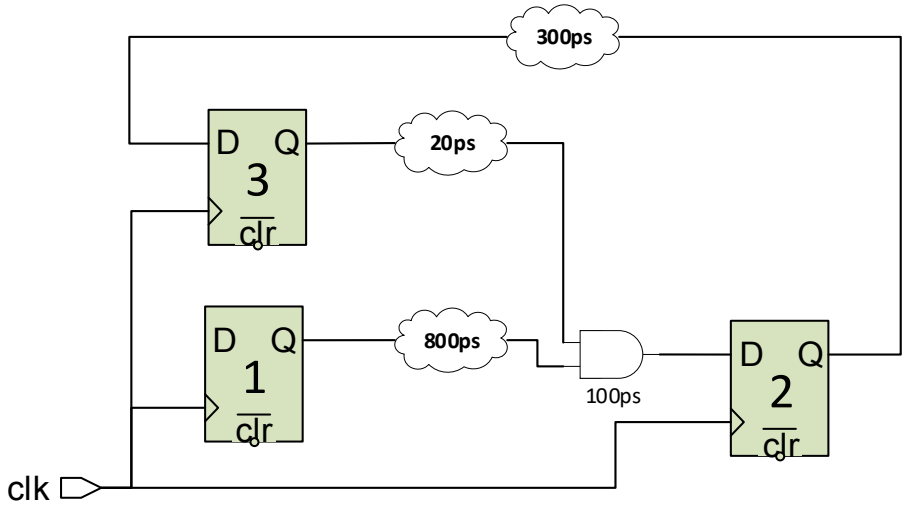
1. **(20 points) Timing**

   Given the following design:

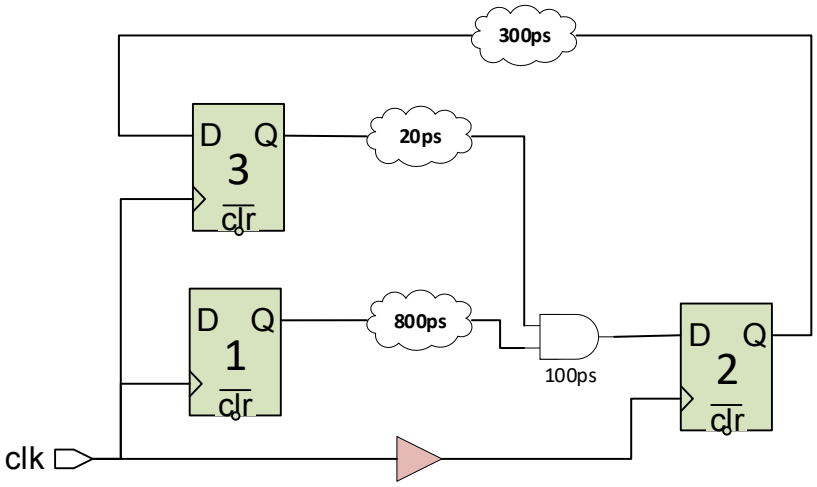


Fclk: 1GHz, ***Tsetup:*** 100ps, ***Thold:*** 150ps, ***Tcq (clock to q):*** 50ps

a. As you can see, there are 3 "flop to flop" timing paths in the design. Use the table below to answer the following questions:
   i. For each path, calculate the data arrival time and the data required time, for both setup and hold.
   ii. Which path has a violation? What is the violation type? What is the slack? Explain and include a wave diagram.

	FF1->FF2 setup	FF1->FF2 hold	FF3->FF2 setup	FF3->FF2 hold	FF2->FF3 setup	FF2->FF3 hold
Data arrival time						
Data required time						
Slack						
Meets timing						

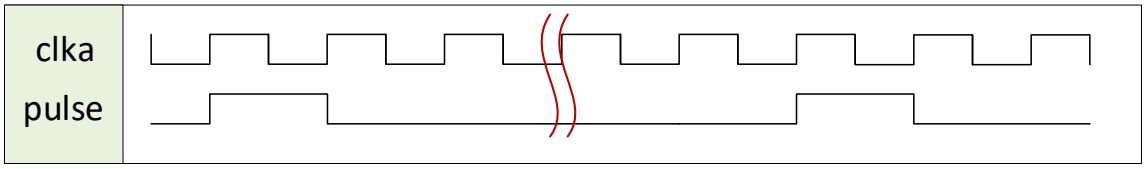
b. In order to fix the violation, one buffer was added as described below:



i. What should be the buffer delay in order to fix the timing violation?
ii. Update the table according to the design fix.
iii. Is there any new timing violation? how would you resolve it?

