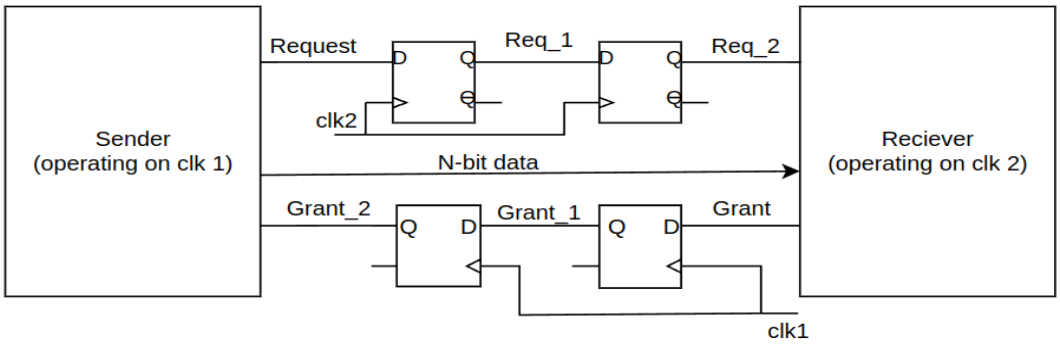
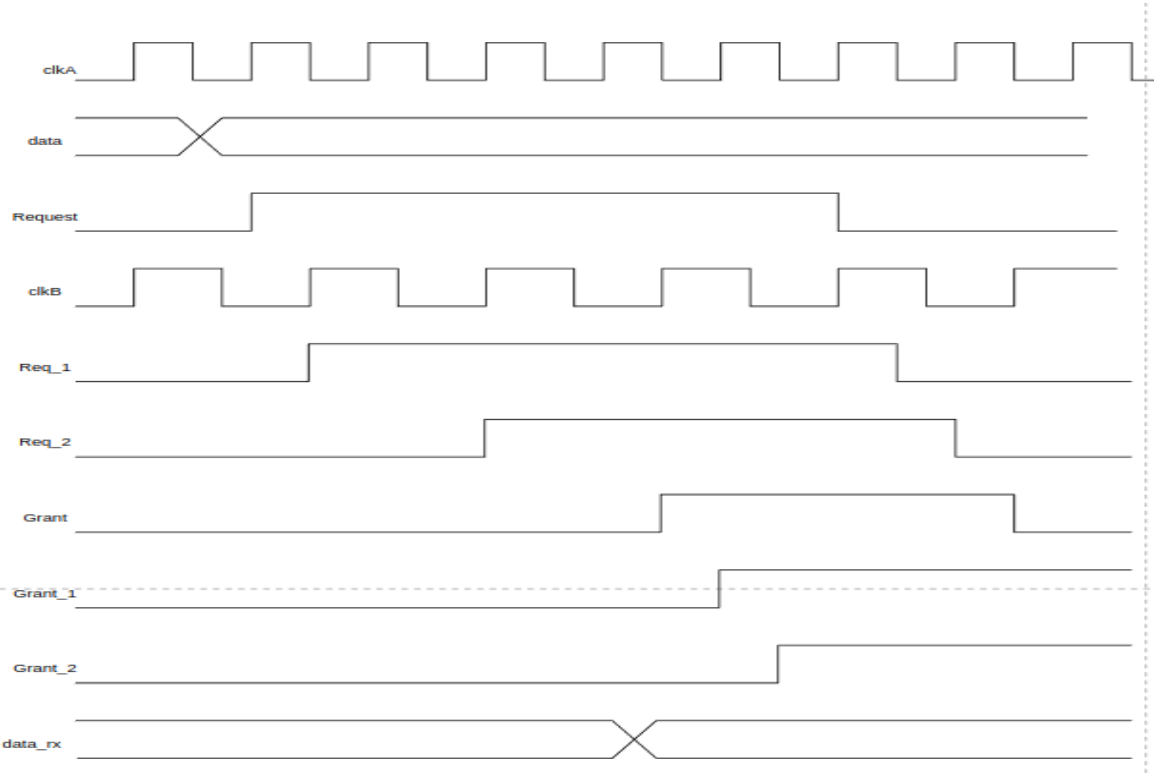
Assignment 5: FIFO

Question # 1

Given a request signal in the clock domain A and a grant signal in the clock domain B, how to process the handshaking between request-grant with the request crossing from domain A to domain B? Similarly, how about signal grant from domain B to domain A?



