DEPARTMENT OF ELECTRONIC AND TELECOMMUNICATION ENGINEERING

UNIVERSITY OF MORATUWA

EN4021 - Advanced Digital Systems

System Bus Design

PASQUAL A.C.	200445V
RATHNAYAKE R.N.P.	200537F

January 2, 2025

Table of Contents

1. Overview

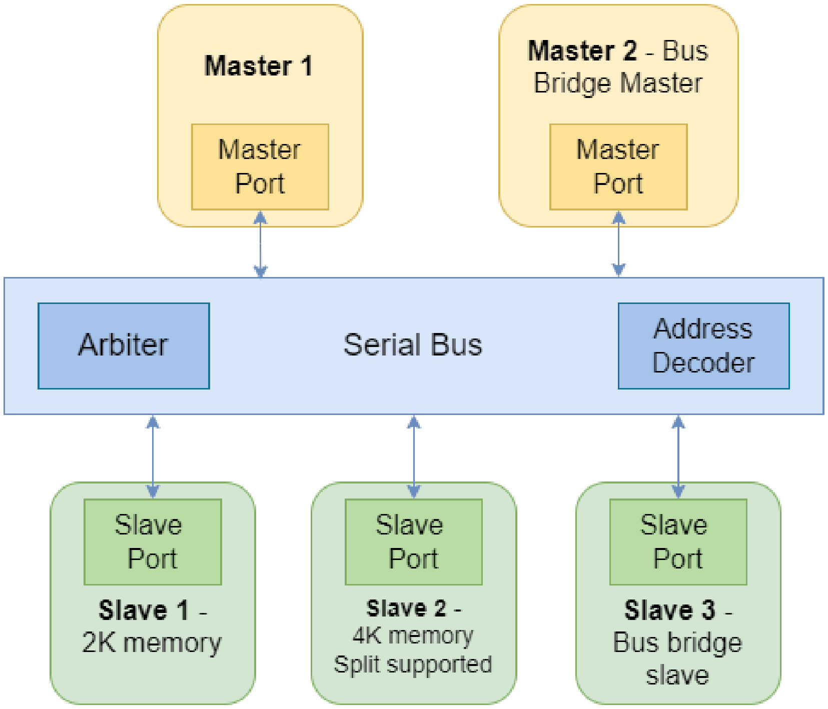


Figure: Overview of the bus architecture

Our bus supports 2 masters and 3 slaves. One master and one slave can be configured as a bus bridge. Slave 2 (or any other slave) can be configured to support split transactions.

The bus interconnect contains several parts:

- **Arbiter:** Gives priority to Master 1 over Master 2 when both masters request access at the same time.

- **Address decoder:** Decodes the address to identify which slave to select

- **Multiplexers and decoders:** Connect required master and slave ports through the bus based on control signals from the arbiter and address decoder

The following figure shows the internal structure of the bus.

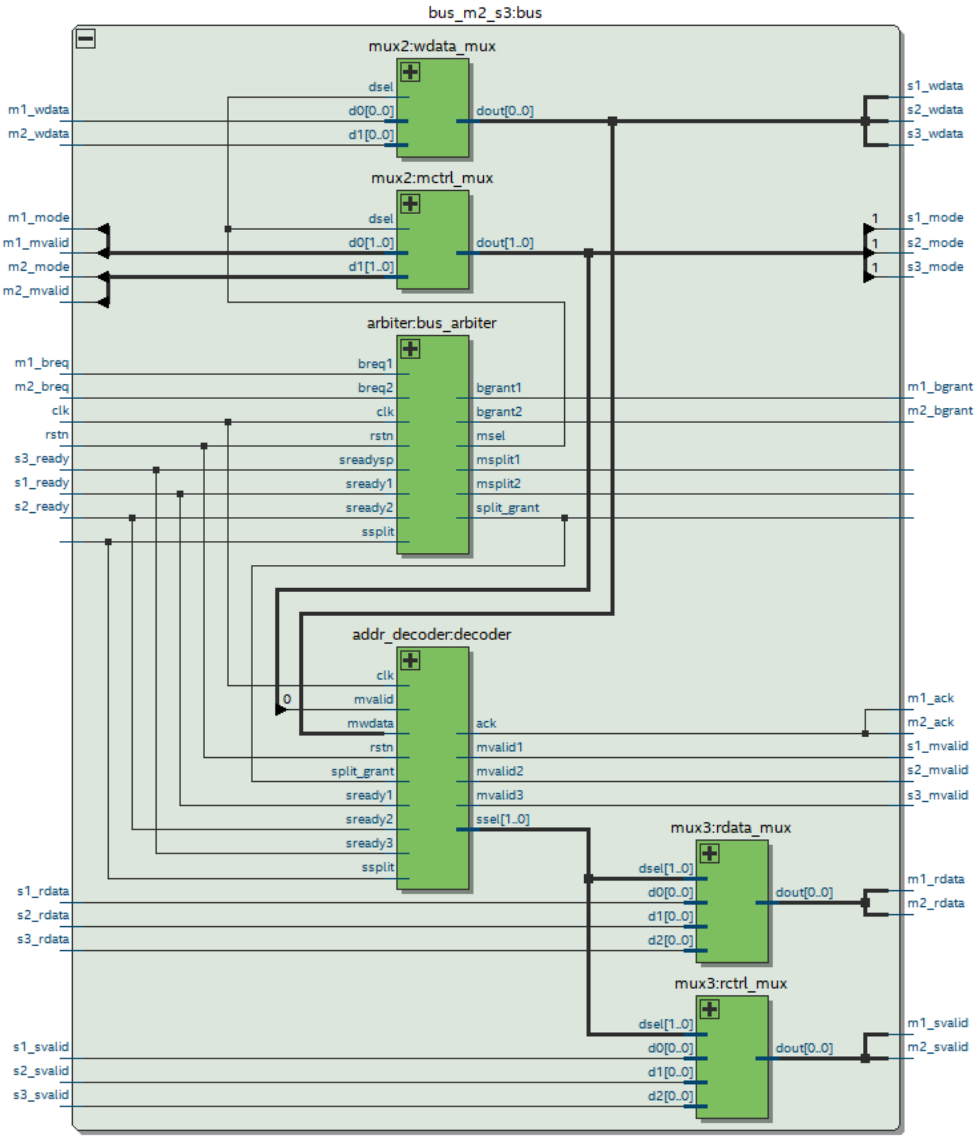


Figure: Internal structure of the bus

2. Address Allocation

2.1. Without Bus Bridge

When the bus bridge is not used, the slaves' internal memories are as follows:

- Slave 1 - 2 KB
- Slave 2 - 4 KB
- Slave 3 - 4 KB

Then, the address space is allocated as follows with a 14-bit address.

Slave	Allocated Address Space
Slave 1 (2 KB)	`00 0XXX XXXX XXXX`
Slave 2 (4 KB)	`01 XXXX XXXX XXXX`
Slave 3 (4 KB)	`10 XXXX XXXX XXXX`

Slave 1 (2K)	▨	Slave 2 (4K)	Slave 3 (4K)	▨

2.2. With Bus Bridge

When the bus bridge is used, our bus bridge slave and master connect to the other bus’s master and slave respectively. The other bus contains 3 non-bus bridge slaves accessible through the bus bridge with the following memories:

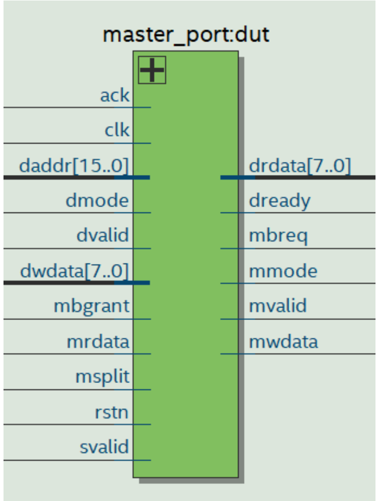
- Slave 1 - 2 KB
- Slave 2 - 4 KB
- Slave 3 - 4 KB

Therefore, masters on our bus view the bus bridge slave as a 10 KB memory space. The address space is allocated as follows with a 16-bit address.

Slave	Allocated Address Space
Slave 1 (2 KB)	`0000 0XXX XXXX XXXX`
Slave 2 (4 KB)	`0100 XXXX XXXX XXXX`
Slave 3 - Bus Bridge (10 KB)	`10XX XXXX XXXX XXXX`

3. Master Port

3.1. Block Diagram

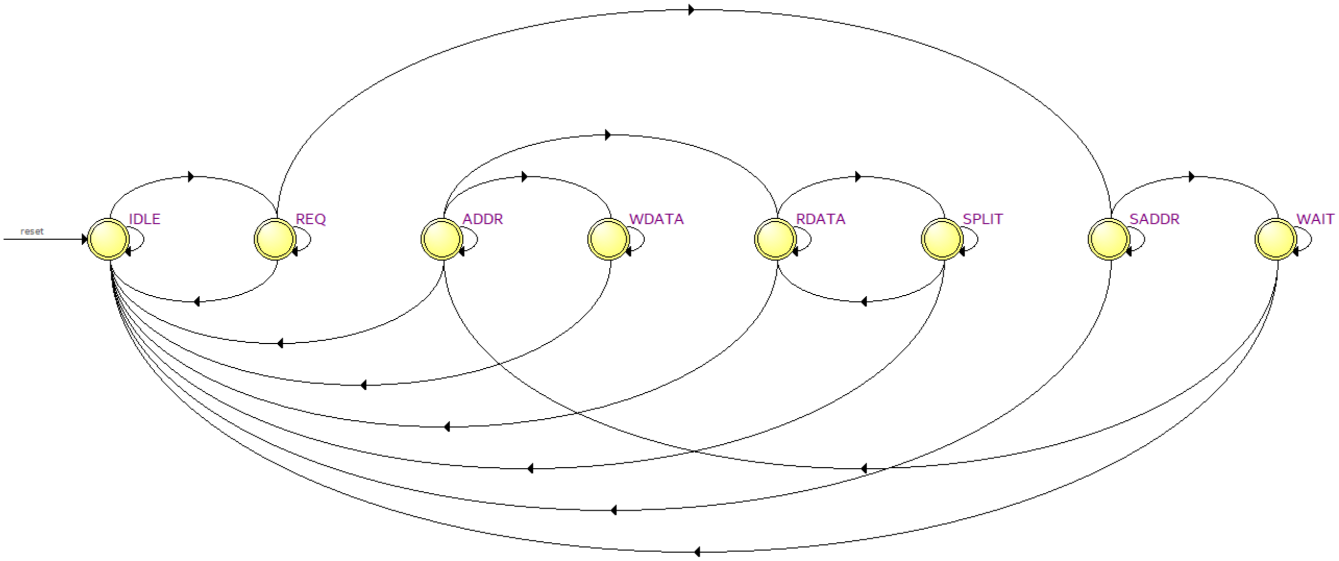


3.2. IO Description

Input Pin	Description
`clk`	Clock
`rstn`	Active low reset
`daddr[15..0]`	Address to be used for transaction
`dwdata[7..0]`	Data to be used for write transaction
`dmode`	Select mode of transaction (0 = Read, 1 = Write)
`dvalid`	Indicates that daddr, dwdata, dmode are valid
`mbgrant`	Indicates that master has access to the bus
`msplit`	Indicates that the currently connected slave has split
`mrdata`	Serial data input line (to read from slave)
`svalid`	Indicates that data on mrdata is valid
`ack`	Acknowledgement that bus has connected to requested slave successfully. Not receiving this indicates that either the address is invalid or the slave is busy (split).

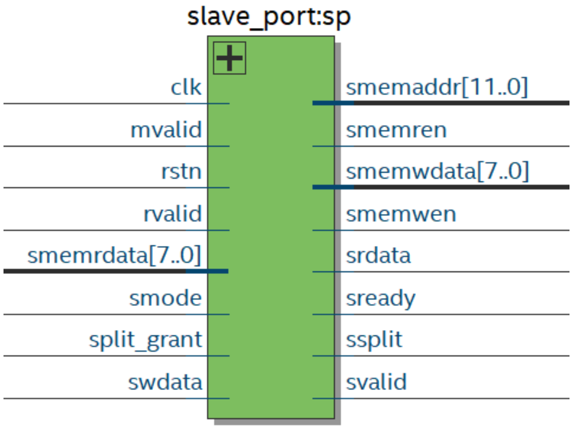
Output Pin	Description
`drdata[7..0]`	Data received from read transaction
`dready`	Indicates that master port is idle and ready for another transaction
`mbreq`	Bus request signal
`mmode`	Mode of transaction sent to bus (0 = Read, 1 = Write)
`mwdata`	Serial data output line (to send address and write data)
`mvalid`	Indicates that data on mwdata is valid

3.3. State Diagram



4. Slave Port

4.1. Block Diagram



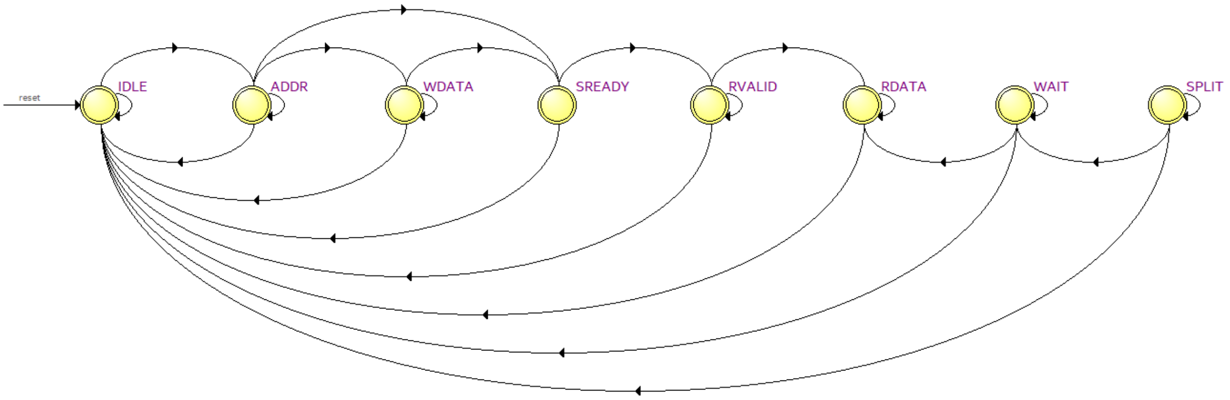
4.2. IO Description

Input Pin	Description
`clk`	Clock
`mvalid`	Indicates receiving write data and write address from master
`rstn`	Active low reset
`rvalid`	Indicates the read data is available from the slave memory (BRAM)
`smemrdata[7..0]`	Data read from slave memory
`smode`	Mode of transaction (Read - 0, Write - 1)
`split_grant`	Indicates sending read data is granted in a split transaction
`swdata`	Write data and address from master