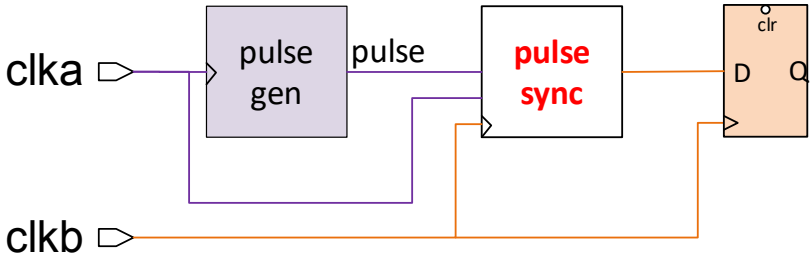
2. (20 points) Synchronization

pulse_gen is a block in **clka** domain, which generates 1 cycle pulses.
It is given that there are at least 100 clka cycles between pulse to pulse.



~100 cycles between pulses

We need to sample that pulse with **clkb**.



You should implement the small module ***pulse_sync*** for each one of the following cases:

a. clkb is 2 times faster than clka, the phase between the clocks is unknown (**pulse_sync1**)
b. clka is 2 times faster than clkb, the phase between the clocks is unknown (**pulse_sync2**)
c. The relationship between clka and clkb is unknown (you can assume ratio less than 10) (**pulse_sync3**)

For each of the 3 cases above choose the most efficient synchronization method.

The interface definition of the ***pulse_sync***

Pin name	Direction	Clock domain	Description
clka	in	-	See 3 cases
resetb_a	in	clka	Active low async reset
clkb	in	-	See 3 cases
resetb_b	in	clkb	Active low async reset
pulse_in	in	clka	1 clka cycle
pulse_out	out	clkb	Synchronized pulse

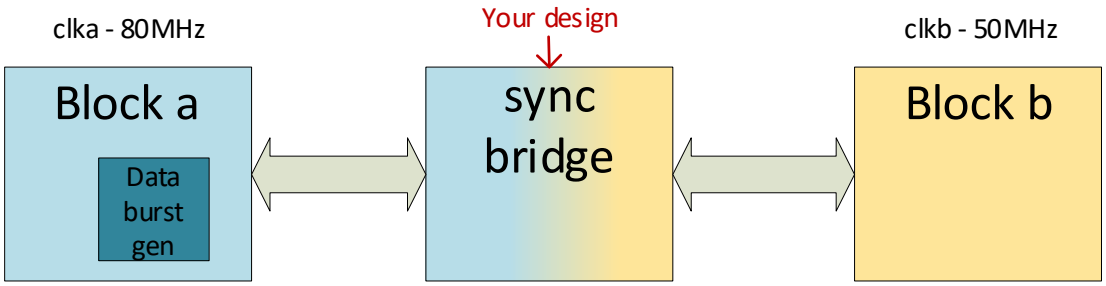
Submit for each case:

1. Schematic of the design
2. Wave diagram
3. Explain your considerations
4. SystemVerilog

Note: The SystemVerilog of all the 3 modules should be delivered **in the same file: pulse_sync.sv**

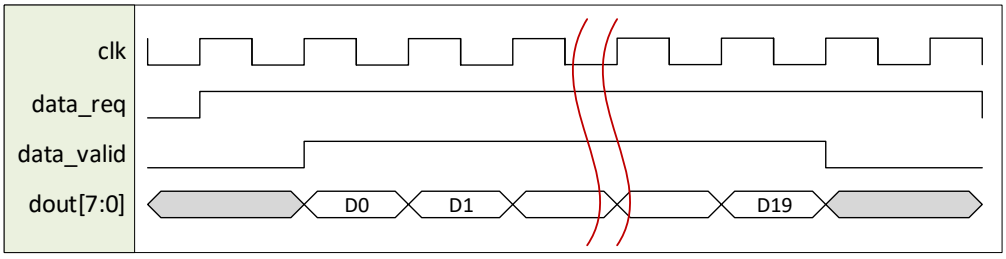
3. (60 points) Synch bridge

Design a module (*sync_bridge*) to synchronize between block **A** and block **B** as describes below. The *sync_bridge* should handle all CDC properly, to guarantee no metastability and no data loss.