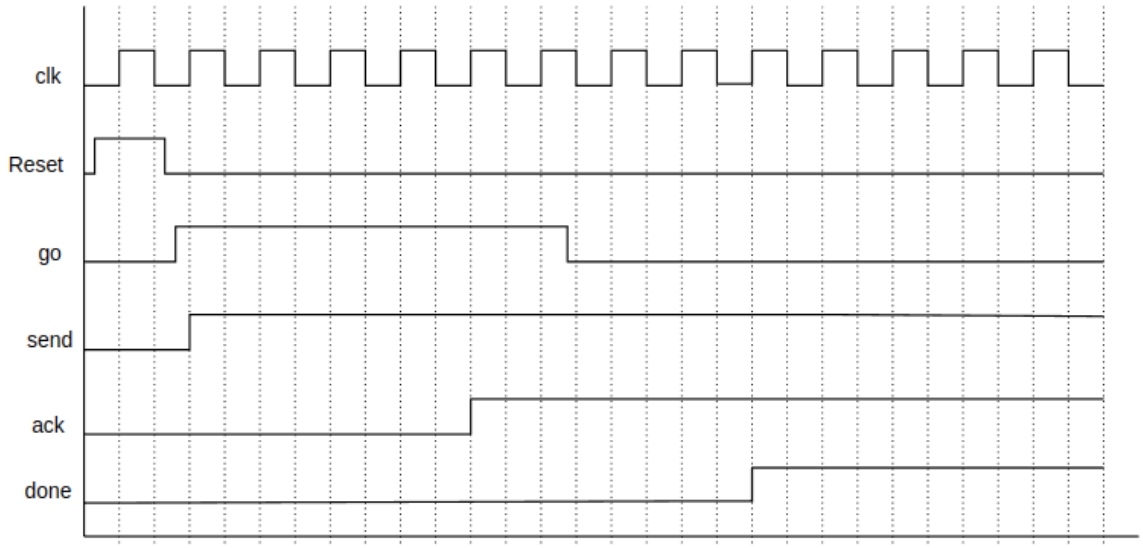
The block diagram for this scenario is shown in the figure above.



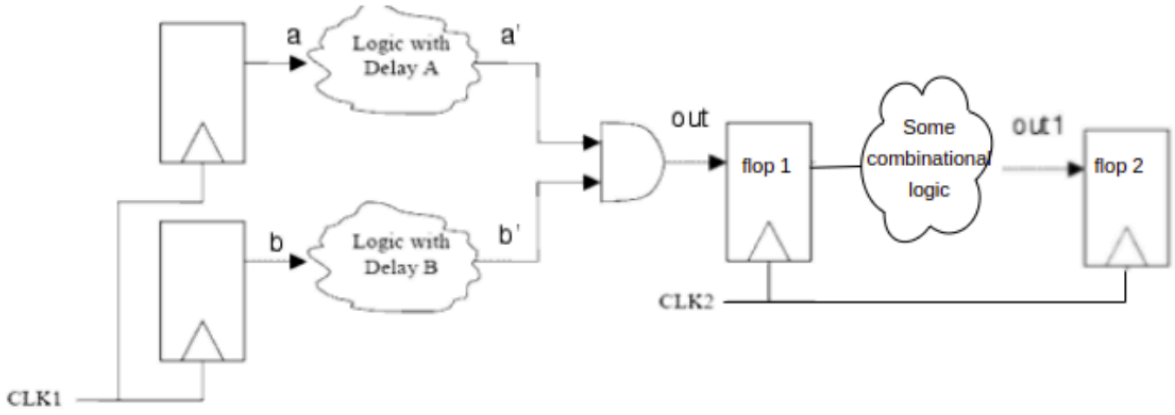
The timing diagram for the above provided block diagram is shown.