Output Pin	Description
`smemaddr[11..0]`	Slave memory address to be read or written
`smemren`	Read enable signal to slave memory

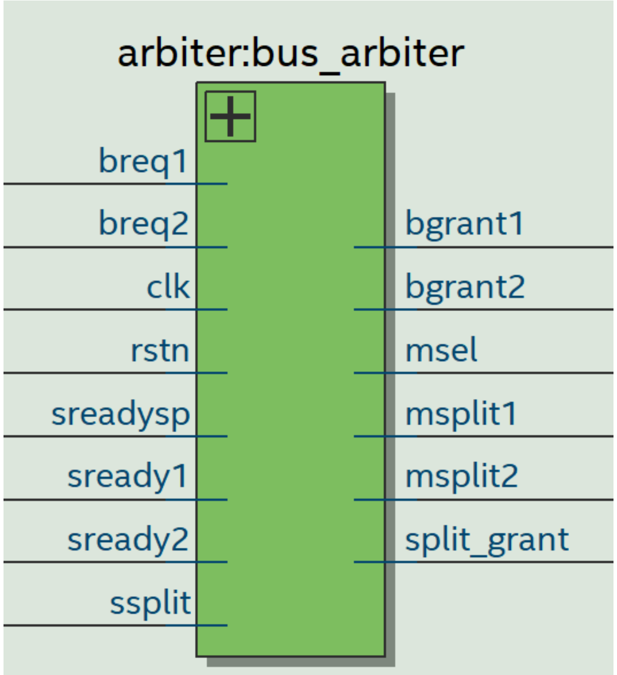
smemwdata[7..0]	Data to be written to slave memory
smemwen	Write enable signal to slave memory
srdata	Read data to transmit to the master
sready	Indicates slave is idle and ready for transaction
ssplit	Indicates split transaction
svalid	Indicates read data is transmitting

4.3. State Diagram



5. Arbiter

5.1. Block Diagram

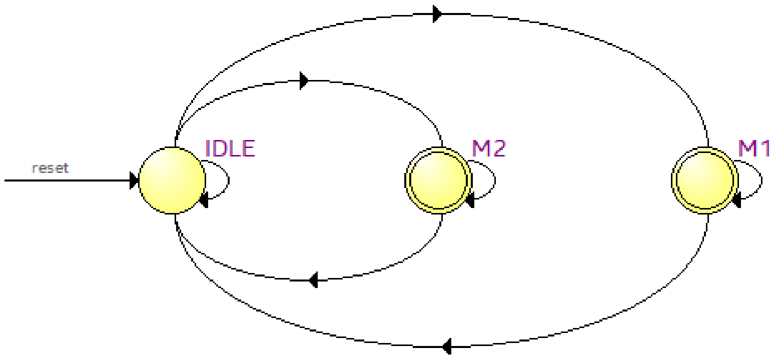


5.2. IO Description

Input Pin	Description
`breq1`	Indicates bus access request from master 1
`breq2`	Indicates bus access request from master 2
`clk`	Clock
`rstn`	Active low reset
`sreadysp`	Indicates the split-supported slave is ready for transaction
`sready1`	Indicates that slave 1 is ready for transaction
`sready2`	Indicates that slave 2 is ready for transaction
`ssplit`	Indicates the split-supported slave is on a split transaction

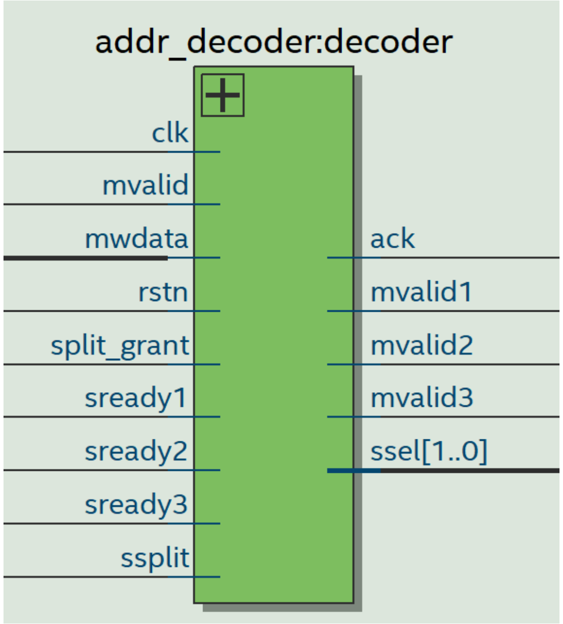
Output Pin	Description
`bgrant1`	Indicates bus access is granted for master 1
`bgrant2`	Indicates bus access is granted for master 2
`msel`	Selects which master initiates the transaction (master 1 - 0, master 2 - 1)
`msplit1`	Indicates the slave is on a split transaction to master 1
`msplit2`	Indicates the slave is on a split transaction to master 2
`split_grant`	Indicates bus access is granted to continue split transaction

5.3. State Diagram



6. Address Decoder

6.1. Block Diagram

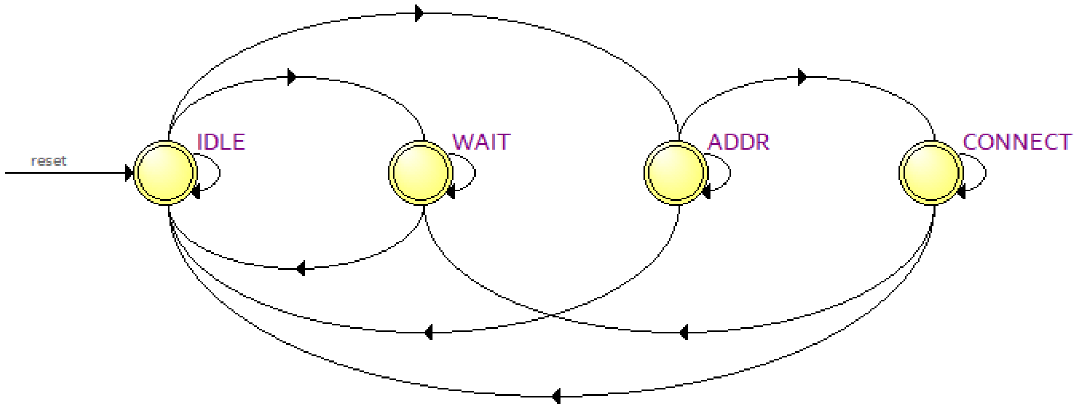


6.2. IO Description

Input Pin	Description
`clk`	Clock signal
`rstn`	Active low reset
`mwdata`	Serial data from master
`mvalid`	Indicates that data on mwdata is valid
`split_grant`	From arbiter, indicates that access is granted to continue split transaction
`sready1`	Indicates that slave 1 is ready for transaction
`sready2`	Indicates that slave 2 is ready for transaction
`sready3`	Indicates that slave 3 is ready for transaction
`ssplit`	Indicates the split-supported slave is on a split transaction

Output Pin	Description
`ack`	Acknowledgement sent to master that the bus successfully connected to the requested slave
`mvalid1`	Serial data from master valid signal sent to slave 1
`mvalid2`	Serial data from master valid signal sent to slave 2
`mvalid3`	Serial data from master valid signal sent to slave 3
`ssel[1..0]`	Slave select (00 = Slave 1, 01 = Slave 2, 10 = Slave 3)

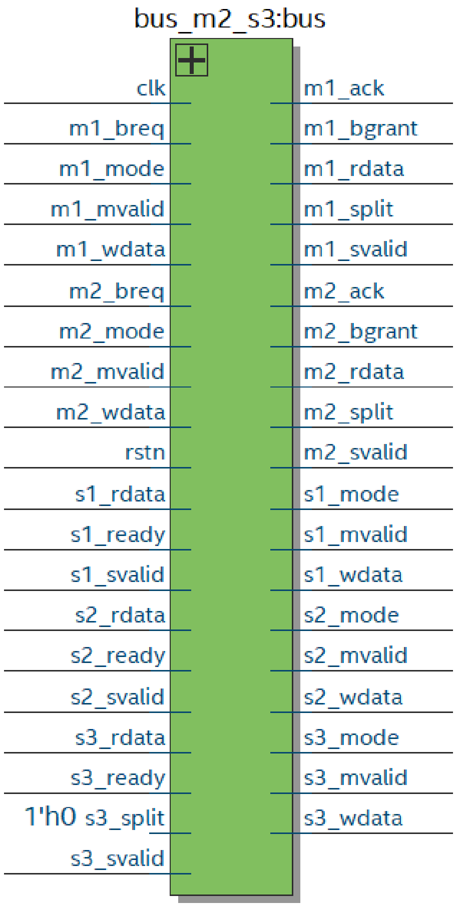
6.3. State Diagram



7. Bus Interconnect

The bus interconnect contains the arbiter, address decoder, multiplexers, and decoders to route connections between masters and slaves connected to the bus. It has ports to connect 2 masters and 3 slaves.

7.1. IO Diagram



7.2. IO Description

Input Pin	Description
`clk`	Clock signal
`rstn`	Active low reset
`mx_breq`	Bus access request from master x

mx_mode	Mode of transaction from master x
mx_wdata	Serial data input from master x (write data or address)
mx_mvalid	Indicates that data on mx_wdata is valid
sx_rdata	Serial data input from slave x (read data)
sx_ready	Indicates that slave x is ready for another transaction
sx_svalid	Indicates that data on sx_rdata is valid
s3_split	Split signal from slave 3 (here slave 3 is configured for split transaction)

Output Pin	**Description**
mx_ack	Acknowledgement to master x that bus has connected to requested slave successfully
mx_bgrant	Bus grant signal to master x
mx_rdata	Serial data output from a slave to master x
mx_svalid	Indicates that mx_rdata is valid
mx_split	Indicates to master x that the slave has split
sx_mode	Mode of transaction given to slave x
sx_wdata	Serial data output from a master to slave x
sx_mvalid	Indicates that sx_wdata is valid

8. Bus Bridge

The bus bridge master and slave modules contain a normal master and slave port within them to interface with the serial bus, and contain additional logic to communicate with the other bus through UART.

8.1. Bus Bridge Master

The bus bridge master interfaces with the other buses' slave bus bridge through UART. It is used to receive read/write transaction commands initiated by the other bus and to send output data from read transactions back to the other bus.

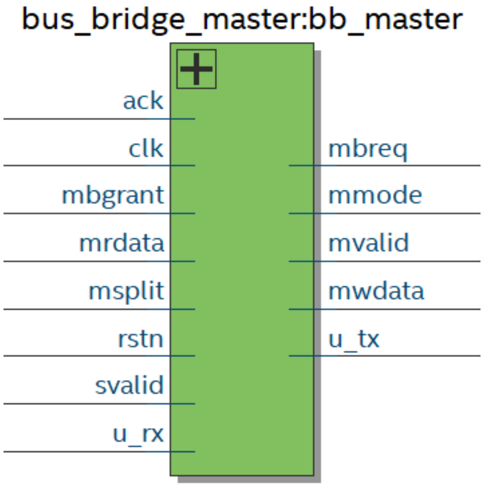
For receiving transaction commands, a UART port configured for 23-bit words is used. The received command is arranged as follows:

1	14	8
Mode	Address	Data

The 14-bit address is converted to the 16-bit format of our bus before the master begins the transaction. Incoming commands are stored in a FIFO until the master is ready.

For sending back the 8-bit read data, a UART port configured for 8-bit words is used.

Functional Block Diagram



IO Description

Input Pin	Description
`clk`	Clock
`rstn`	Active low reset
`mbgrant`	Indicates that master has access to the bus
`mrdata`	Serial data input line (to read from slave)
`svalid`	Indicates that data on mrdata is valid
`msplit`	Indicates that the slave in the current transaction has split
`ack`	Acknowledgement that bus has connected to requested slave successfully. Not receiving this indicates that either the address is invalid or the slave is busy (split).
`u_rx`	UART receiver data line

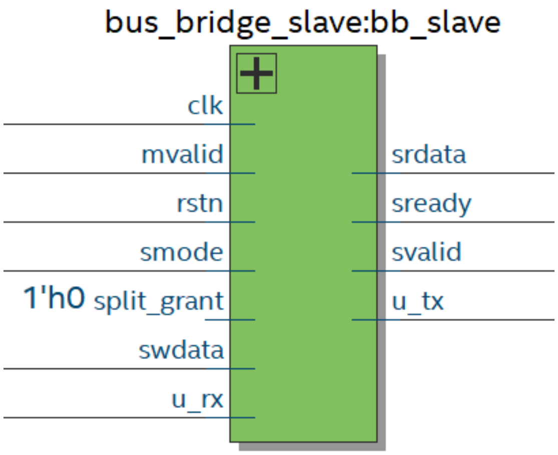
Output Pin	Description
`mbreq`	Bus request signal
`mmode`	Mode of transaction sent to bus (0 = Read, 1 = Write)
`mwdata`	Serial data output line (to send address and write data)
`mvalid`	Indicates that data on mwdata is valid
`u_tx`	UART transmitter data line

8.2. Bus Bridge Slave

The bus bridge slave interfaces with the other buses' master bus bridge through UART. It is used to send read/write transaction commands to the other bus and to receive output data from read transactions made across the bus bridge.

The bus bridge slave uses a 23-bit UART port and an 8-bit UART port for sending transaction commands and receiving read data respectively, using the same data format as the bus bridge master.

Functional Block Diagram



IO Description

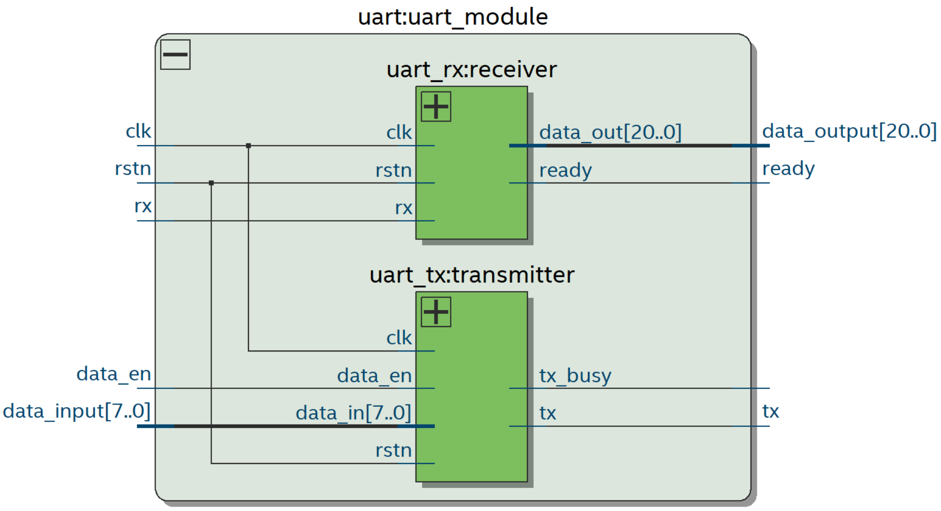
Input Pin	Description
clk	Clock
mvalid	Indicates receiving write data and write address from master
rstn	Active low reset
smode	Read - 0, Write - 1
split_grant	Indicates sending read data is granted in a split transaction
swdata	Write data and address from master
u_rx	UART receiver data line

Output Pin	Description
srdata	Read data to transmit to the master
sready	Indicates slave is idle and ready for transaction
svalid	Indicates read data is transmitting
u_tx	UART transmitter data line

8.3. UART Module

UART is used in the bus bridge to have reliable communication between different clock domains. The module has transmitter and receiver ports configurable for different word lengths.