Block A works at 80Mhz and Block B works at 50Mhz.

Block A generates a burst of 20 random numbers on *dout[7:0]* upon *data_req* is high.

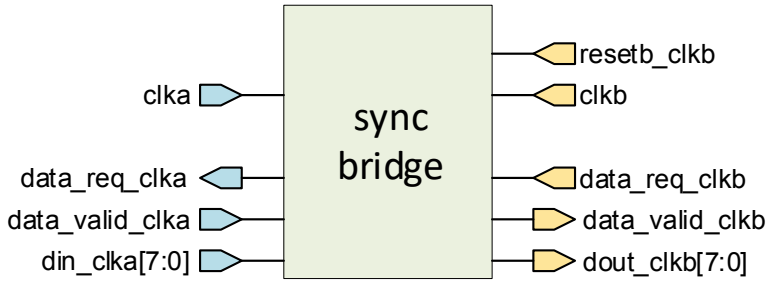


Block B drive the *data_req* high for one cycle and expect to get a burst of random data in response.

synch_bridge interface

Port name	Dir.	Clock domain	Description
clka	In	clka	80MHz clock
clkb	In	clkb	50MHz clock
resetb_clkb	In	clkb	Async active low reset, with positive edge synchronized with clkb
din_clka[7:0]	In	clka	Data from burst generator
data_req_clkb	In	clkb	Data request from block B (1 cycle)
data_valid_clka	In	clka	Data valid indication from block A
data_req_clka	Out	clka	Data request to block A
data_valid_clkb	Out	clkb	Data valid indication to block B
dout_clkb[7:0]	Out	clkb	Random data to block B

sync_bridge interface diagram



Implementation Guide

1) You probably noticed that an asynchronous FIFO is involved.
    a. Choose the FIFO depth and explain your calculations.
    b. Implement 2 **parameterized width** modules:
        i. bin2gray - Binary to Gray converter
        ii. gray2bin - Gray to Binary converter.
        You can find the algorithms in the web.
    c. Implement a standard asynchronous FIFO module according to the structure we learned in class.
2) Draw a diagram of your design - include the FIFO, the logic around, synchronizers etc. and the connectivity. (The FIFO's internal structure isn't needed)
3) Implement the top level design (sync_bridge.sv) in SystemVerilog, according to your diagram.
4) Write a testbench.
    a. Draw the testbench diagram.
    b. Implement separate drivers that mimic block A and block B behavior.
    c. Implement a monitor that collect the data and trigger the checker. For that purpose, you can use SystemVerilog queue data type.

```
integer queue1[$];                // Queue declaration
queue1.push_back(data);           // Push data
popped_data = queue1.pop_front(); // Pop data to a variable.
```

    d. Implement a checker which verifies that all transmitted data are successfully arrived to block B.

- Choose wisely an efficient synchronization method for each CDC signal. Explain your considerations in the report.

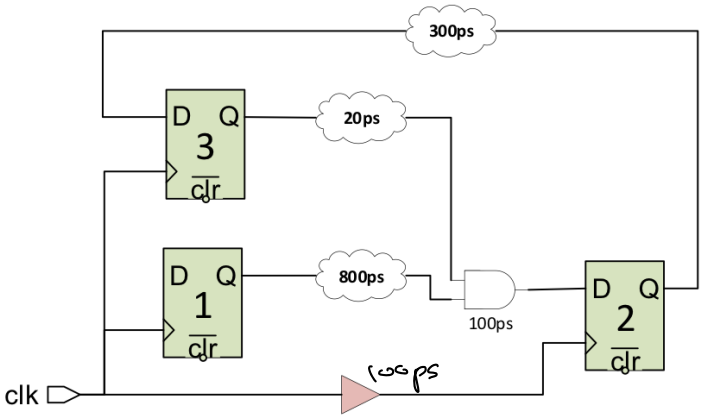
- Double ff synchronizer, with parameterized width and uncertainty model is given (dff_sync.sv) – Note: this module is only for simulation it can't be synthesized.
- You can assume long delay between data transaction and next *data_req*. (next data request will be sent a safe time after the system settled down).
- You can specify any other reasonable assumption you have and design accordingly.

