(rst) Q <= 3'b001;
    else begin
        if(en) Q <= Q + 1;
        else Q <= Q; //也可不 assign
    end
end
endmodule

/*3bit counter tail pointer*/
module tail_ptr(
    input clk,
    input en,
    output reg [2:0] Q,
    input rst
);
```

```
always@(negedge clk) begin
    if(rst) Q <= 3'b001;
    else begin
        if(en) Q <= Q + 1;
        else Q <= Q; //也可不 assign
    end
end
endmodule

/*reg for full & empty flag*/
module Flags(
    input [2:0] head,
    input [2:0] tail,
    output reg full,
    output reg empty,
    input clear
);
always@(clear, head, tail) begin
    if(clear) {full, empty} = 2'b01;
    else begin
        if(head == tail)            {full, empty} = 2'b01;
        else if((tail + 1) == head) {full, empty} = 2'b10;
        else                        {full, empty} = 2'b00;
    end
end
endmodule

/*Circular buffer for Queue storage*/
module Circular_buffer(
    input [7:0] data_in,
    output reg [7:0] data_out,
    input [2:0] addr, //for head or tail
    input rw //rw = 0 (read); rw = 1 (write)
);
reg [7:0] mem [0:7];
always@(rw, addr) begin
    if(rw) begin //rw = 1 write
        mem[addr] = data_in;
        data_out = 8'bzzzz_zzzz;
    end
    else begin //rw = 0 read
        data_out = mem[addr];
```

```
    end
end
endmodule

/*Insert_FSM*/
module Insert_FSM(
    input clk,
    input ins,
    input full,
    input clear,
    output reg tail_en,
    output reg rw
);
reg state, next_state;
parameter A = 1'b0,
          B = 1'b1;
always@(posedge clk) begin
    if(clear) state <= 1'b0;
    else state <= next_state;
end
always@(ins, full, state) begin
    case(state)
        A : next_state = ({ins, full} == 2'b10) ? B : A;
        B : next_state = ({ins, full} == 2'b10) ? B : A;
    endcase
end
always@(ins, full, state) begin
    case(state)
        A : {rw, tail_en} = ({ins, full} == 2'b10) ? 2'b11 : 2'b00;
        B : {rw, tail_en} = ({ins, full} == 2'b10) ? 2'b11 : 2'b00;
    endcase
end
endmodule

/*Retrive_FSM*/
module Retrive_FSM(
    input clk,
    input ret,
    input empty,
    input clear,
    output reg head_en,
    output reg rw
```

```
);
reg state, next_state;
parameter C = 1'b0,
          D = 1'b1;
always@(negedge clk) begin
    if(clear) state <= 1'b0;
    else state <= next_state;
end
always@(ret, empty, state) begin
    case(state)
        C : next_state = ({ret, empty} == 2'b10) ? D : C;
        D : next_state = ({ret, empty} == 2'b10) ? D : C;
    endcase
end
always@(ret, empty, state) begin
    case(state)
        C : {rw, head_en} = ({ret, empty} == 2'b10) ? 2'b01 : 2'b00;
        D : {rw, head_en} = ({ret, empty} == 2'b10) ? 2'b01 : 2'b00;
    endcase
end
endmodule

module Queue(
    input clk,
    input clear,
    input [7:0] data_in,
    input insert,
    input retrieve,
    output [7:0] data_out,
    output full,
    output empty
);
wire h_en, t_en;
wire [2:0] h_addr, t_addr, in_addr;
wire in_full, in_empty;
wire Ins_rw, Ret_rw, in_rw;

head_ptr head_ptr(clk, h_en, h_addr, clear);
tail_ptr tail_ptr(clk, t_en, t_addr, clear);
Flags Flags(h_addr, t_addr, in_full, in_empty, clear);
Circular_buffer Circular_buffer(data_in, data_out, in_addr, in_rw);
Insert_FSM Insert_FSM(clk, insert, in_full, clear, t_en, Ins_rw);
```

```
Retrive_FSM Retrive_FSM(clk, retrieve, in_empty, clear, h_en, Ret_rw);

assign in_addr = (in_rw) ? t_addr : h_addr;
assign in_rw = Ins_rw | Ret_rw;
assign full = in_full;
assign empty = in_empty;

endmodule
```