Functional Block Diagram



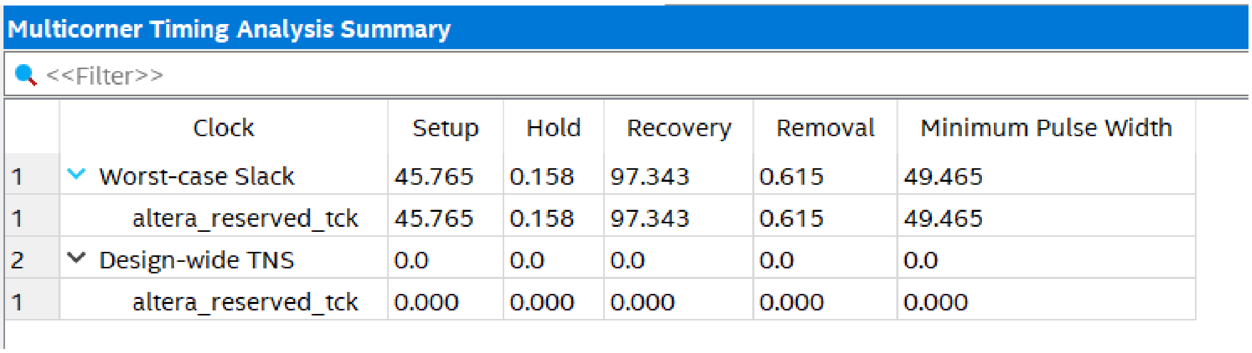
IO Description

Input Pin	Description
`clk`	Clock
`rstn`	Active low reset
`rx`	Receiver input data
`data_input[T..0]`	Data to be transmitted, configurable bit width of T
`data_en`	Signal to start transmitter (data_input is valid)

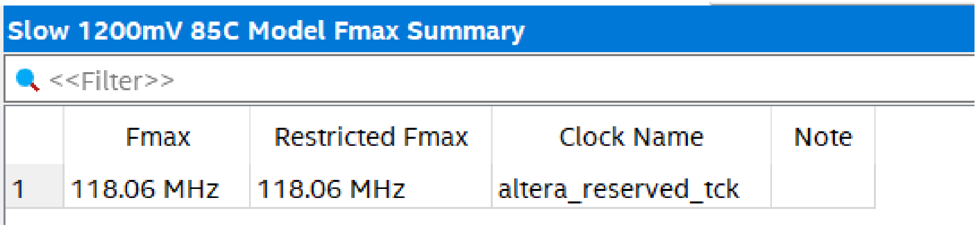
Output Pin	Description
`tx`	Bus request signal
`data_output[R..0]`	Received data from UART, configurable bit width of R
`ready`	Indicates receiver output is ready

tx_busy	Indicates transmitter is within a transaction

9. Timing Analysis Report

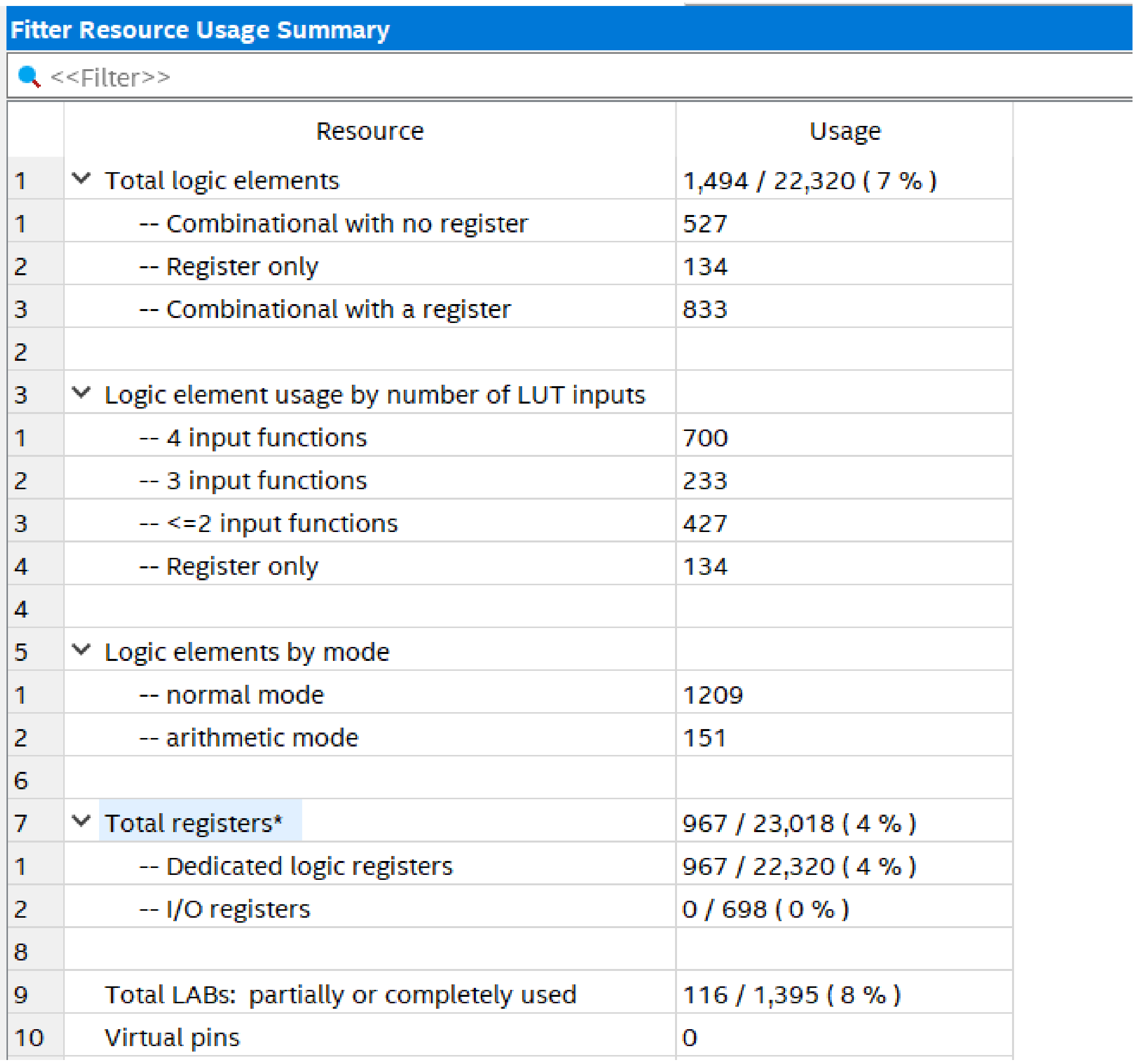

Multicorner Timing Analysis Summary

	Clock	Setup	Hold	Recovery	Removal	Minimum Pulse Width
1	Worst-case Slack	45.765	0.158	97.343	0.615	49.465
1	altera_reserved_tck	45.765	0.158	97.343	0.615	49.465
2	Design-wide TNS	0.0	0.0	0.0	0.0	0.0
1	altera_reserved_tck	0.000	0.000	0.000	0.000	0.000


Slow 1200mV 85C Model Fmax Summary

	Fmax	Restricted Fmax	Clock Name	Note
1	118.06 MHz	118.06 MHz	altera_reserved_tck	

10. Resource Utilization Report



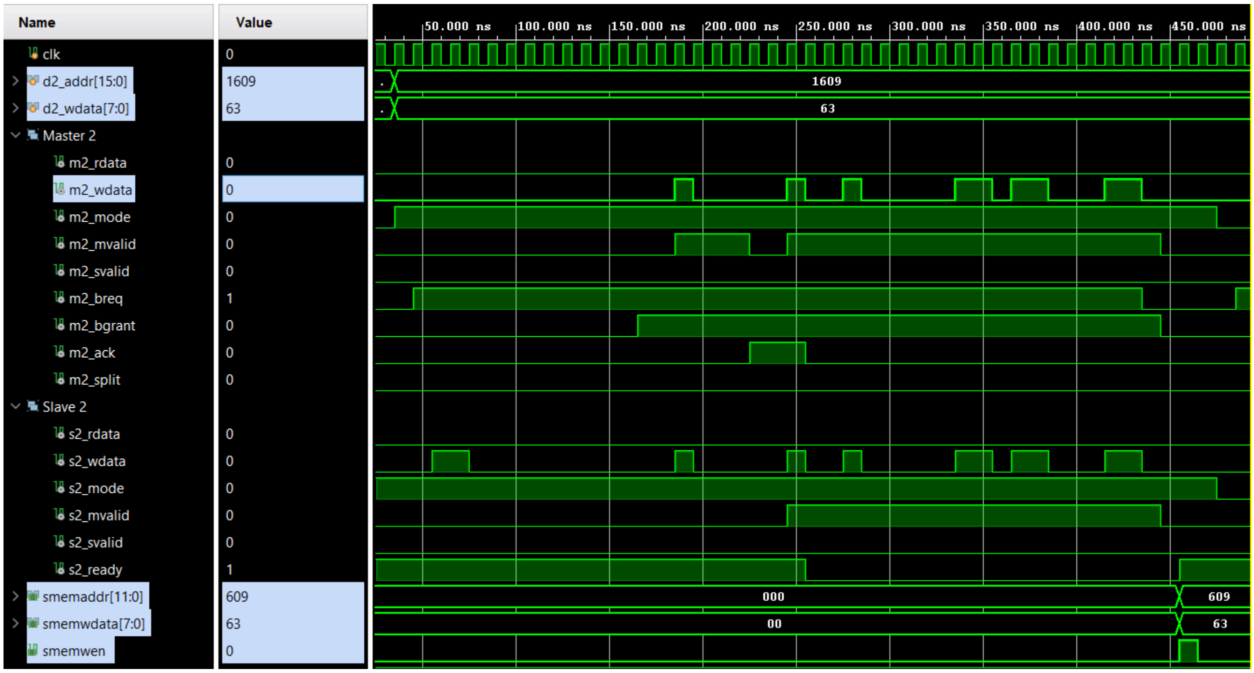
Fitter Resource Usage Summary

<<Filter>>

	Resource	Usage
1	Total logic elements	1,494 / 22,320 (7 %)
1	-- Combinational with no register	527
2	-- Register only	134
3	-- Combinational with a register	833
2		
3	Logic element usage by number of LUT inputs	
1	-- 4 input functions	700
2	-- 3 input functions	233
3	-- <=2 input functions	427
4	-- Register only	134
4		
5	Logic elements by mode	
1	-- normal mode	1209
2	-- arithmetic mode	151
6		
7	Total registers*	967 / 23,018 (4 %)
1	-- Dedicated logic registers	967 / 22,320 (4 %)
2	-- I/O registers	0 / 698 (0 %)
8		
9	Total LABs: partially or completely used	116 / 1,395 (8 %)
10	Virtual pins	0

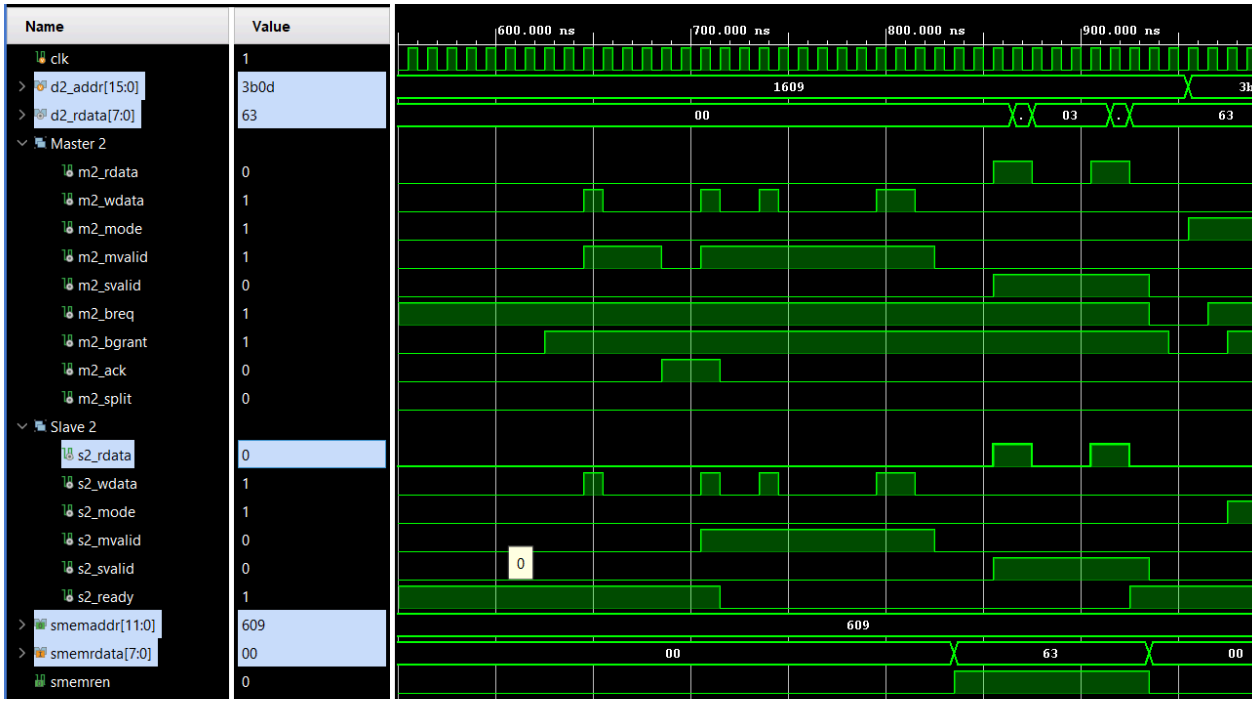
11. Simulation Results

11.1. Single master write transaction



Name	Value
clk	0
d2_addr[15:0]	1609
d2_wdata[7:0]	63
Master 2	
m2_rdata	0
m2_wdata	0
m2_mode	0
m2_mvalid	0
m2_svalid	0
m2_breq	1
m2_bgrant	0
m2_ack	0
m2_split	0
Slave 2	
s2_rdata	0
s2_wdata	0
s2_mode	0
s2_mvalid	0
s2_svalid	0
s2_ready	1
smemaddr[11:0]	609
smemwdata[7:0]	63
smemwen	0