Submit:

1) **(10 points)** Detailed diagram of the sync_bridge
2) **(10 points)** Block diagram of the testbench
3) **(10 points)** Design report (FIFO depth and other considerations explanation)
4) SystemVerilog files:
   (20 points) sync_bridge.sv (include all the needed sub modules and functions)
   (10 points) sync_bridge_tb.sv (include all the needed sub modules and functions)

Good luck!

b. In order to fix the violation, one buffer was added as described below:



i. What should be the buffer delay in order to fix the timing violation?
ii. Update the table according to the design fix.
iii. Is there any new timing violation? how would you resolve it?

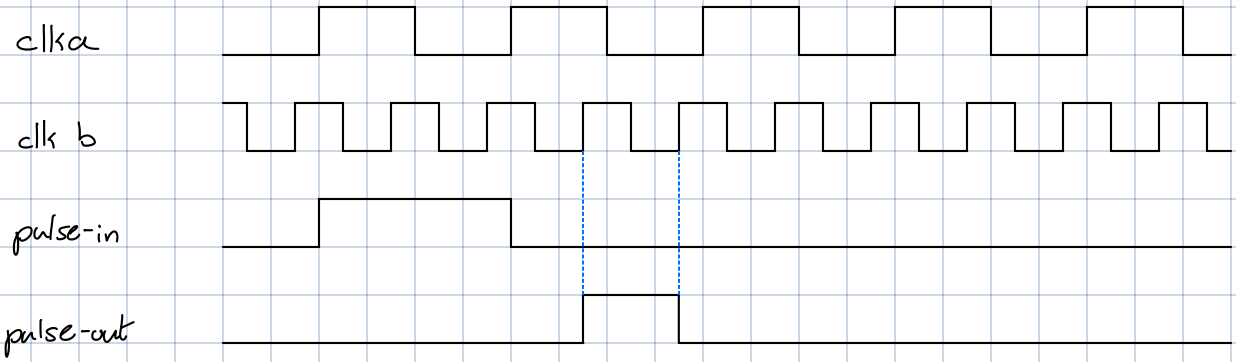
i) the buffer delay needs to be at least 50ps in order to avoid the setup violation. we will add 100ps for the calculation.

ii) the updated table:

	FF1->FF2 setup	FF1->FF2 hold	FF3->FF2 setup	FF3->FF2 hold	FF2->FF3 setup	FF2->FF3 hold
Data arrival time	950 ps	950 ps	170 ps	170 ps	350 ps	350 ps
Data required time	1000 ps	250 ps	1000 ps	250 ps	900 ps	150 ps
Slack	50 ps	700 ps	830 ps	-80 ps	550 ps	200 ps
Meets timing	yes	yes	yes	no	yes	yes

iii) yes there is a violation for FF3→FF2 regarding its hold time.
in order to resolve it we need to increase the delay between FF1 to FF2 from 20ps to at least 100ps.