Similarly for Domain B to domain A , domain B will generate the request signal and domain A will generate the grant signal. The protocol followed will be the same.

Question # 2
Following shows the implementation of the handshake synchronizer. The dotted line shows the clock domain crossing. Complete the timing diagram



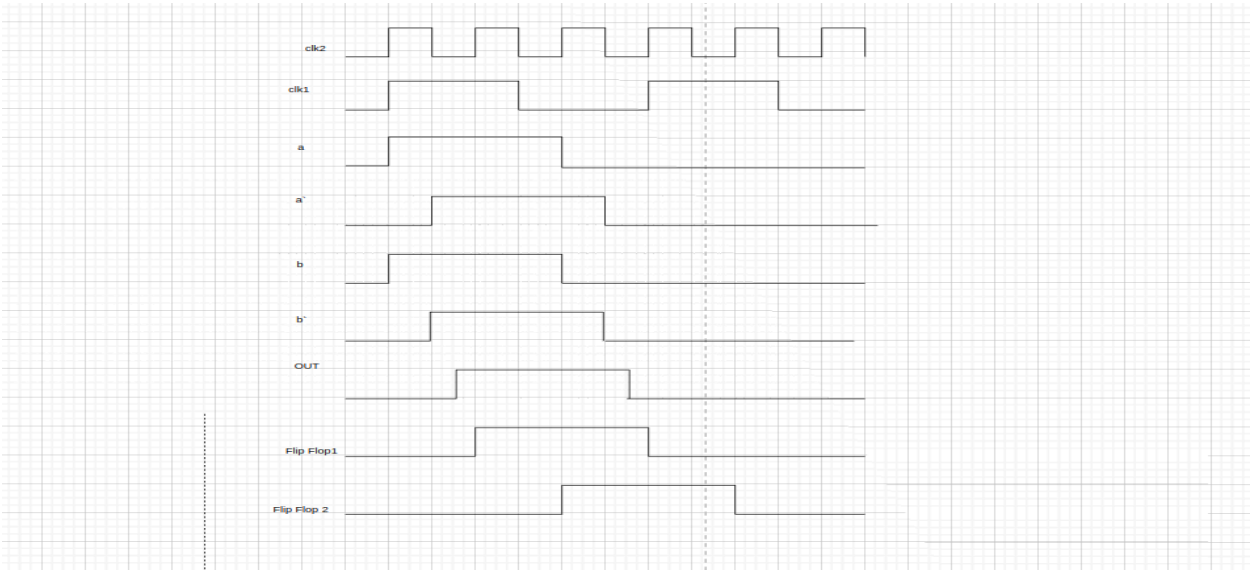
Question # 3
Following shows a two level synchronizer



Flop 1 and flop 2 are used for synchronization of CDC from clk 1 to clk 2. What is the issue with this design? Correct it and draw a waveform. Keep propagation delays in mind.