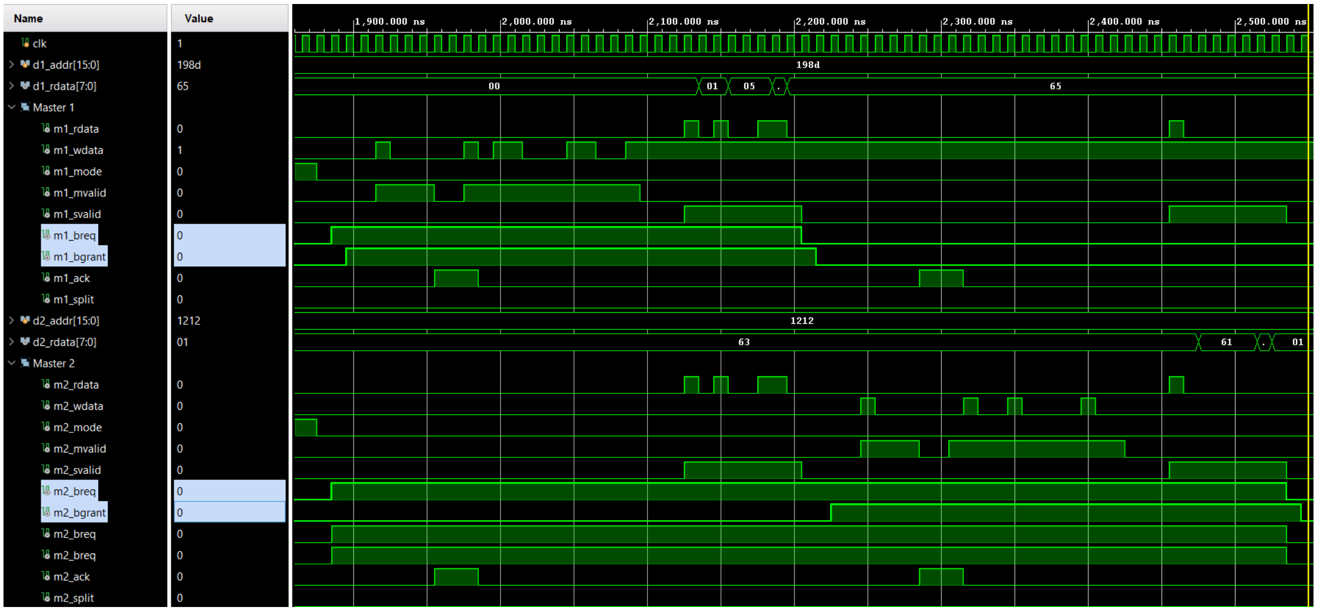
11.2. Single master read transaction



Name	Value
clk	1
d2_addr[15:0]	3b0d
d2_rdata[7:0]	63
Master 2	
m2_rdata	0
m2_wdata	1
m2_mode	1
m2_mvalid	1
m2_svalid	0
m2_breq	1
m2_bgrant	1
m2_ack	0
m2_split	0
Slave 2	
s2_rdata	0
s2_wdata	1
s2_mode	1
s2_mvalid	0
s2_svalid	0
s2_ready	1
smemaddr[11:0]	609
smemrdata[7:0]	00
smemren	0

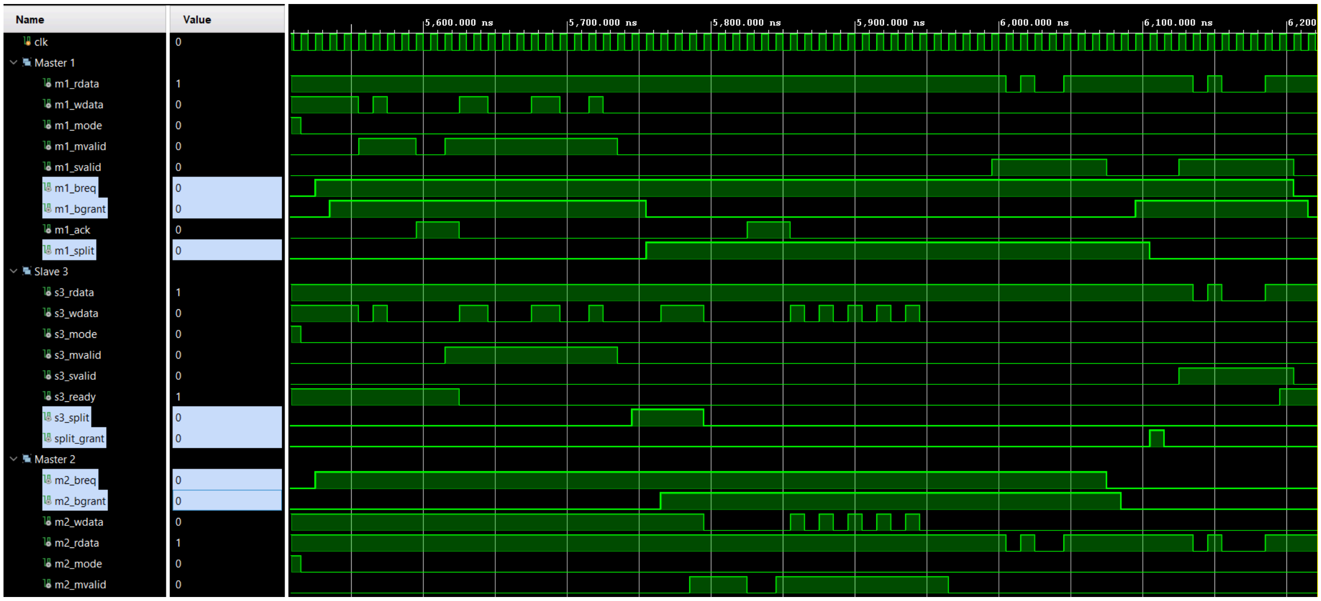
11.3. Both masters request at the same time

Master 1 and Master 2 both send read requests, Master 1 gets priority and Master 2 gets bus access after the Master 1 transaction is complete.



11.4. Split Transaction

Master 1 starts a read transaction to Slave 3, Slave 3 splits, Master 2 receives bus access until split is complete.



12. Appendix

12.1. Master Port

```
module master_port #(
        parameter ADDR_WIDTH = 16,
        parameter DATA_WIDTH = 8,
        parameter SLAVE_MEM_ADDR_WIDTH = 12
)(
        input clk, rstn,

        // Signals connecting to master device
        input [DATA_WIDTH-1:0] dwdata, // write data
        output [DATA_WIDTH-1:0] drdata,  // read data
        input [ADDR_WIDTH-1:0] daddr,
        input dvalid,                                 // ready valid interface
        output dready,
        input dmode,                           // 0 - read, 1 - write

        // Signals connecting to serial bus
        input mrdata,// read data
        output reg mwdata,  // write data and address
        output mmode,// 0 -  read, 1 - write
        output reg mvalid,  // wdata valid
        input svalid,// rdata valid

        // Signals to arbiter
        output mbreq,
        input mbgrant,
        input msplit,

        // Acknowledgement from address decoder
        input ack
);
        localparam SLAVE_DEVICE_ADDR_WIDTH = ADDR_WIDTH - SLAVE_MEM_ADDR_WIDTH; //
Part of address to identify slave
        localparam TIMEOUT_TIME = 5;

        /* Internal signals */

        // registers to accept data from master device and slave
        reg [DATA_WIDTH-1:0] wdata;
        reg [ADDR_WIDTH-1:0] addr;
        reg mode;
        reg [DATA_WIDTH-1:0] rdata;
```

```
       // counters
       reg [7:0] counter, timeout;

       // States
    localparam IDLE  = 3'b000,
               ADDR  = 3'b001,   // Send address to slave
               RDATA = 3'b010,    // Read data from slave
                       WDATA = 3'b011,  // Write data to slave
                       REQ        = 3'b100,    // Request bus access
                       SADDR = 3'b101,
                       WAIT  = 3'b110,  // Send slave device address
                       SPLIT = 3'b111;  // Wait for split slave to be ready

       // State variables
       reg [2:0] state, next_state;

       // Next state logic
       always @(*) begin
             case (state)
                    IDLE  : next_state = (dvalid) ? REQ : IDLE;
                    REQ      : next_state = (mbgrant) ? SADDR : REQ;
                    SADDR : next_state = (counter == SLAVE_DEVICE_ADDR_WIDTH-1) ?
WAIT : SADDR;
                    WAIT  : next_state = (ack) ? ADDR : ((timeout == TIMEOUT_TIME)
? IDLE : WAIT);
                    ADDR  : next_state = (counter == SLAVE_MEM_ADDR_WIDTH-1) ?
((mode) ? WDATA : RDATA) : ADDR;
                    RDATA : next_state = (msplit) ? SPLIT : ((svalid && (counter ==
DATA_WIDTH-1)) ? IDLE : RDATA);
                    WDATA : next_state = (counter == DATA_WIDTH-1) ? IDLE : WDATA;
                    SPLIT : next_state = (!msplit && mbgrant) ? RDATA : SPLIT;
                    default: next_state = IDLE;
             endcase
       end

       // State transition logic
       always @(posedge clk) begin
             state <= (!rstn) ? IDLE : next_state;
       end

       // Combinational output assignments
       assign dready = (state == IDLE);
       assign drdata = rdata;
       assign mmode = mode;
       assign mbreq = (state != IDLE);        // Keep bus request while master is
in need of the bus
```

```
    // Sequential output logic
    always @(posedge clk) begin
        if (!rstn) begin
            wdata <= 'b0;
            rdata <= 'b0;
            addr <= 'b0;
            mode <= 0;
            counter <= 'b0;
            mvalid <= 0;
            mwdata <= 0;
            timeout <= 'b0;
        end
        else begin
            case (state)
                IDLE : begin
                    counter <= 'b0;
                    mvalid <= 0;
                    timeout <= 'b0;

                    if (dvalid) begin   // Have to send data
                        wdata <= dwdata;
                        addr <= daddr;
                        mode <= dmode;
                    end else begin
                        wdata <= wdata;
                        addr <= addr;
                        mode <= mode;
                    end
                end

                REQ : begin

                end

                SADDR : begin// Send slave device address
                    mwdata <= addr[SLAVE_MEM_ADDR_WIDTH + counter];
                    mvalid <= 1'b1;

                    if (counter == SLAVE_DEVICE_ADDR_WIDTH-1) begin
                        counter <= 'b0;
                    end else begin
                        counter <= counter + 1;
                    end
                end

                WAIT : begin
                    mvalid <= 1'b0;
                    timeout <= timeout + 1;
                end

                ADDR : begin // Send slave mem address
                    mwdata <= addr[counter];
                    mvalid <= 1'b1;

                    if (counter == SLAVE_MEM_ADDR_WIDTH-1) begin
                        counter <= 'b0;
                    end else begin
                        counter <= counter + 1;
                    end
                end

                RDATA : begin// Receive data from slave
                    mvalid <= 1'b0;
                    if (svalid) begin
                        rdata[counter] <= mrdata;

                        if (counter == DATA_WIDTH-1) begin
                            counter <= 'b0;
                        end else begin
                            counter <= counter + 1;
                        end

                    end else begin
                        rdata <= rdata;
                        counter <= counter;
                    end
                end

                WDATA : begin// Send data to slave
                    mwdata <= wdata[counter];
                    mvalid <= 1'b1;

                    if (counter == DATA_WIDTH-1) begin
                        counter <= 'b0;
                    end else begin
                        counter <= counter + 1;
                    end
                end

                SPLIT : begin
                    mvalid <= 1'b0;
                end

                default: begin
```

```
                    wdata <= wdata;
                    rdata <= rdata;
                    addr <= addr;
                    mode <= mode;
                    counter <= counter;
                    mvalid <= mvalid;
                    mwdata <= mwdata;
                    timeout <= timeout;
                end
            endcase
        end
    end


endmodule
```

12.2. Slave Port

```
module slave_port #(
    parameter ADDR_WIDTH = 12, DATA_WIDTH = 8, SPLIT_EN = 0)
(
    input clk, rstn,

    // Signals connecting to slave memory
    input [DATA_WIDTH-1:0] smemrdata, // data read from the slave memory
    input rvalid,
    output reg smemwen, smemren,
    output reg [ADDR_WIDTH-1:0] smemaddr, //input address of slave
    output reg [DATA_WIDTH-1:0] smemwdata, // data written to the slave memory

    // Signals connecting to serial bus
    input swdata,// write data and address from master
    output reg srdata,  // read data to the master
    input smode, // 0 -  read, 1 - write, from master
    input mvalid,// wdata valid - (recieving data and address from master)
    input split_grant, //grant to send read data
    output reg svalid,  // rdata valid - (sending data from slave)
    output sready, //slave is ready for transaction
    output ssplit // 1 - split
);

    /* Internal signals */

    // registers to accept data from master and slave memory
    reg [DATA_WIDTH-1:0] wdata;  //write data from master
    reg [ADDR_WIDTH-1:0] addr;
```

```
       wire [DATA_WIDTH-1:0] rdata;     //read data from slave memory
       reg mode;
       // counters
       reg [7:0] counter;

       localparam LATENCY = 4;
       reg [LATENCY-1:0] rcounter;

       // States
    localparam IDLE  = 3'b000,    //0
               ADDR  = 3'b001,   // Receive address from slave //1
               RDATA = 3'b010,    // Send data to master //2
                       WDATA = 3'b011,  // Receive data from master //3
                       SREADY = 3'b101, //5
                       SPLIT = 3'b100, // 4
                       WAIT = 3'b110, //6
                       RVALID = 3'b111; //7


       // State variables
       reg [2:0] state, next_state;

       // Next state logic
       always @(*) begin
             case (state)
                    IDLE   : next_state = (mvalid) ? ADDR : IDLE;
                    ADDR   : next_state = (counter == ADDR_WIDTH-1) ? ((mode) ?
WDATA : SREADY) : ADDR;
                    SREADY : next_state = (mode) ? IDLE : ((SPLIT_EN) ? SPLIT :
RVALID);
                    RVALID : next_state = (rvalid) ? RDATA : RVALID;
                    SPLIT : next_state = (rcounter == LATENCY) ? WAIT : SPLIT;
                    WAIT : next_state = (split_grant) ? RDATA : WAIT;
                    RDATA : next_state = (counter == DATA_WIDTH-1) ? IDLE : RDATA;
                    WDATA : next_state = (counter == DATA_WIDTH-1) ? SREADY :
WDATA;
                    default: next_state = IDLE;
             endcase
       end

       // State transition logic
       always @(posedge clk) begin
             state <= (!rstn) ? IDLE : next_state;
       end

       // Combinational output assignments
       assign rdata =      smemrdata;
       assign sready = (state == IDLE);
       assign ssplit = (state == SPLIT);

       // Sequential output logic
       always @(posedge clk) begin
             if (!rstn) begin
                    wdata <= 'b0;
                    addr <= 'b0;
                    counter <= 'b0;
                    svalid <= 0;
                    smemren <= 0;
                    smemwen <= 0;
                    mode <= 0;
                    smemaddr <= 0;
                    smemwdata <= 0;
                    srdata <= 0;
                    rcounter <= 'b0;
             end
             else begin
                    case (state)

                        IDLE : begin
                            counter <= 'b0;
                            svalid <= 0;
                            smemren <= 0;
                            smemwen <= 0;

                            if (mvalid) begin
                                mode <= smode;
                                addr[counter] <= swdata;
                                counter <= counter + 1;

                            end else begin
                                addr <= addr;
                                counter <= counter;
                                mode <= mode;
                            end


                        end

                        ADDR : begin
                            svalid <= 1'b0;
                            if (mvalid) begin
                                addr[counter] <= swdata;

                                if (counter == ADDR_WIDTH-1) begin
```

```
                                                counter <= 'b0;
                                        end else begin
                                                counter <= counter + 1;
                                        end

                                end else begin
                                        addr <= addr;
                                        counter <= counter;
                                end

                        end

                        SREADY: begin

                                svalid <= 1'b0;
                                if (mode) begin
                                        smemwen <= 1'b1;
                                        smemwdata <= wdata;
                                        smemaddr <= addr;
                                end else begin
                                        smemren <= 1'b1;

                                        smemaddr <= addr;
                                end
                        end

                        RVALID: begin
                                //waiting
                        end

                        SPLIT : begin //wait for sometime
                                rcounter <= rcounter + 1;
                        end

                        WAIT : begin //wait until grant bus access for split
transfer

                                rcounter <= 'b0;
                        end

                        RDATA : begin// Send data to master
                                srdata <= rdata[counter];
                                svalid <= 1'b1;

                                if (counter == DATA_WIDTH-1) begin
                                        counter <= 'b0;
                                end else begin
                                        counter <= counter + 1;
```

```
                    end

                end

                WDATA : begin// Receive data from master
                    svalid <= 1'b0;
                    if (mvalid) begin
                        wdata[counter] <= swdata;

                        if (counter == DATA_WIDTH-1) begin
//                          smemwen <= 1'b1;
                            counter <= 'b0;
                        end else begin
                            counter <= counter + 1;
                        end
                    end else begin
                        wdata <= wdata;
                        counter <= counter;

                    end
                end

                default: begin
                    wdata <= wdata;
                    addr <= addr;
                    counter <= counter;
                    svalid <= svalid;
                    smemwen <= smemwen;
                    smemren <= smemren;
                    rcounter <= rcounter;
                end

            endcase
        end
    end


endmodule
```