wave diagram :




```
module pulse_sync1 (
    input  logic clka,
    input  logic clkb,
    input  logic rst_n_a,
    input  logic rst_n_b,
    input  logic pulse_in,
    output logic pulse_out
);

    logic  [2:0] sync_ff;

    always_ff @(posedge clkb or negedge rst_n_b)
        begin
        if (!rst_n_b)
        begin
            sync_ff <= 3'b0;
        end
        else
        begin
            sync_ff <= {sync_ff[1:0], pulse_in};
        end
    end

    assign pulse_out = sync_ff[1] & ~sync_ff[2];

endmodule
```

```
module pulse_sync2 (
    input  logic clka,
    input  logic clkb,
    input  logic rst_n_a,
    input  logic rst_n_b,
    input  logic pulse_in,
    output logic pulse_out
);

    logic  [2:0] sync_ff_b;
    logic  [2:0] sync_ff_a;

    always_ff @(posedge clkb or negedge rst_n_b)
        begin
        if (! rst_n_b)
        begin
            sync_ff_b <= 3'b0;
        end
        else
        begin
            sync_ff_b <= {sync_ff_b[1:0], sync_ff_a[2]};
        end
    end

    assign pulse_out = sync_ff_b[1] & ~ sync_ff_b[2];

    always_ff @(posedge clka or negedge rst_n_a)
        begin
        if (! rst_n_a)
        begin
            sync_ff_a <= 3'b0;
        end
        else
        begin
            sync_ff_a[0] <= pulse_in;
            sync_ff_a[1] <= pulse_in || sync_ff_a[0];
            sync_ff_a[2] <= pulse_in || sync_ff_a[1];
        end
    end


endmodule
```

```
module pulse_sync3 (
    input  logic clka,
    input  logic clkb,
    input  logic rst_n_a,
    input  logic rst_n_b,
    input  logic pulse_in,
    output logic pulse_out
);

    logic  [2:0] sync_ff_b;
    logic  original_pulse;
    logic  [1:0] sync_ff_a;

    always_ff @(posedge clkb or negedge rst_n_b)
        begin
        if (!rst_n_b)
        begin
            sync_ff_b <= 3'b0;
        end
        else
        begin
            sync_ff_b <= {sync_ff_b[1:0], sync_ff_a[2]};
        end
    end

    assign pulse_out = sync_ff_b[1] & ~ sync_ff_b[2];

    always_ff @(posedge clka or negedge rst_n_a)
        begin
        if (! rst_n_a)
        begin
            original_pulse <= 0;
        end
        else
        begin
            original_pulse <= pulse_in || (~sync_ff_a[1] && original_pulse);
        end
    end

    always_ff @(posedge clka or negedge rst_n_a)
        begin
        if (!rst_n_a)
        begin
            sync_ff_a <= 2'b0;
        end
        else
        begin
            sync_ff_a <= {sync_ff_b[0], sync_ff_b[1]};
        end
    end


endmodule
```

Question 3

1)

We calculate the minimum FIFO depth according to what was learned in the lecture:

f_a = 80[Mhz], T_a = 12.5[ns]
f_b = 50[Mhz], T_b = 20[ns]
$Burst_a$ = 20

- All writes A: $Burst_a$ * T_a = 20 * 12.5[ns] = 250[ns]
- According to what we learned in the lecture, there is a synchronization delay of 3 cycles in Block B's clock domain to ensure we have fully read the data: T_{sync}=3×20ns=60[ns]
- The remaining time available for reading after synchronization is: T_{read}=T_{write}−T_{sync}=250[ns] – 60[ns]=190[ns]
- Each read takes 20 ns. The number of data words read during this time: N_{read}= T_{read} / $T_{cycle,B}$ = 190[ns] / 20[ns] = 9.5 ≈ 9
- Out of the 20 data words written by Block A, 9 words are read by Block B during the available time, so The remaining data in the FIFO: $N_{remaining}$ = 20 – 9 = 11
- According to what was learned in the lecture, we need to leave 2 extra unused addresses in the FIFO for safety: $N_{required}$ = $N_{remaining}$ + 2 = 11 + 2 = 13
- FIFO sizes are powers of 2, so 13 is rounded up to 16
- The required FIFO depth is 16 words.