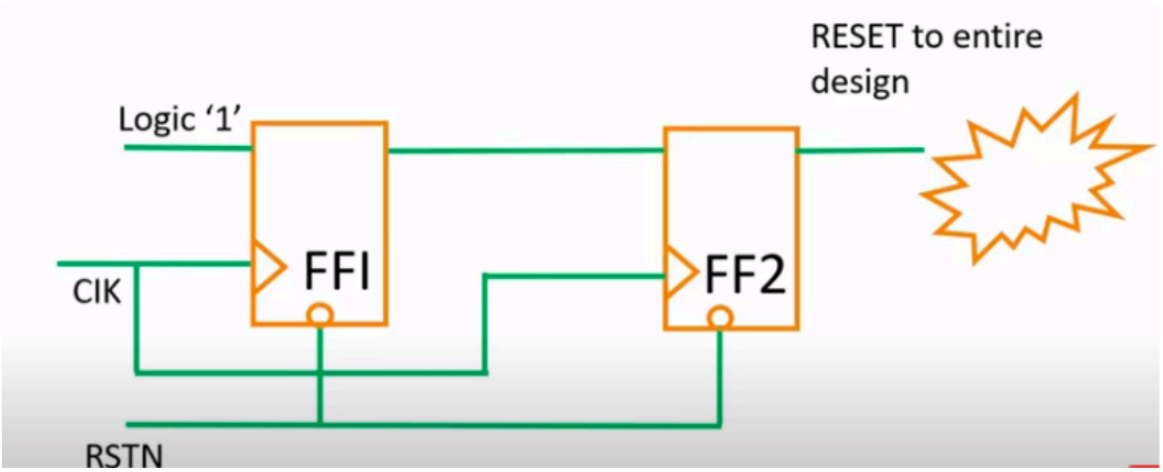
We should remove the combinational circuit between flop 1 and flop 2 by moving it before flop 1. The reason is that in case the output of the and gate changes somewhere between the setup and hold time of flop1, it will cause metastability and metastability will take some time to settle to a stable value. Now in this case the combinational circuit will increase the delay and hence wrong output might be propagated to the output of the circuit.

The screenshot of the corrected circuit will be somewhat like this.



Question # 4

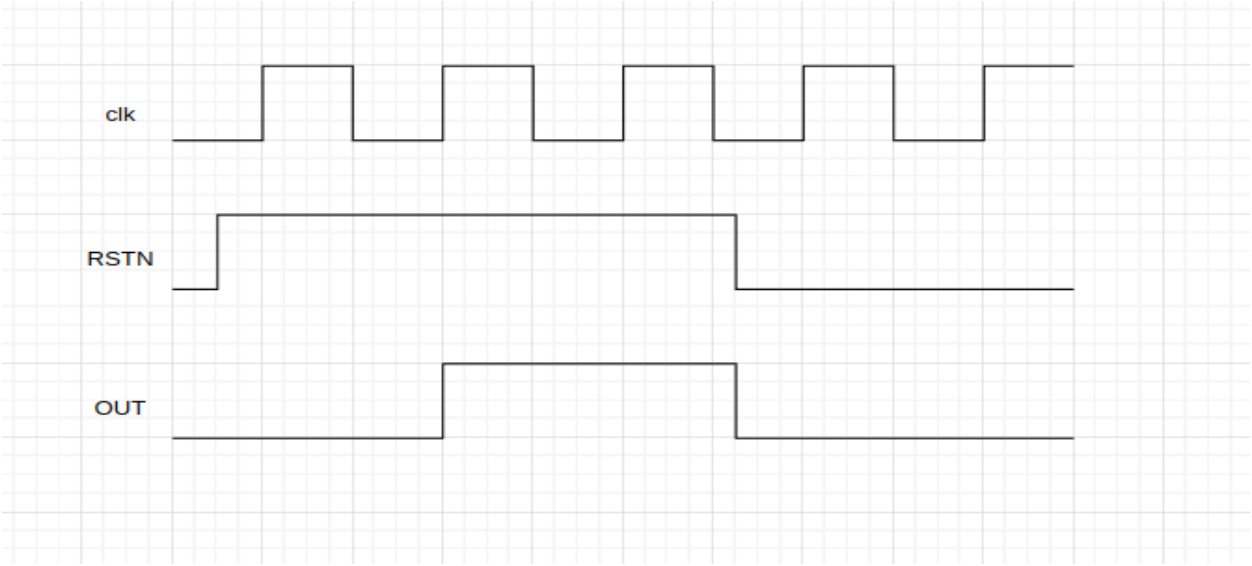
Why is the data synchronizer not used for reset synchronization?
Problem description: The most commonly used reset synchronizer is