12.3. Slave with port and memory

```
module slave_memory_bram #(
    parameter ADDR_WIDTH = 12, DATA_WIDTH = 8, MEM_SIZE = 4096)
(
    input clk, rstn, wen, ren,
    input [ADDR_WIDTH-1:0] addr, //input address of slave
    input [DATA_WIDTH-1:0] wdata, // data to be written in the slave

    output [DATA_WIDTH-1:0] rdata, // data to be read from the slave
    output  reg rvalid
);

    localparam MEM_ADDR_WIDTH = $clog2(MEM_SIZE);
    reg ren_prev;
    generate
        if (MEM_SIZE == 4096) begin
            slave_bram memory (
                .address(addr[MEM_ADDR_WIDTH-1:0]),
                .clock(clk),
                .data(wdata),
                .rden(ren),
                .wren(wen),
                .q(rdata)
            );
        end else begin
            slave_bram_2k memory (
                .address(addr[MEM_ADDR_WIDTH-1:0]),
                .clock(clk),
                .data(wdata),
                .rden(ren),
                .wren(wen),
                .q(rdata)
            );
        end
    endgenerate

    always @(posedge clk) begin
        if (!rstn) begin
            rvalid <= 0;
            ren_prev <= 0;
        end
        else begin
            if ((!ren_prev) && ren) begin
                rvalid <= 0;
            end   else if (ren) begin
                rvalid <= 1;
```

```
                  end else begin
                        rvalid <= 0;
                  end
                  ren_prev <= ren;
            end
      end
endmodule

module slave #(parameter ADDR_WIDTH = 12, DATA_WIDTH = 8, SPLIT_EN = 0,
MEM_SIZE = 4096)
(
    input clk, rstn,
    // Signals connecting to serial bus
      input swdata,     // write data and address from master
      output srdata,    // read data to the master
      input smode,      // 0 -  read, 1 - write, from master
      input mvalid,     // wdata valid - (recieving data and address from
master)
    input split_grant, // grant bus access in split
      output svalid,    // rdata valid - (sending data from slave)
    output sready, //slave is ready for transaction
    output ssplit
);

      wire [DATA_WIDTH-1:0] smemrdata;
      wire smemwen;
    wire smemren;
      wire [ADDR_WIDTH-1:0] smemaddr;
      wire [DATA_WIDTH-1:0] smemwdata;
    wire rvalid;

    slave_port #(
        .ADDR_WIDTH(ADDR_WIDTH),
        .DATA_WIDTH(DATA_WIDTH),
        .SPLIT_EN(SPLIT_EN)
    )sp(
        .clk(clk),
        .rstn(rstn),
        .smemrdata(smemrdata),
        .rvalid(rvalid),
        .smemwen(smemwen),
        .smemren(smemren),
        .smemaddr(smemaddr),
        .smemwdata(smemwdata),
        .swdata(swdata),
        .srdata(srdata),
        .smode(smode),
        .mvalid(mvalid),
        .split_grant(split_grant),
        .svalid(svalid),
        .sready(sready),
        .ssplit(ssplit)
    );

    slave_memory #(
        .ADDR_WIDTH(ADDR_WIDTH),
        .DATA_WIDTH(DATA_WIDTH),
        .MEM_SIZE(MEM_SIZE)
    )sm(
        .clk(clk),
        .rstn(rstn),
        .wen(smemwen),
        .ren(smemren),
        .addr(smemaddr),
        .wdata(smemwdata),
        .rdata(smemrdata),
        .rvalid(rvalid)
    );

endmodule
```

12.4. Arbiter

```
module arbiter
(
      input clk, rstn,
      input breq1, breq2,  //bus requests from 2 masters
      input sready1, sready2, sreadysp,   //slave ready, sreadysp = split
supported slave
      input ssplit,             // slave split

      output bgrant1, bgrant2,  //bus grant signals for 2 masters
      output msel, //master select; 0 - master 1, 1 - master 2
      output reg msplit1, msplit2,            // Split signals given to master
      output reg split_grant                  // grant access to continue split
transaction (send back to slave)
);

      //priority based: high priority for master 1 - breq1

      wire sready, sready_nsplit;
      reg [1:0] split_owner;

      assign sready = sready1 & sready2 & sreadysp;
      assign sready_nsplit = sready1 & sready2;           // not split slaves are
ready

      // Split owner encoding
      localparam NONE = 2'b00,
                      SM1 = 2'b01,
                      SM2 = 2'b10;

      // States
   localparam IDLE  = 3'b000,    //0
              M1  = 3'b001,     // M1 uses bus//1
                      M2 = 3'b010;     // M2 uses bu3 //3

      // State variables
      reg [2:0] state, next_state;

      // Next state logic
      always @(*) begin
            case (state)
                   IDLE  : begin
                         if (!ssplit) begin  // either split was released or no
split was there
                                 if (split_owner == SM1) next_state = M1;
                                 else if (breq1 & sready) next_state = M1;
                                 else if (split_owner == SM2) next_state = M2;
                                 else if (breq2 & sready) next_state = M2;
                                 else next_state = IDLE;
                         end
                         else begin          // One master is waiting for a split
transaction, other master can continue
                                 if ((split_owner == SM1) && breq2 &&
sready_nsplit) next_state = M2;
                                 else if ((split_owner == SM2) && breq1 &&
sready_nsplit) next_state = M1;
                                 else next_state = IDLE;
                         end
                   end
                   M1  : next_state = (!breq1 | (split_owner == NONE && ssplit)) ?
```

```
IDLE : M1;
            M2 : next_state = (!breq2 | (split_owner == NONE && ssplit)) ?
IDLE : M2;
            default: next_state = IDLE;
        endcase
    end

    // State transition logic
    always @(posedge clk) begin
        state <= (!rstn) ? IDLE : next_state;
    end

    // Combinational output assignments
    assign bgrant1 = (state == M1);
    assign bgrant2 = (state == M2);
    assign msel = (state == M2);

    // Sequential output assignments (for split)
    always @(posedge clk) begin
        if (!rstn) begin
            msplit1 <= 1'b0;
            msplit2 <= 1'b0;
            split_owner <= NONE;
            split_grant <= 1'b0;
        end
        else begin
            case (state)

                M1 : begin
                    if (split_owner == NONE && ssplit) begin
                        msplit1 <= 1'b1;
                        split_owner <= SM1;
                        split_grant <= 1'b0;
                    end else if (split_owner == SM1 && !ssplit) begin
                        msplit1 <= 1'b0;
                        split_owner <= NONE;
                        split_grant <= 1'b1;
                    end else begin
                        msplit1 <= msplit1;
                        split_owner <= split_owner;
                        split_grant <= 1'b0;
                    end
                end

                M2 : begin
                    if (split_owner == NONE && ssplit) begin
                        msplit2 <= 1'b1;
                        split_owner <= SM2;
                        split_grant <= 1'b0;
                    end else if (split_owner == SM2 && !ssplit) begin
                        msplit2 <= 1'b0;
                        split_owner <= NONE;
                        split_grant <= 1'b1;
                    end else begin
                        msplit2 <= msplit2;
                        split_owner <= split_owner;
                        split_grant <= 1'b0;
                    end
                end

                default : begin
                    msplit1 <= msplit1;
                    msplit2 <= msplit2;
                    split_owner <= split_owner;
                    split_grant <= split_grant;
                end
            endcase
        end
    end

endmodule
```

12.5. Address Decoder

```
module addr_decoder #(
    parameter ADDR_WIDTH = 16,
    parameter DEVICE_ADDR_WIDTH = 4
) (
    input clk, rstn,

    input mwdata,       // write data bus
    input mvalid,            // valid from master
    input ssplit,       // split signal from slave
    input split_grant,      // signal from arbiter ending split

    // ready signals from slave
    input sready1, sready2, sready3,

    // valid signals going to slaves
    output mvalid1, mvalid2, mvalid3,

    output reg [1:0] ssel,       // Slave select going to muxes
    output ack              // Acknowledgement going back to master
);

    // Internal signals
    reg [DEVICE_ADDR_WIDTH-1:0] slave_addr;
    reg slave_en;       // Enable slave connection
    wire mvalid_out;
    wire slave_addr_valid;  // Valid slave address
    wire [2:0] sready;
    reg [3:0] counter;
    reg [DEVICE_ADDR_WIDTH-1:0] split_slave_addr;

    // To give the correct wen signals
    dec3 mvalid_decoder (
        .sel(ssel),
        .en(mvalid_out),
        .out1(mvalid1),
        .out2(mvalid2),
        .out3(mvalid3)
    );

    // States
    localparam IDLE  = 2'b00,
               ADDR  = 2'b01,    // Receive address from master
               CONNECT = 2'b10,  // Enable correct slave connection
               WAIT = 2'b11;

    // State variables
      reg [1:0] state, next_state;

    // Next state logic
      always @(*) begin
             case (state)
                    IDLE    : next_state = (mvalid) ? ADDR : ((split_grant) ? WAIT
: IDLE);
                    ADDR    : next_state = (counter == DEVICE_ADDR_WIDTH-1) ?
CONNECT : ADDR;
                    CONNECT : next_state = (slave_addr_valid) ? ((mvalid) ? WAIT :
CONNECT) : IDLE;
            WAIT    : next_state = (sready[slave_addr] | ssplit) ? IDLE : WAIT;
                    default: next_state = IDLE;
             endcase
      end

    // State transition logic
      always @(posedge clk) begin
             state <= (!rstn) ? IDLE : next_state;
      end
    // Combinational assignments
    assign mvalid_out = mvalid & slave_en;
    assign slave_addr_valid = (slave_addr < 3) & sready[slave_addr];    // check
whether ready and valid
    assign ack = (state == CONNECT) & slave_addr_valid;     // If address invalid,
do not ack
    assign sready = {sready3, sready2, sready1};

    // Sequential output logic
      always @(posedge clk) begin
             if (!rstn) begin
                    slave_addr <= 'b0;
            slave_en <= 0;
            counter <= 'b0;
            ssel <= 'b0;
            split_slave_addr <= 'b0;
             end
             else begin
                    case (state)
                           IDLE : begin
                                  slave_en <= 0;

                                  if (mvalid) begin   // Have to send data
                        slave_addr[0] <= mwdata;
                        counter <= 1;
                    end else if (split_grant) begin
                        slave_addr <= split_slave_addr;
                        counter <= 'b0;
                    end else begin
                        slave_addr <= slave_addr;
                        counter <= 'b0;
                    end
                           end

                           ADDR : begin // Send slave mem address
                                  slave_addr[counter] <= mwdata;

                                  if (counter == DEVICE_ADDR_WIDTH-1) begin
                                         counter <= 'b0;
                                  end else begin
                                         counter <= counter + 1;
                                  end
                           end

                           CONNECT : begin     // Receive data from slave
                    slave_en <= 1;
                    ssel <= slave_addr[1:0];
                          end

                WAIT : begin
                    slave_en <= 1;
                    ssel <= slave_addr[1:0];

                    if (ssplit)
                        split_slave_addr <= slave_addr;
                    else begin
                        split_slave_addr <= split_slave_addr;
                    end
                end

                          default: begin
                                slave_addr <= slave_addr;
                    slave_en <= slave_en;
                    counter <= counter;
                    ssel <= ssel;
                    split_slave_addr <= split_slave_addr;
                          end
                    endcase
             end
       end

endmodule
```