b)

```
module bin2gray #(
    parameter PTR_WIDTH = 4
)(
    input  logic [PTR_WIDTH-1:0] bin_in,
    output logic [PTR_WIDTH-1:0] gray_out
);
    assign gray_out = bin_in ^ (bin_in >> 1);
endmodule

module gray2bin #(
    parameter PTR_WIDTH = 4
)(
    input  logic [PTR_WIDTH-1:0] gray_in,
    output logic [PTR_WIDTH-1:0] bin_out
);
    integer i;
    always_comb begin
        bin_out[PTR_WIDTH-1] = gray_in[PTR_WIDTH-1];
        for (i = PTR_WIDTH-2; i >= 0; i = i - 1) begin
            bin_out[i] = bin_out[i+1] ^ gray_in[i];
        end
    end
endmodule
```

```
module pulse_sync1 (
    input  logic clka,
    input  logic clkb,
    input  logic rst_n_a,
    input  logic rst_n_b,
    input  logic pulse_in,
    output logic pulse_out
);

    logic  [2:0] sync_ff;

    always_ff @(posedge clkb or negedge rst_n_b)
        begin
        if (!rst_n_b)
        begin
            sync_ff <= 3'b0;
        end
        else
        begin
            sync_ff <= {sync_ff[1:0], pulse_in};
        end
    end

    assign pulse_out = sync_ff[1] & ~sync_ff[2];

endmodule



module pulse_sync2 (
    input  logic clka,
    input  logic clkb,
    input  logic rst_n_a,
    input  logic rst_n_b,
    input  logic pulse_in,
    output logic pulse_out
);

    logic  [2:0] sync_ff_b;
    logic  [2:0] sync_ff_a;

    always_ff @(posedge clkb or negedge rst_n_b)
        begin
        if (! rst_n_b)
        begin
            sync_ff_b <= 3'b0;
        end
        else
        begin
            sync_ff_b <= {sync_ff_b[1:0], sync_ff_a[2]};
        end
    end

    assign pulse_out = sync_ff_b[1] & ~ sync_ff_b[2];
```

```
    always_ff @(posedge clka or negedge rst_n_a)
        begin
        if (! rst_n_a)
        begin
            sync_ff_a <= 3'b0;
        end
        else
        begin
            sync_ff_a[0] <= pulse_in;
            sync_ff_a[1] <= pulse_in || sync_ff_a[0];
            sync_ff_a[2] <= pulse_in || sync_ff_a[1];
        end
    end


endmodule

module pulse_sync3 (
    input  logic clka,
    input  logic clkb,
    input  logic rst_n_a,
    input  logic rst_n_b,
    input  logic pulse_in,
    output logic pulse_out
);

    logic  [2:0] sync_ff_b;
    logic  original_pulse;
    logic  [1:0] sync_ff_a;

    always_ff @(posedge clkb or negedge rst_n_b)
        begin
        if (!rst_n_b)
        begin
            sync_ff_b <= 3'b0;
        end
        else
        begin
            sync_ff_b <= {sync_ff_b[1:0], sync_ff_a[2]};
        end
    end

    assign pulse_out = sync_ff_b[1] & ~ sync_ff_b[2];

    always_ff @(posedge clka or negedge rst_n_a)
        begin
        if (! rst_n_a)
        begin
            original_pulse <= 0;
        end
        else
        begin
            original_pulse <= pulse_in || (~sync_ff_a[1] &&
```

```
original_pulse);
        end
    end

    always_ff @(posedge clka or negedge rst_n_a)
        begin
        if (!rst_n_a)
        begin
            sync_ff_a <= 2'b0;
        end
        else
        begin
            sync_ff_a <= {sync_ff_b[0], sync_ff_b[1]};
        end
    end


endmodule
```

```
// Parameter Declarations
parameter DATA_WIDTH = 8;
parameter PTR_WIDTH = 4;
parameter FIFO_DEPTH = 16;

// SYNC_BRIDGE MODULE
module sync_bridge #(parameter DATA_WIDTH = 8) (
    input logic clka,
    input logic clkb,
    input logic resetb_clkb,
    input logic data_req_clkb,
    input logic data_valid_clka,
    output logic data_req_clka,
    output logic data_valid_clkb,
    input logic [DATA_WIDTH-1:0] din_clka,
    output logic [DATA_WIDTH-1:0] dout_clkb