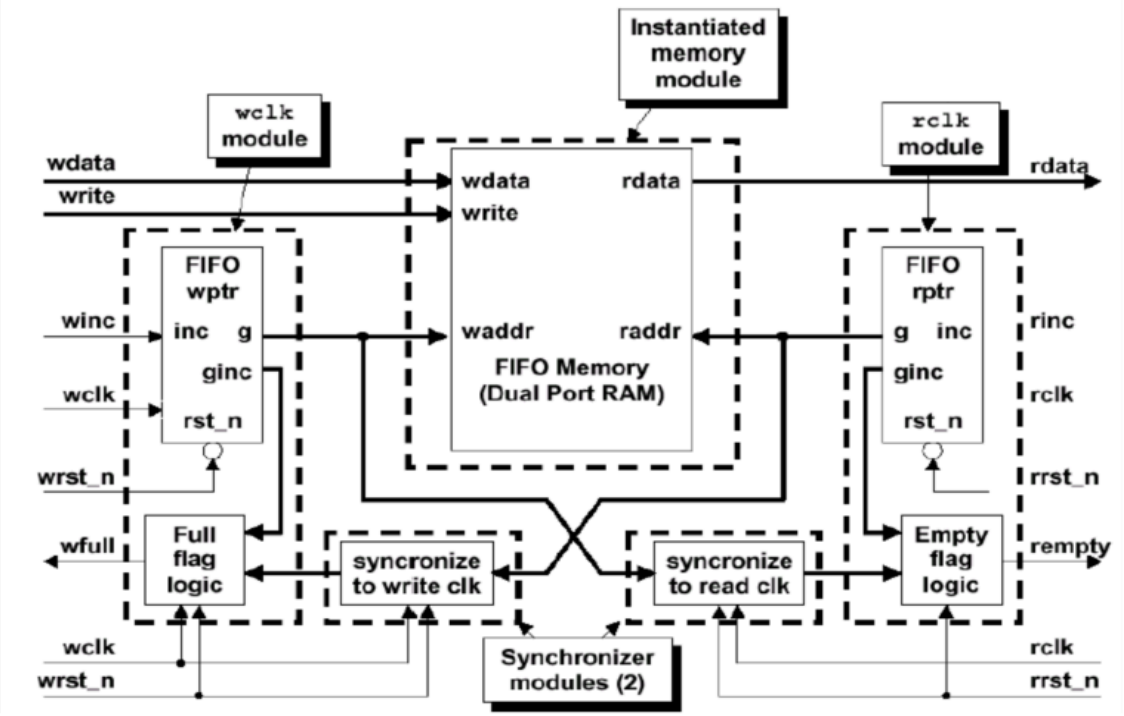
The reset signal is applied at the reset of both flip flops instead at the input of the first flop. And the output of the second flop is used as a reset signal for the design . Your task is to draw the waveform of the above circuit. Why is this configuration used in case of reset signals?

Data synchronizers are not used for reset synchronization because sometimes we need to have an asynchronous reset that should propagate to the design as soon as it is asserted and the reset is de-asserted synchronously.

This configuration is used in reset signals because it can assert reset asynchronously and de-assert it synchronously which is the requirement in some cases. If we use a data synchronizer we cannot have an asynchronous reset.



Question#5
Design an asynchronous FIFO in system verilog as shown in the diagram.



Sol:

Open FIFO_asyn.sv ~/Desktop/Resources/System Verilog

```
/*////////////////////////////////////////////////////////////////////////////////////////////////

                                        Memory Module

////////////////////////////////////////////////////////////////////////////////////////////////*/

module SRAM #(parameter data_bits=8,
              parameter address_bits=4
             )
  (
    input logic wclken,                     // write enable signal
    input logic wclk,                       // write clock
    input logic wfull,                      // status signal to indicate full memory
    input logic [data_bits-1:0] wdata,      // data to be written
    input logic [address_bits-1:0] waddr,   // write address
    output logic [data_bits-1:0] rdata,     // read data
    input logic [address_bits-1:0] raddr    // read address
  );

  logic [data_bits-1:0] memory [2**address_bits-1:0];        // memory

  always_ff@(posedge wclk)
    begin
      if(wclken & !wfull)
        memory[waddr]<=wdata;
    end

  assign rdata=memory[raddr];

endmodule