12.6. Bus Structure

```
module bus_m2_s3 #(
    parameter ADDR_WIDTH = 16,
    parameter DATA_WIDTH = 8,
    parameter SLAVE_MEM_ADDR_WIDTH = 12
) (
    input clk, rstn,

    // Master 1
    output        m1_rdata,      // read data
       input         m1_wdata,   // write data and address
       input         m1_mode,    // 0 -  read, 1 - write
       input         m1_mvalid,  // wdata valid
       output        m1_svalid,  // rdata valid
       input         m1_breq,
       output        m1_bgrant,
    output        m1_ack,
    output        m1_split,
    // Master 2
    output        m2_rdata,      // read data
      input         m2_wdata,    // write data and address
      input         m2_mode,     // 0 -  read, 1 - write
      input         m2_mvalid,   // wdata valid
      output        m2_svalid,   // rdata valid
      input         m2_breq,
      output        m2_bgrant,
    output        m2_ack,
    output        m2_split,

    // Slave 1
    input         s1_rdata,      // read data
      output        s1_wdata,    // write data and address
      output        s1_mode,     // 0 -  read, 1 - write
      output        s1_mvalid,   // wdata valid
      input         s1_svalid,   // rdata valid
    input         s1_ready,

    // Slave 2
    input         s2_rdata,      // read data
      output        s2_wdata,    // write data and address
      output        s2_mode,     // 0 -  read, 1 - write
      output        s2_mvalid,   // wdata valid
      input         s2_svalid,   // rdata valid
    input         s2_ready,

    // Slave 3
    input         s3_rdata,      // read data
      output        s3_wdata,    // write data and address
      output        s3_mode,     // 0 -  read, 1 - write
      output        s3_mvalid,   // wdata valid
      input         s3_svalid,   // rdata valid
    input         s3_ready,
    input         s3_split,      // s3 is the split slave

    output        split_grant
);
    localparam DEVICE_ADDR_WIDTH = ADDR_WIDTH - SLAVE_MEM_ADDR_WIDTH;

    // Internal signals
    wire m_select;      // master select for control
    wire m_wdata, m_mode, m_mvalid;       // master muxed signals
    wire [1:0] s_select;      // Slave select for read mux
    wire m_ack;         // Acknowledgement from addr decoder
    wire s_rdata, s_svalid, s_split;
```

```
// Instantiate modules in bus

// Bus arbiter
arbiter bus_arbiter (
    .clk(clk),
    .rstn(rstn),
    .breq1(m1_breq),
    .breq2(m2_breq),
    .bgrant1(m1_bgrant),
    .bgrant2(m2_bgrant),
    .msel(m_select),
    .sready1(s1_ready),
    .sready2(s2_ready),
    .sreadysp(s3_ready),
    .ssplit(s_split),
    .msplit1(m1_split),
    .msplit2(m2_split),
    .split_grant(split_grant)
);

// Address decoder
addr_decoder #(
    .ADDR_WIDTH(ADDR_WIDTH),
    .DEVICE_ADDR_WIDTH(DEVICE_ADDR_WIDTH)
) decoder (
    .clk(clk),
    .rstn(rstn),
    .mwdata(m_wdata),
    .mvalid(m_mvalid),
    .mvalid1(s1_mvalid),
    .mvalid2(s2_mvalid),
    .mvalid3(s3_mvalid),
    .sready1(s1_ready),
    .sready2(s2_ready),
    .sready3(s3_ready),
    .ssel(s_select),
    .ack(m_ack),
    .ssplit(s_split),
    .split_grant(split_grant)
);

// Write data mux
mux2 #(.DATA_WIDTH(1)) wdata_mux (
    .dsel(m_select),
    .d0(m1_wdata),
    .d1(m2_wdata),
        .dout(m_wdata)
    );

    // Master control muxes
    mux2 #(.DATA_WIDTH(2)) mctrl_mux (
        .dsel(m_select),
        .d0({m1_mode, m1_mvalid}),
        .d1({m2_mode, m2_mvalid}),
        .dout({m_mode, m_mvalid})
    );

    // Read data mux
    mux3 #(.DATA_WIDTH(1)) rdata_mux (
        .dsel(s_select),
        .d0(s1_rdata),
        .d1(s2_rdata),
        .d2(s3_rdata),
        .dout(s_rdata)
    );

    // Read control mux
    mux3 #(.DATA_WIDTH(1)) rctrl_mux (
        .dsel(s_select),
        .d0(s1_svalid),
        .d1(s2_svalid),
        .d2(s3_svalid),
        .dout(s_svalid)
    );

    // Assignments
    assign m1_rdata = s_rdata;
    assign m1_svalid = s_svalid;

    assign m2_rdata = s_rdata;
    assign m2_svalid = s_svalid;

    assign s1_wdata = m_wdata;
    assign s1_mode = m_mode;

    assign s2_wdata = m_wdata;
    assign s2_mode = m_mode;

    assign s3_wdata = m_wdata;
    assign s3_mode = m_mode;

    assign m1_ack = m_ack;
    assign m2_ack = m_ack;
    assign s_split = s3_split;

endmodule
```

12.7. Bus Bridge Master

```
/*
    Data from UART should be sent as
    {mode, data, addr}
*/

module bus_bridge_master #(
      parameter ADDR_WIDTH = 16,
      parameter DATA_WIDTH = 8,
      parameter SLAVE_MEM_ADDR_WIDTH = 12,
    parameter BB_ADDR_WIDTH = 12,
    parameter UART_CLOCKS_PER_PULSE = 5208
)(
      input clk, rstn,

      // Signals connecting to serial bus
      input mrdata,// read data
      output mwdata,      // write data and address
      output mmode,// 0 -   read, 1 - write
      output mvalid,      // wdata valid
      input svalid,// rdata valid

      // Signals to arbiter
      output mbreq,
      input mbgrant,
      input msplit,

      // Acknowledgement from address decoder
      input ack,

    // Bus bridge UART signals
    output u_tx,
    input u_rx
);
    localparam UART_RX_DATA_WIDTH = DATA_WIDTH + BB_ADDR_WIDTH + 1;    // Receive
all 3 info
    localparam UART_TX_DATA_WIDTH = DATA_WIDTH;     // Transmit only read data

      // Signals connecting to master port
      reg [DATA_WIDTH-1:0] dwdata; // write data
      wire [DATA_WIDTH-1:0] drdata;    // read data
      wire [ADDR_WIDTH-1:0] daddr;
      reg dvalid;                            // ready valid interface
      wire dready;
      reg dmode;                             // 0 - read, 1 - write

    // Signals connecting to FIFO
    reg fifo_enq;
    reg fifo_deq;
    reg [UART_RX_DATA_WIDTH-1:0] fifo_din;
    wire [UART_RX_DATA_WIDTH-1:0] fifo_dout;
    wire fifo_empty;

    // Signals connecting to UART
    reg [UART_TX_DATA_WIDTH-1:0] u_din;
    reg u_en;
    wire u_tx_busy;
    wire u_rx_ready;
    wire [UART_RX_DATA_WIDTH-1:0] u_dout;

    reg [BB_ADDR_WIDTH-1:0] bb_addr;
    reg expect_rdata;
    reg prev_u_ready, prev_m_ready;

    // Instantiate modules

    // Master port
    master_port #(
        .ADDR_WIDTH(ADDR_WIDTH),
        .DATA_WIDTH(DATA_WIDTH),
        .SLAVE_MEM_ADDR_WIDTH(SLAVE_MEM_ADDR_WIDTH)
    ) master (
        .clk(clk),
        .rstn(rstn),
        .dwdata(dwdata),
        .drdata(drdata),
        .daddr(daddr),
        .dvalid(dvalid),
        .dready(dready),
        .dmode(dmode),
        .mrdata(mrdata),
        .mwdata(mwdata),
        .mmode(mmode),
        .mvalid(mvalid),
        .svalid(svalid),
        .mbreq(mbreq),
        .mbgrant(mbgrant),
        .msplit(msplit),
        .ack(ack)
    );

    // FIFO module
    fifo #(
        .DATA_WIDTH(UART_RX_DATA_WIDTH),
        .DEPTH(8)
    ) fifo_queue (
        .clk(clk),
        .rstn(rstn),
        .enq(fifo_enq),
        .deq(fifo_deq),
        .data_in(fifo_din),
        .data_out(fifo_dout),
        .empty(fifo_empty)
    );

    // UART module
    uart #(
        .CLOCKS_PER_PULSE(UART_CLOCKS_PER_PULSE),
        .TX_DATA_WIDTH(UART_TX_DATA_WIDTH),
        .RX_DATA_WIDTH(UART_RX_DATA_WIDTH)
    ) uart_module (
        .data_input(u_din),
        .data_en(u_en),
        .clk(clk),
        .rstn(rstn),
        .tx(u_tx),  // Transmitter output (tx)
        .tx_busy(u_tx_busy),
        .rx(u_rx),
        .ready(u_rx_ready),
        .data_output(u_dout)
    );

    // Address converter
    addr_convert #(
        .BB_ADDR_WIDTH(BB_ADDR_WIDTH),
        .BUS_ADDR_WIDTH(ADDR_WIDTH),
        .BUS_MEM_ADDR_WIDTH(SLAVE_MEM_ADDR_WIDTH)
    ) addr_convert_module (
        .bb_addr(bb_addr),
        .bus_addr(daddr)
    );

    // Send UART received data to FIFO
    always @(posedge clk) begin
```

```
        if (!rstn) begin
            fifo_din <= 'b0;
            fifo_enq <= 1'b0;
            prev_u_ready <= 1'b0;
        end
        else begin
            prev_u_ready <= u_rx_ready;

            if (u_rx_ready && !prev_u_ready) begin
                fifo_din <= u_dout;
                fifo_enq <= 1'b1;
            end
            else begin
                fifo_din <= fifo_din;
                fifo_enq <= 1'b0;
            end
        end
    end

    // Send FIFO data to master port
    always @(posedge clk) begin
        if (!rstn) begin
            bb_addr <= 'b0;
            dwdata <= 'b0;
            dmode <= 1'b0;
            dvalid <= 1'b0;
            fifo_deq <= 1'b0;
            expect_rdata <= 1'b0;
        end
        else begin
            if (dready & !fifo_empty & !dvalid) begin
                bb_addr <= fifo_dout[BB_ADDR_WIDTH-1:0];
                dwdata <= fifo_dout[BB_ADDR_WIDTH+:DATA_WIDTH];
                dmode <= fifo_dout[BB_ADDR_WIDTH + DATA_WIDTH];
                dvalid <= 1'b1;
                fifo_deq <= 1'b1;
                expect_rdata <= !(fifo_dout[BB_ADDR_WIDTH + DATA_WIDTH]);
            end
            else begin
                bb_addr <= bb_addr;
                dwdata <= dwdata;
                dmode <= dmode;
                dvalid <= 1'b0;
                fifo_deq <= 1'b0;
                expect_rdata <= expect_rdata;
            end
        end
    end

    // Send bus read data to UART TX
    always @(posedge clk) begin
        if (!rstn) begin
            u_din <= 'b0;
            u_en <= 1'b0;
            prev_m_ready <= 1'b0;
        end
        else begin
            prev_m_ready <= dready;
            // Read request finished
            if (!prev_m_ready & dready & expect_rdata) begin
                u_din <= drdata;
                u_en <= 1'b1;
            end
            else begin
                u_din <= u_din;
                u_en <= 1'b0;
            end
        end
    end

endmodule
```

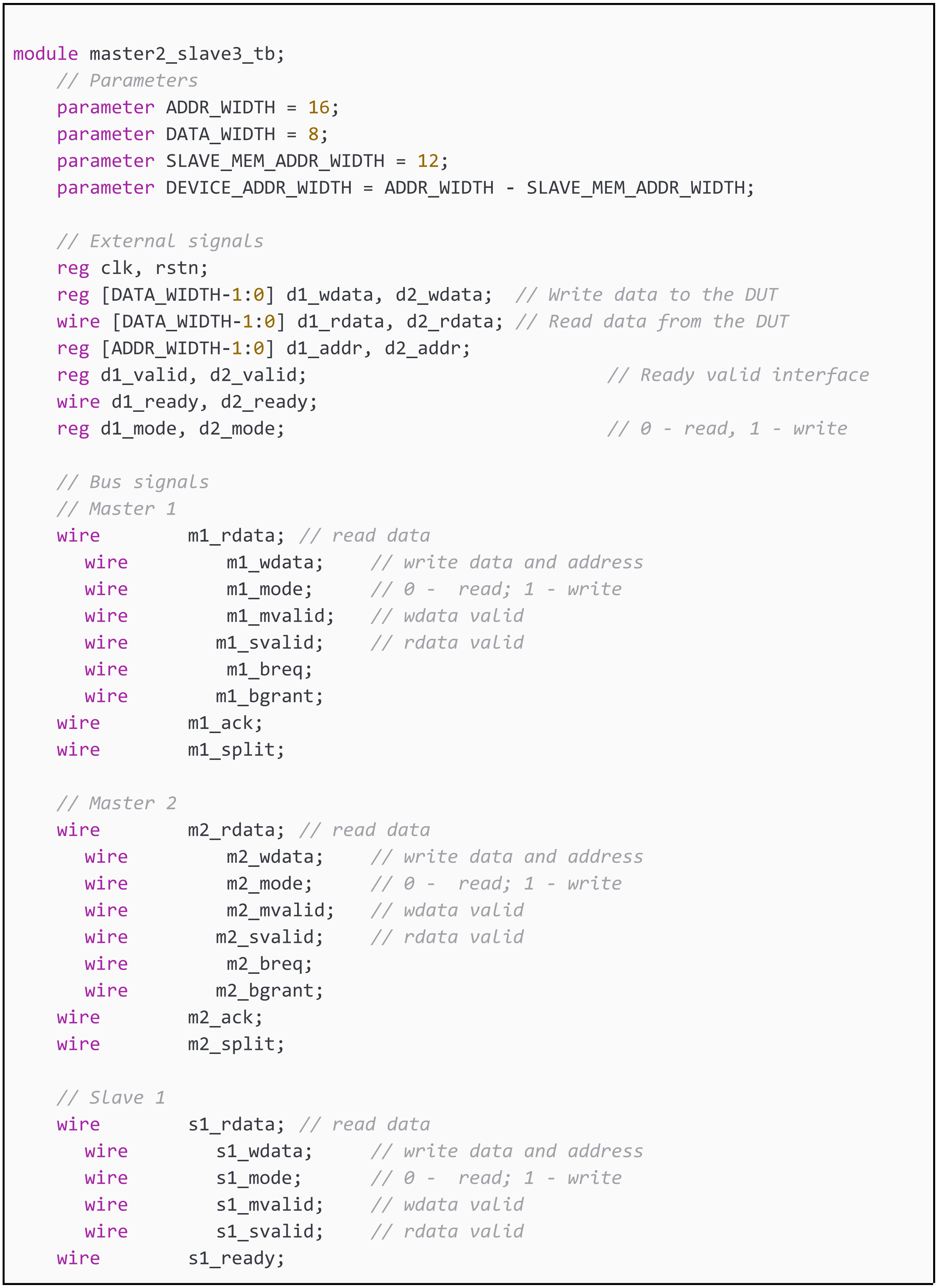
12.8. Bus Bridge Slave

```
module bus_bridge_slave #(
       parameter DATA_WIDTH = 8,
       parameter ADDR_WIDTH = 12,
    parameter UART_CLOCKS_PER_PULSE = 5208
)(
    input clk, rstn,
    // Signals connecting to serial bus
       input swdata,// write data and address from master
       input smode, // 0 -  read, 1 - write, from master
       input mvalid,// wdata valid - (recieving data and address from master)
    input split_grant, // grant bus access in split

    output srdata,  // read data to the master
       output svalid,      // rdata valid - (sending data from slave)
    output sready, //slave is ready for transaction
    output ssplit,

    // Bus bridge UART signals
    output u_tx,
    input u_rx
);
    localparam UART_TX_DATA_WIDTH = DATA_WIDTH + ADDR_WIDTH + 1;    // Transmit all
3 info
    localparam UART_RX_DATA_WIDTH = DATA_WIDTH;     // Receive only read data
    localparam SPLIT_EN = 1'b0;

    // Slave port ready
    wire spready;

      // Signals connecting to slave port
      wire [DATA_WIDTH-1:0] smemrdata;
      wire smemwen;
    wire smemren;
      wire [ADDR_WIDTH-1:0] smemaddr;
      wire [DATA_WIDTH-1:0] smemwdata;
    wire rvalid;

    // Signals connecting to UART
    reg [UART_TX_DATA_WIDTH-1:0] u_din;
    reg u_en;
    wire u_tx_busy;
    wire u_rx_ready;
    wire [UART_RX_DATA_WIDTH-1:0] u_dout;

    // Instantiate modules

    // Slave port
    slave_port #(
        .ADDR_WIDTH(ADDR_WIDTH),
        .DATA_WIDTH(DATA_WIDTH),
        .SPLIT_EN(SPLIT_EN)
    )slave(
        .clk(clk),
        .rstn(rstn),
        .smemrdata(smemrdata),
        .rvalid(rvalid),
        .smemwen(smemwen),
        .smemren(smemren),
        .smemaddr(smemaddr),
        .smemwdata(smemwdata),
        .swdata(swdata),
        .srdata(srdata),
        .smode(smode),
        .mvalid(mvalid),
        .split_grant(split_grant),
        .svalid(svalid),
        .sready(spready),
        .ssplit(ssplit)
    );


    // UART module
    uart #(
        .CLOCKS_PER_PULSE(UART_CLOCKS_PER_PULSE),
        .TX_DATA_WIDTH(UART_TX_DATA_WIDTH),
        .RX_DATA_WIDTH(UART_RX_DATA_WIDTH)
    ) uart_module (
        .data_input(u_din),
        .data_en(u_en),
        .clk(clk),
        .rstn(rstn),
        .tx(u_tx),  // Transmitter output (tx)
        .tx_busy(u_tx_busy),
        .rx(u_rx),
        .ready(u_rx_ready),
        .data_output(u_dout)
    );

    localparam IDLE  = 2'b00,    //0
               WSEND  = 2'b01,   // Write data
               RSEND = 2'b10,    // Read data
               RDATA = 2'b11;    // Wait until receive
      // State variables
      reg [1:0] state, next_state;

      // Next state logic
      always @(*) begin
             case (state)
                    IDLE   : next_state = (smemwen) ? WSEND : ((smemren) ? RSEND :
IDLE);
                    WSEND  : next_state = (u_tx_busy) ? WSEND : IDLE;
            RSEND  : next_state = (u_tx_busy) ? RSEND : RDATA;
            RDATA  : next_state = (!smemren) ? IDLE : RDATA;
                    default: next_state = IDLE;
             endcase
      end

      // State transition logic
      always @(posedge clk) begin
             state <= (!rstn) ? IDLE : next_state;
      end
    // Send write data from slave port to UART TX
    always @(posedge clk) begin
```