);

    // Internal logic signals
    logic data_req_clkb_sync;
    logic resetb_clka;
    logic wr_enable;
    logic fifo_full;
    logic fifo_empty;

    // Reset Synchronization
    dff_sync #(.WIDTH(1)) reset_sync (
        .clk(clka),
        .resetb(resetb_clkb),
        .d(1'b1),
        .q(resetb_clka)
    );

    // Synchronize data_req_clkb to clka domain
    always_ff @(posedge clkb or negedge resetb_clkb) begin
        if (!resetb_clkb)
            data_req_clkb_sync <= 1'b0;
        else
            data_req_clkb_sync <= data_req_clkb;
    end

    dff_sync #(.WIDTH(1)) data_req_sync (
        .clk(clka),
        .resetb(resetb_clka),
        .d(data_req_clkb_sync),
        .q(data_req_clka)
    );

    // Write enable logic
    assign wr_enable = ~fifo_full & data_valid_clka & resetb_clka &
resetb_clkb;

    // Generate data_valid_clkb based on FIFO status
    always_ff @(posedge clkb or negedge resetb_clkb) begin
```

```
        if (!resetb_clkb)
            data_valid_clkb <= 1'b0;
        else
            data_valid_clkb <= ~fifo_empty;
    end

    // Instantiate Asynchronous FIFO
    async_fifo #(.FIFO_DEPTH(FIFO_DEPTH)) data_fifo (
        .wr_clk(clka),
        .rd_clk(clkb),
        .wr_resetb(resetb_clka),
        .rd_resetb(resetb_clkb),
        .wr_en(wr_enable),
        .rd_en(1'b1),
        .wr_full(fifo_full),
        .rd_empty(fifo_empty),
        .wr_data(din_clka),
        .rd_data(dout_clkb)
    );
endmodule

// ASYNC_FIFO MODULE
module async_fifo #(parameter FIFO_DEPTH = 16) (
    input logic wr_clk,
    input logic rd_clk,
    input logic wr_resetb,
    input logic rd_resetb,
    input logic wr_en,
    input logic rd_en,
    output logic wr_full,
    output logic rd_empty,
    input logic [DATA_WIDTH-1:0] wr_data,
    output logic [DATA_WIDTH-1:0] rd_data
);
    logic [PTR_WIDTH-1:0] wr_ptr_bin, rd_ptr_bin;
    logic [PTR_WIDTH-1:0] wr_ptr_gray, rd_ptr_gray;
    logic [PTR_WIDTH-1:0] wr_next_bin, rd_next_bin;
    logic [DATA_WIDTH-1:0] memory [FIFO_DEPTH-1:0];

    // Writing Data
    always_ff @(posedge wr_clk)
        if (wr_en & ~wr_full)
            memory[wr_ptr_bin] <= wr_data;

    // Read Data
    assign rd_data = (wr_resetb & rd_resetb) ? memory[rd_ptr_bin] :
8'b0;

    // Pointer Increment Logic
    always_ff @(posedge wr_clk or negedge wr_resetb)
        if (!wr_resetb)
            wr_ptr_bin <= 4'b0;
        else if (wr_en & ~wr_full)
            wr_ptr_bin <= wr_next_bin;
```

```
    always_ff @(posedge rd_clk or negedge rd_resetb)
        if (!rd_resetb)
            rd_ptr_bin <= 4'b1111;
        else if (rd_en & ~rd_empty)
            rd_ptr_bin <= rd_next_bin;

    assign wr_next_bin = (wr_ptr_bin < 4'b1111) ? wr_ptr_bin + 1 :
4'b0000;
    assign rd_next_bin = (rd_ptr_bin < 4'b1111) ? rd_ptr_bin + 1 :
4'b0000;

    // Synchronization and full/empty logic
    dff_sync #(.WIDTH(4)) wr_ptr_sync (
        .clk(rd_clk),
        .resetb(rd_resetb),
        .d(wr_ptr_gray),
        .q(wr_ptr_gray_sync)
    );

    dff_sync #(.WIDTH(4)) rd_ptr_sync (
        .clk(wr_clk),
        .resetb(wr_resetb),
        .d(rd_ptr_gray),
        .q(rd_ptr_gray_sync)
    );

    gray2bin rd_ptr_converter
(.gray_in(rd_ptr_gray_sync), .bin_out(rd_bin_ptr_comparator));
    gray2bin wr_ptr_converter
(.gray_in(wr_ptr_gray_sync), .bin_out(wr_bin_ptr_comparator));

    assign wr_full = (wr_next_bin == rd_bin_ptr_comparator);
    assign rd_empty = (rd_next_bin == wr_bin_ptr_comparator);
endmodule

// Binary to Gray Code Converter