```

Open

FIFO_asyn.sv
~/Desktop/Resources/System Verilog

```
/*////////////////////////////////////////////////////////////////////////////////////////////////////

                                    FIFO EMPTY MODULE

////////////////////////////////////////////////////////////////////////////////////////////////////*/


module FIFO_empty #(parameter data_bits=8,
                    parameter address_bits=4
                   )
  (
    input logic rclk,                                   // read clock
    input logic rinc,                                   // read enable signal
    input logic rrst_n,                                 // read counter reset
    input logic [address_bits:0] rq2_wptr,              // gray code pointer(write) passed from FIFO full module
    output logic rempty,                                // memory is empty
    output logic [address_bits-1:0] raddr,              // read address
    output logic [address_bits:0] rptr                  // gray code pointer to be passed to FIFO full module
  );
  logic carry,carry_next;
  logic [address_bits-1:0] raddr_next;
  logic [address_bits:0] rptr_next;
  always_comb
    begin
      if(!rrst_n)
      rempty=1'b1;                          // always block for empty signal
      else
      rempty=(rq2_wptr==rptr);
    end

  always_ff@(posedge rclk,negedge rrst_n)      // always block for raddr and rptr
    begin
      if(!rrst_n)
      {raddr,rptr,carry}<=0;
      else
      {raddr,rptr,carry}<={raddr_next,rptr_next,carry_next};
    end

  always_comb
    begin                                   // always comb for changing read address
      if(rinc & !rempty)
       begin
         {carry_next,raddr_next}={carry,raddr}+1;
         rptr_next = ({carry, raddr} >> 1) ^ {carry, raddr};
       end
      else
        begin
          {carry_next,raddr_next}={carry,raddr};
          rptr_next=rptr;
        end

    end
```

Open

FIFO_asyn.sv
~/Desktop/Resources/System Verilog

```
/*////////////////////////////////////////////////////////////////////////////////////////////////////

                                    FIFO_FULL MODULE

////////////////////////////////////////////////////////////////////////////////////////////////////*/

module FIFO_FULL #(parameter data_bits=8,
                   parameter address_bits=4
                   )
  (
    input logic wclk,                               // write clock
    input logic winc,                               // write enable
    input logic wrst_n,                             // write counter reset
    input logic [address_bits:0] wq2_rptr,          //read pointer passed from FIFO empty module
    output logic wfull,                             // memory full
    output logic [address_bits-1:0] waddr,          // write address
    output logic [address_bits:0] wptr              // write pointer
  );
  logic carryf;
  logic carryf_next;
  logic [address_bits-1:0] waddr_next;
  logic [address_bits:0] wptr_next;

  always_comb
    begin
      if(!wrst_n)
      wfull=1'b0;                           // always block for wfull signal
      else
        wfull=(wq2_rptr==(wptr ^ 5'b11000));
    end

  always_ff@(posedge wclk, negedge wrst_n)
    begin
      if(!wrst_n)
      {carryf,waddr,wptr}<=0;                                       // always block for write address and write pointer
      else
      {carryf,waddr,wptr}<={carryf_next,waddr_next,wptr_next};
    end

  always_comb
    begin
      if(winc & !wfull)
      {carryf_next,waddr_next}={carryf,waddr}+1;
      else
        begin
      {carryf_next,waddr_next}={carryf,waddr};
        end

      wptr_next = ({carryf_next, waddr_next} >> 1) ^ {carryf_next, waddr_next};

    end

endmodule
```