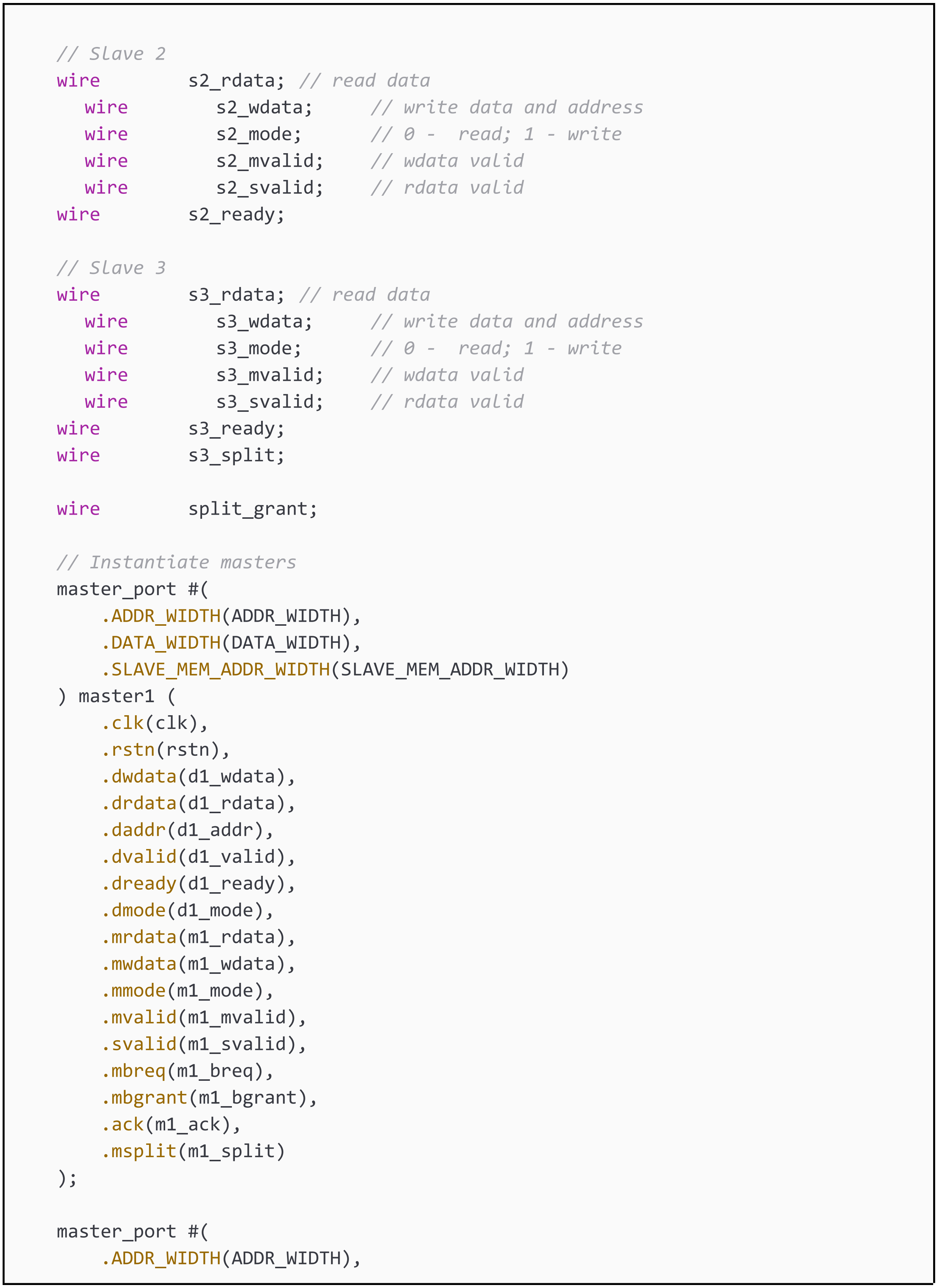
```
        if (!rstn) begin
            u_din <= 'b0;
            u_en <= 1'b0;
        end
        else begin
            case (state)
                IDLE : begin
                    u_din <= u_din;
                    u_en <= 1'b0;
                end

                WSEND : begin
                    // Send address , data, mode
                    u_din <= {1'b1, smemwdata, smemaddr}; //[0:11] ADDR  [12:19]
WDATA [20] mode
                    u_en  <= 1'b1;
                end
                RSEND : begin
                    // Send read address, mode
                    u_din <= {1'b0, {DATA_WIDTH{1'b0}}, smemaddr}; //[0:11] ADDR
[12:19] WDATA [20] mode
                    u_en  <= 1'b1;
                end
                RDATA : begin
                    // No transmission when not writing
                    u_din <= u_din;
                    u_en <= 1'b0;
                end

                default : begin
                    u_din <= u_din;
                    u_en <= 1'b0;
                end
            endcase
        end
    end

    assign rvalid = (state == RDATA) && (!u_tx_busy) && (u_rx_ready);
    assign smemrdata = (smemren) ? u_dout : {DATA_WIDTH{1'b0}};
    assign sready = spready && !smemwen && !smemren && (state == IDLE);

endmodule
```