module bin2gray #(parameter PTR_WIDTH = 4) (
    input logic [PTR_WIDTH-1:0] bin_in,
    output logic [PTR_WIDTH-1:0] gray_out
);
    assign gray_out = (bin_in >> 1) ^ bin_in;
endmodule

// Gray to Binary Code Converter
module gray2bin #(parameter PTR_WIDTH = 4) (
    input logic [PTR_WIDTH-1:0] gray_in,
    output logic [PTR_WIDTH-1:0] bin_out
);
    integer i;
    always_comb begin
        bin_out[PTR_WIDTH-1] = gray_in[PTR_WIDTH-1];
        for (i = PTR_WIDTH-2; i >= 0; i = i - 1)
            bin_out[i] = bin_out[i+1] ^ gray_in[i];
```

```
    end
endmodule
```

```
`timescale 1ns/1ns
module sync_bridge_tb();

    logic clka;
    logic clkb;
    logic data_req_clka;
    logic data_valid_clka;
    logic [7:0] din_clka;
    logic resetb_clkb;
    logic data_req_clkb;
    logic data_valid_clkb;
    logic [7:0] dout_clkb;
    logic [7:0] popped_data;
    integer queue1[$]; // Queue declaration

    // Instantiate the sync_bridge module
    sync_bridge uut (
        .clka(clka),
        .clkb(clkb),
        .resetb_clkb(resetb_clkb),
        .data_req_clkb(data_req_clkb),
        .data_valid_clka(data_valid_clka),
        .data_req_clka(data_req_clka),
        .data_valid_clkb(data_valid_clkb),
        .din_clka(din_clka),
        .dout_clkb(dout_clkb)
    );

    // Clock Generation
    always #6.25ns clka = ~clka;
    always #10ns clkb = ~clkb;

    // Synchronization Task
    task automatic sync();
        @(posedge clkb);
        #1ns;
    endtask

    // Random Input Generator
    function void generate_random_data();
        din_clka = $random();
    endfunction

    // Driver A - Mimicking Block A Behavior
    task automatic drive_a();
        @(posedge clka);
        generate_random_data();
        data_valid_clka = 1'b1;
    endtask

    // Driver B - Mimicking Block B Behavior
    task automatic drive_b();
        sync();
        resetb_clkb = 1'b1;
```

```
        #20ns;
        sync();
        data_req_clkb = 1'b1;
    endtask

    // Monitor Block - Capturing Data Transactions
    initial forever begin
        @(posedge clka);
        #1ns;
        if (data_valid_clka == 1'b1)
            queue1.push_back(din_clka);
    end

    initial forever begin
        @(posedge clkb);
        #1ns;
        if (data_valid_clkb == 1'b1)
            verify_data();
    end

    // Data Checker Function
    function void verify_data();
        popped_data = queue1.pop_front();
        if (popped_data != dout_clkb)
            $display("ERROR: Mismatch! Queue Data = %b, dout_clkb =
%b", popped_data, dout_clkb);
        else
            $display("SUCCESS: Data transmitted correctly!");
    endfunction

    // Checking data_req_clka signal
    task automatic validate_req_clka();
        fork
            begin
                wait (data_req_clka);
                $display("data_req_clka asserted correctly.");
            end
            begin
                #52ns;
                $display("ERROR: data_req_clka was not asserted in
time!");
                $stop;
            end
        join_any
        disable fork;
    endtask

    // Testbench Execution
    initial begin
        {clka, clkb, resetb_clkb, data_req_clka, data_req_clkb,
data_valid_clka, data_valid_clkb, din_clka, dout_clkb, popped_data}
= 0;
        #10ns;
        drive_b();
```

```
        validate_req_clka();
        wait(data_req_clka);
        repeat (21) drive_a();
        din_clka = 8'b0; // Clear Data
        #13ns;
        data_valid_clka = 1'b0;
        #1000ns;
        $stop;
    end

endmodule
```