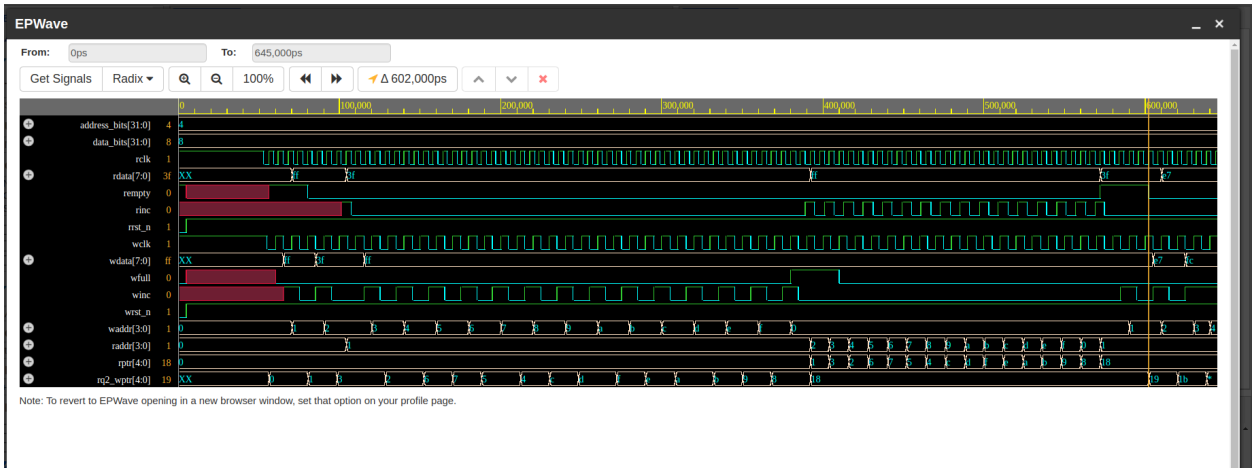
Open

FIFO_asyn.sv
~/Desktop/Resources/System Verilog

```
/*////////////////////////////////////////////////////////////////////////////////////////////////

 2 FLOP Synchronizer Modules for passing read and write pointers between modules
 to generate full and empty signals.

////////////////////////////////////////////////////////////////////////////////////////////////*/


module r2wsynchronize #(parameter data_bits=8,
                  parameter address_bits=4
                  )
 (
   input logic [address_bits:0] rptr,
   input logic wclk,
   input logic wrst_n,
   output logic [address_bits:0] wq2_rptr
 );
 logic [address_bits:0] r2wintermediate;
 always_ff@(posedge wclk,wrst_n)
   begin
     r2wintermediate<=rptr;
     wq2_rptr<=r2wintermediate;
   end
 endmodule


 module w2rsynchronize #(parameter data_bits=8,
                  parameter address_bits=4
                  )
 (
   input logic [address_bits:0] wptr,
   input logic rclk,
   input logic rrst_n,
   output logic [address_bits:0] rq2_wptr
 );
   logic [address_bits:0] w2rintermediate;
   always_ff@(posedge rclk,rrst_n)
   begin
     w2rintermediate<=wptr;
     rq2_wptr<=w2rintermediate;
   end
 endmodule
```

```
module top #(parameter data_bits=8,                          // top module
                  parameter address_bits=4
                  )
  (
    input logic [data_bits-1:0] wdata,
    input logic winc,wclk,wrst_n,rinc,rclk,rrst_n,
    output logic [data_bits-1:0] rdata,
    output logic wfull,rempty
  );
  logic [address_bits:0] rq2_wptr;
  logic [address_bits:0] wq2_rptr;
  logic [address_bits:0] rptr,wptr;
  logic [address_bits-1:0] raddr,waddr;
  SRAM DUT1(.wclken(winc),.wclk,.raddr,.waddr,.wdata,.rdata,.wfull);
  FIFO_empty DUT2(.rclk,.rinc,.rrst_n,.rq2_wptr,.rempty,.raddr,.rptr);          //instantiation of modules
  FIFO_FULL DUT3(.wclk,.winc,.wrst_n,.wq2_rptr,.wfull,.waddr,.wptr);
  r2wsynchronize DUT4(.rptr,.wclk,.wrst_n,.wq2_rptr);
  w2rsynchronize DUT5(.wptr,.rclk,.rrst_n,.rq2_wptr);


endmodule

```

All the modules are shown in the screenshots above.



This is the final waveform which verifies the functionality.

The EDA playground link is given

https://www.edaplayground.com/x/Puhs