12.9. Testbench for bus functionality and split transactions

```
`timescale 1ns/1ps
```



```
module master2_slave3_tb;
    // Parameters
    parameter ADDR_WIDTH = 16;
    parameter DATA_WIDTH = 8;
    parameter SLAVE_MEM_ADDR_WIDTH = 12;
    parameter DEVICE_ADDR_WIDTH = ADDR_WIDTH - SLAVE_MEM_ADDR_WIDTH;

    // External signals
    reg clk, rstn;
    reg [DATA_WIDTH-1:0] d1_wdata, d2_wdata;  // Write data to the DUT
    wire [DATA_WIDTH-1:0] d1_rdata, d2_rdata; // Read data from the DUT
    reg [ADDR_WIDTH-1:0] d1_addr, d2_addr;
    reg d1_valid, d2_valid;                            // Ready valid interface
    wire d1_ready, d2_ready;
    reg d1_mode, d2_mode;                              // 0 - read, 1 - write

    // Bus signals
    // Master 1
    wire        m1_rdata; // read data
      wire         m1_wdata;    // write data and address
      wire         m1_mode;     // 0 -   read; 1 - write
      wire         m1_mvalid;   // wdata valid
      wire        m1_svalid;    // rdata valid
      wire         m1_breq;
      wire        m1_bgrant;
    wire        m1_ack;
    wire        m1_split;

    // Master 2
    wire        m2_rdata; // read data
      wire         m2_wdata;    // write data and address
      wire         m2_mode;     // 0 -   read; 1 - write
      wire         m2_mvalid;   // wdata valid
      wire        m2_svalid;    // rdata valid
      wire         m2_breq;
      wire        m2_bgrant;
    wire        m2_ack;
    wire        m2_split;

    // Slave 1
    wire        s1_rdata; // read data
      wire        s1_wdata;     // write data and address
      wire        s1_mode;      // 0 -   read; 1 - write
      wire        s1_mvalid;    // wdata valid
      wire        s1_svalid;    // rdata valid
    wire        s1_ready;
```



```
// Slave 2
wire        s2_rdata; // read data
   wire        s2_wdata;     // write data and address
   wire        s2_mode;      // 0 -  read; 1 - write
   wire        s2_mvalid;    // wdata valid
   wire        s2_svalid;    // rdata valid
wire        s2_ready;

// Slave 3
wire        s3_rdata; // read data
   wire        s3_wdata;     // write data and address
   wire        s3_mode;      // 0 -  read; 1 - write
   wire        s3_mvalid;    // wdata valid
   wire        s3_svalid;    // rdata valid
wire        s3_ready;
wire        s3_split;

wire        split_grant;

// Instantiate masters
master_port #(
    .ADDR_WIDTH(ADDR_WIDTH),
    .DATA_WIDTH(DATA_WIDTH),
    .SLAVE_MEM_ADDR_WIDTH(SLAVE_MEM_ADDR_WIDTH)
) master1 (
    .clk(clk),
    .rstn(rstn),
    .dwdata(d1_wdata),
    .drdata(d1_rdata),
    .daddr(d1_addr),
    .dvalid(d1_valid),
    .dready(d1_ready),
    .dmode(d1_mode),
    .mrdata(m1_rdata),
    .mwdata(m1_wdata),
    .mmode(m1_mode),
    .mvalid(m1_mvalid),
    .svalid(m1_svalid),
    .mbreq(m1_breq),
    .mbgrant(m1_bgrant),
    .ack(m1_ack),
    .msplit(m1_split)
);

master_port #(
    .ADDR_WIDTH(ADDR_WIDTH),
        .DATA_WIDTH(DATA_WIDTH),
        .SLAVE_MEM_ADDR_WIDTH(SLAVE_MEM_ADDR_WIDTH)
    ) master2 (
        .clk(clk),
        .rstn(rstn),
        .dwdata(d2_wdata),
        .drdata(d2_rdata),
        .daddr(d2_addr),
        .dvalid(d2_valid),
        .dready(d2_ready),
        .dmode(d2_mode),
        .mrdata(m2_rdata),
        .mwdata(m2_wdata),
        .mmode(m2_mode),
        .mvalid(m2_mvalid),
        .svalid(m2_svalid),
        .mbreq(m2_breq),
        .mbgrant(m2_bgrant),
        .ack(m2_ack),
        .msplit(m2_split)
    );

    // Initialize slave
    slave #(
        .ADDR_WIDTH(SLAVE_MEM_ADDR_WIDTH),
        .DATA_WIDTH(DATA_WIDTH)
    ) slave1 (
        .clk(clk),
        .rstn(rstn),
        .srdata(s1_rdata),
        .swdata(s1_wdata),
        .smode(s1_mode),
        .svalid(s1_svalid),
        .mvalid(s1_mvalid),
        .sready(s1_ready),
        .ssplit(),
        .split_grant(0)
    );

    slave #(
        .ADDR_WIDTH(SLAVE_MEM_ADDR_WIDTH),
        .DATA_WIDTH(DATA_WIDTH)
    ) slave2 (
        .clk(clk),
        .rstn(rstn),
        .srdata(s2_rdata),
        .swdata(s2_wdata),
        .smode(s2_mode),
        .svalid(s2_svalid),
        .mvalid(s2_mvalid),
        .sready(s2_ready),
        .ssplit(),
        .split_grant(0)
    );

    slave #(
        .ADDR_WIDTH(SLAVE_MEM_ADDR_WIDTH),
        .DATA_WIDTH(DATA_WIDTH),
        .SPLIT_EN(1)
    ) slave3 (
        .clk(clk),
        .rstn(rstn),
        .srdata(s3_rdata),
        .swdata(s3_wdata),
        .smode(s3_mode),
        .svalid(s3_svalid),
        .mvalid(s3_mvalid),
        .sready(s3_ready),
        .ssplit(s3_split),
        .split_grant(split_grant)
    );

    // Bus
    bus_m2_s3 #(
        .ADDR_WIDTH(ADDR_WIDTH),
        .DATA_WIDTH(DATA_WIDTH),
        .SLAVE_MEM_ADDR_WIDTH(SLAVE_MEM_ADDR_WIDTH)
    ) bus (
        .clk(clk),
        .rstn(rstn),

        // Master 1 connections
        .m1_rdata(m1_rdata),
        .m1_wdata(m1_wdata),
        .m1_mode(m1_mode),
        .m1_mvalid(m1_mvalid),
        .m1_svalid(m1_svalid),
        .m1_breq(m1_breq),
        .m1_bgrant(m1_bgrant),
        .m1_ack(m1_ack),
        .m1_split(m1_split),

        // Master 2 connections
        .m2_rdata(m2_rdata),
        .m2_wdata(m2_wdata),
        .m2_mode(m2_mode),
        .m2_mvalid(m2_mvalid),
        .m2_svalid(m2_svalid),
        .m2_breq(m2_breq),
        .m2_bgrant(m2_bgrant),
        .m2_ack(m2_ack),
        .m2_split(m2_split),

        // Slave 1 connections
        .s1_rdata(s1_rdata),
        .s1_wdata(s1_wdata),
        .s1_mode(s1_mode),
        .s1_mvalid(s1_mvalid),
        .s1_svalid(s1_svalid),
        .s1_ready(s1_ready),

        .s2_rdata(s2_rdata),
        .s2_wdata(s2_wdata),
        .s2_mode(s2_mode),
        .s2_mvalid(s2_mvalid),
        .s2_svalid(s2_svalid),
        .s2_ready(s2_ready),

        .s3_rdata(s3_rdata),
        .s3_wdata(s3_wdata),
        .s3_mode(s3_mode),
        .s3_mvalid(s3_mvalid),
        .s3_svalid(s3_svalid),
        .s3_ready(s3_ready),
        .s3_split(s3_split),

        .split_grant(split_grant)
    );

    wire s_ready;
    assign s_ready = s1_ready & s2_ready & s3_ready;

    // Generate Clock
    initial begin
        clk = 0;
        forever #5 clk = ~clk; // Clock period is 10 units
    end

    integer i;
    reg [ADDR_WIDTH-1:0] rand_addr1, rand_addr2, rand_addr3;
    reg [DATA_WIDTH-1:0] rand_data1, rand_data2;
    reg [DATA_WIDTH-1:0] slave_mem_data1, slave_mem_data2;
    reg [1:0] slave_id1, slave_id2;

    task random_delay;
        integer delay;
        begin
            delay = $urandom % 10;  // Generate a random delay multiplier between 0
and 4
            $display("Random delay: %d", delay * 10);
            #(delay * 10);  // Delay in multiples of 10 time units (clock period)
        end
    endtask

    // Test Stimulus
    initial begin
        // Reset the DUT
        rstn = 0;
        d1_valid = 0;
        d1_wdata = 8'b0;
        d1_addr = 16'b0;
        d1_mode = 0;
        d2_valid = 0;
        d2_wdata = 8'b0;
        d2_addr = 16'b0;
        d2_mode = 0;

        #15 rstn = 1; // Release reset after 15 time units

        // Repeat the write and read tests 10 times
        for (i = 0; i < 20; i = i + 1) begin

            // Generate random address and data
            rand_addr1 = $random & 14'h3FFF;
            rand_data1 = $random;
            rand_addr2 = $random & 14'h3FFF;
            rand_data2 = $random;

            slave_id1 = rand_addr1[ADDR_WIDTH-DEVICE_ADDR_WIDTH+:2];
            slave_id2 = rand_addr2[ADDR_WIDTH-DEVICE_ADDR_WIDTH+:2];

            // Write Operation: Sending data to the bus
            // Do 2 request next to each other from different masters

            wait (d1_ready == 1 && d2_ready == 1 && s_ready == 1);
            @(posedge clk);
            d1_addr = rand_addr1[ADDR_WIDTH-1:0];  // Set address with random value
            d1_wdata = rand_data1[DATA_WIDTH-1:0]; // Write data value
            d1_mode = 1;                                  // Set mode to write
            d1_valid = 1;                                 // Assert valid signal

            random_delay();

            // Make request from m2
            @(posedge clk)
            d2_addr = rand_addr2[ADDR_WIDTH-1:0];  // Set address with random value
            d2_wdata = rand_data2[DATA_WIDTH-1:0]; // Write data value
            d2_mode = 1;                                  // Set mode to write
            d2_valid = 1;                                 // Assert valid signal

            #20;
            d1_valid = 0;
            d2_valid = 0;

            wait (d1_ready == 1 && d2_ready == 1 && s_ready == 1);
            #20;

            if (slave_id1 == 2'b00)  slave_mem_data1 =
slave1.sm.memory[d1_addr[11:0]];
            else if (slave_id1 == 2'b01)  slave_mem_data1 =
slave2.sm.memory[d1_addr[11:0]];
            else if (slave_id1 == 2'b10)  slave_mem_data1 =
slave3.sm.memory[d1_addr[11:0]];

            if (slave_id1 != 2'b11 && slave_mem_data1 != d1_wdata) begin
                $display("Master 1 write failed at iteration %0d: location %0x,
expected %x, actual %x",
                            i, d1_addr, d1_wdata, slave_mem_data1);
            end else begin
                $display("Master 1 write to %0x successful at iteration %0d",
d1_addr, i);
            end

            if (slave_id2 == 2'b00)  slave_mem_data2 =
slave1.sm.memory[d2_addr[11:0]];
            else if (slave_id2 == 2'b01)  slave_mem_data2 =
slave2.sm.memory[d2_addr[11:0]];
            else if (slave_id2 == 2'b10)  slave_mem_data2 =
slave3.sm.memory[d2_addr[11:0]];


            if (slave_id2 != 2'b11 && slave_mem_data2 != d2_wdata) begin
                $display("Master 2 write failed at iteration %0d: location %0x,
expected %x, actual %x",
                            i, d2_addr, d2_wdata, slave_mem_data2);
            end else begin
                $display("Master 2 write to %0x successful at iteration %0d",
d2_addr, i);
            end

            // Read operation: make both requests on the same clock cycle
            @(posedge clk);
            d1_mode = 0;                                 // Set mode to read
            d1_valid = 1;                                // Assert valid signal
            d2_mode = 0;
            d2_valid = 1;

            #20;
            d1_valid = 0;
            d2_valid = 0;
            wait (d1_ready == 1 && d2_ready == 1 && s_ready == 1);

            #20;
            if (slave_id1 != 2'b11 && d1_wdata != d1_rdata) begin
                $display("Master 1 read failed at iteration %0d: location %0x,
expected %x, actual %x",
                            i, d1_addr, d1_wdata, d1_rdata);
            end else begin
                $display("Master 1 read from %0x successful at iteration %0d",
d1_addr, i);
            end

            if (slave_id2 != 2'b11 && d2_wdata != d2_rdata) begin
                $display("Master 2 read failed at iteration %0d: location %0x,
expected %x, actual %x",
                            i, d2_addr, d2_wdata, d2_rdata);
            end else begin
                $display("Master 2 read from %0x successful at iteration %0d",
d2_addr, i);
            end

            // Master 2 write and master 1 read
            rand_addr3 = $random & 14'h3FFF;
            slave_id1 = rand_addr3[ADDR_WIDTH-DEVICE_ADDR_WIDTH+:2];

            @(posedge clk);
            d2_addr = rand_addr3[ADDR_WIDTH-1:0];  // Set address with random value
            d2_wdata = rand_data1 + rand_data2; // Write data value
            d2_mode = 1;                                  // Set mode to write
            d2_valid = 1;                                 // Assert valid signal

            random_delay();
            @(posedge clk)
            d1_addr = d2_addr;
            d1_mode = 0;                         // Set mode to read
            d1_valid = 1;                        // Assert valid signal

            #20;
            d1_valid = 0;
            d2_valid = 0;
            wait (d1_ready == 1 && d2_ready == 1 && s_ready == 1);

            #20;

            if (slave_id1 == 2'b00)  slave_mem_data1 =
slave1.sm.memory[d2_addr[11:0]];
            else if (slave_id1 == 2'b01)  slave_mem_data1 =
slave2.sm.memory[d2_addr[11:0]];
            else if (slave_id1 == 2'b10)  slave_mem_data1 =
slave3.sm.memory[d2_addr[11:0]];

            if (slave_id1 != 2'b11 && slave_mem_data1 != d2_wdata) begin
                $display("Master 2 write failed at iteration %0d: location %0x,
expected %x, actual %x",
                            i, d2_addr, d2_wdata, slave_mem_data1);
            end else begin
                $display("Master 2 write to %0x successful at iteration %0d",
d2_addr, i);
            end

            if (slave_id1 != 2'b11 && d2_wdata != d1_rdata) begin
                $display("Master 1 read failed at iteration %0d: location %0x,
expected %x, actual %x",
                            i, d1_addr, d2_wdata, d1_rdata);
            end else begin
                $display("Master 1 read from %0x successful at iteration %0d",
d1_addr, i);
            end

            // Small delay before next iteration
            #10;
        end

        #10 $finish;
    end


endmodule
```

12.10. Testbench for bus bridge connectivity

```
`timescale 1ns/1ps

module bb_loop_tb;
    // Parameters
    localparam ADDR_WIDTH = 16;
    localparam DATA_WIDTH = 8;
    localparam SLAVE_MEM_ADDR_WIDTH = 13;
    localparam BB_ADDR_WIDTH = 13;

    localparam DEVICE_ADDR_WIDTH = ADDR_WIDTH - SLAVE_MEM_ADDR_WIDTH;
    localparam UART_RX_DATA_WIDTH = DATA_WIDTH + BB_ADDR_WIDTH + 1;
    localparam UART_TX_DATA_WIDTH = DATA_WIDTH;     // Transmit only read data
    localparam UART_CLOCKS_PER_PULSE = 5208;

    // External signals
    reg clk, rstn;
    reg [DATA_WIDTH-1:0] d1_wdata;  // Write data to the DUT
    wire [DATA_WIDTH-1:0] d1_rdata; // Read data from the DUT
    reg [ADDR_WIDTH-1:0] d1_addr;
    reg d1_valid;                        // Ready valid interface
    wire d1_ready;
    reg d1_mode;                                // 0 - read, 1 - write

    wire d1_sready;      // slaves are ready

    reg [DATA_WIDTH-1:0] d2_wdata;  // Write data to the DUT
    wire [DATA_WIDTH-1:0] d2_rdata; // Read data from the DUT
    reg [ADDR_WIDTH-1:0] d2_addr;
    reg d2_valid;                        // Ready valid interface
    wire d2_ready;
    reg d2_mode;                                // 0 - read, 1 - write

    wire d2_sready;      // slaves are ready

    // UART signals
    wire m_u_rx, s_u_rx;
    wire m_u_tx, s_u_tx;

    // Instantiate masters
    top_with_bb #(
        .ADDR_WIDTH(ADDR_WIDTH),
        .DATA_WIDTH(DATA_WIDTH),
        .SLAVE_MEM_ADDR_WIDTH(SLAVE_MEM_ADDR_WIDTH),
        .BB_ADDR_WIDTH(BB_ADDR_WIDTH),
        .UART_CLOCKS_PER_PULSE(UART_CLOCKS_PER_PULSE)
    ) bus1 (
        .clk(clk),
        .rstn(rstn),
        .d1_wdata(d1_wdata),
        .d1_rdata(d1_rdata),
        .d1_addr(d1_addr),
        .d1_valid(d1_valid),
        .d1_ready(d1_ready),
        .d1_mode(d1_mode),
        .s_ready(d1_sready),
        .m_u_rx(m_u_rx),
        .m_u_tx(m_u_tx),
        .s_u_rx(s_u_rx),
        .s_u_tx(s_u_tx)
    );

    top_with_bb #(
        .ADDR_WIDTH(ADDR_WIDTH),
        .DATA_WIDTH(DATA_WIDTH),
        .SLAVE_MEM_ADDR_WIDTH(SLAVE_MEM_ADDR_WIDTH),
        .BB_ADDR_WIDTH(BB_ADDR_WIDTH),
        .UART_CLOCKS_PER_PULSE(UART_CLOCKS_PER_PULSE)
    ) bus2 (
        .clk(clk),
        .rstn(rstn),
        .d1_wdata(d2_wdata),
        .d1_rdata(d2_rdata),
        .d1_addr(d2_addr),
        .d1_valid(d2_valid),
        .d1_ready(d2_ready),
        .d1_mode(d2_mode),
        .s_ready(d2_sready),
        .m_u_rx(s_u_tx),
        .m_u_tx(s_u_rx),
        .s_u_rx(m_u_tx),
        .s_u_tx(m_u_rx)
    );

    // Generate Clock
    initial begin
        clk = 0;
        forever #5 clk = ~clk; // Clock period is 10 units
    end

    integer i;
    reg [ADDR_WIDTH-1:0] rand_addr1, rand_addr2, rand_addr3;
    reg [DATA_WIDTH-1:0] rand_data1, rand_data2;
```

```
  // Test Stimulus
initial begin

    // Reset the DUT
    rstn = 0;
    d1_valid = 0;
    d1_wdata = 8'b0;
    d1_addr = 16'b0;
    d1_mode = 0;

    d2_valid = 0;
    d2_wdata = 8'b0;
    d2_addr = 16'b0;
    d2_mode = 0;

    #15 rstn = 1; // Release reset after 15 time units

    // Repeat the write and read tests 10 times
    for (i = 0; i < 10; i = i + 1) begin

        // Generate random address and data
        rand_addr1 = $random & 14'h3FFF;
        rand_data1 = $random;
        rand_addr2 = $random & 12'hFFF;
        rand_data2 = $random;
        slave_id = i & 1;

        // Write request to random location in slave 0 across bus bridge
        wait (d1_ready == 1);
        d1_wdata = rand_data2;
        d1_addr = {3'b010, slave_id, rand_addr2[11:0]};
        d1_mode = 1;
        d1_valid = 1;

        #20 d1_valid = 0;

        // Send read request
        if (slave_id == 0) begin
            @(posedge bus2.s1_ready);
        end else begin
            @(posedge bus2.s2_ready);
        end

        d2_addr = {2'b00, slave_id, 1'b0, rand_addr2[11:0]};
        d2_mode = 0;
        d2_valid = 1;
```

```
            #20 d2_valid = 0;

            wait (d2_ready == 1 && d2_sready == 1);

            if (rand_data2 != d2_rdata) begin
                $display("Bus bridge write failed at iteration %0d: location %x,
expected %x, actual %x",
                            i, rand_addr2[11:0], rand_data2, d2_rdata);
            end else begin
                $display("Bus bridge write successful at iteration %0d", i);
            end

            // Read request across bus bridge
            wait (d1_ready == 1);
            d1_wdata = 8'b0;
            d1_addr = {3'b010, slave_id, rand_addr2[11:0]};
            d1_mode = 0;
            d1_valid = 1;

            #20 d1_valid = 0;

            // Send read request
            wait (d1_ready == 1 && d1_sready == 1 && d2_sready == 1);

            if (rand_data2 != d1_rdata) begin
                $display("Bus bridge read failed at iteration %0d: location %x,
expected %x, actual %x",
                            i, rand_addr2[11:0], rand_data2, d1_rdata);
            end else begin
                $display("Bus bridge read successful at iteration %0d", i);
            end

        end

        #10 $finish;
    end


endmodule
```