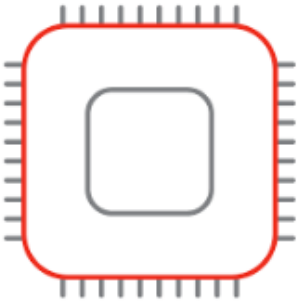

SYSTEM BUS DESIGN

FOR 2 MASTER, 3 SLAVE

Designed by

Kaushan

Rumesh

Vigeitharan

Updated on

7 August, 2022

Contents

1 Introduction

This document is intended to describe the design and simulation of a system bus. Required details for each of the given tasks are included in different sections of this document along with functional descriptions, IO definitions, timing diagrams, RTL & testbench codes. The design was done using Verilog. Xilinx ISE 14.7 is the software used for simulation and compilation for checking errors. Complete Source code is available at github.

A system bus is a communication medium in the digital domain which facilitates data transfer between components and devices connected to it. Best example is the system bus in a computer which connects the Central Processing Unit (CPU), main memory and other major components together. Generally a system bus consists of a Data Bus to carry data, an Address Bus to determined where data should be sent and a Control Bus to maintain the proper control of the system bus without any collisions of data transfers.

In this project, we designed a custom serial system bus with a custom protocol. The design consists of a serial address bus with a 16-bit address format, There is another major component called arbiter which directly communicates with any master or slave device, connected to the bus in order to maintain access and control of the bus. Figure 1 shows the architecture of the design.

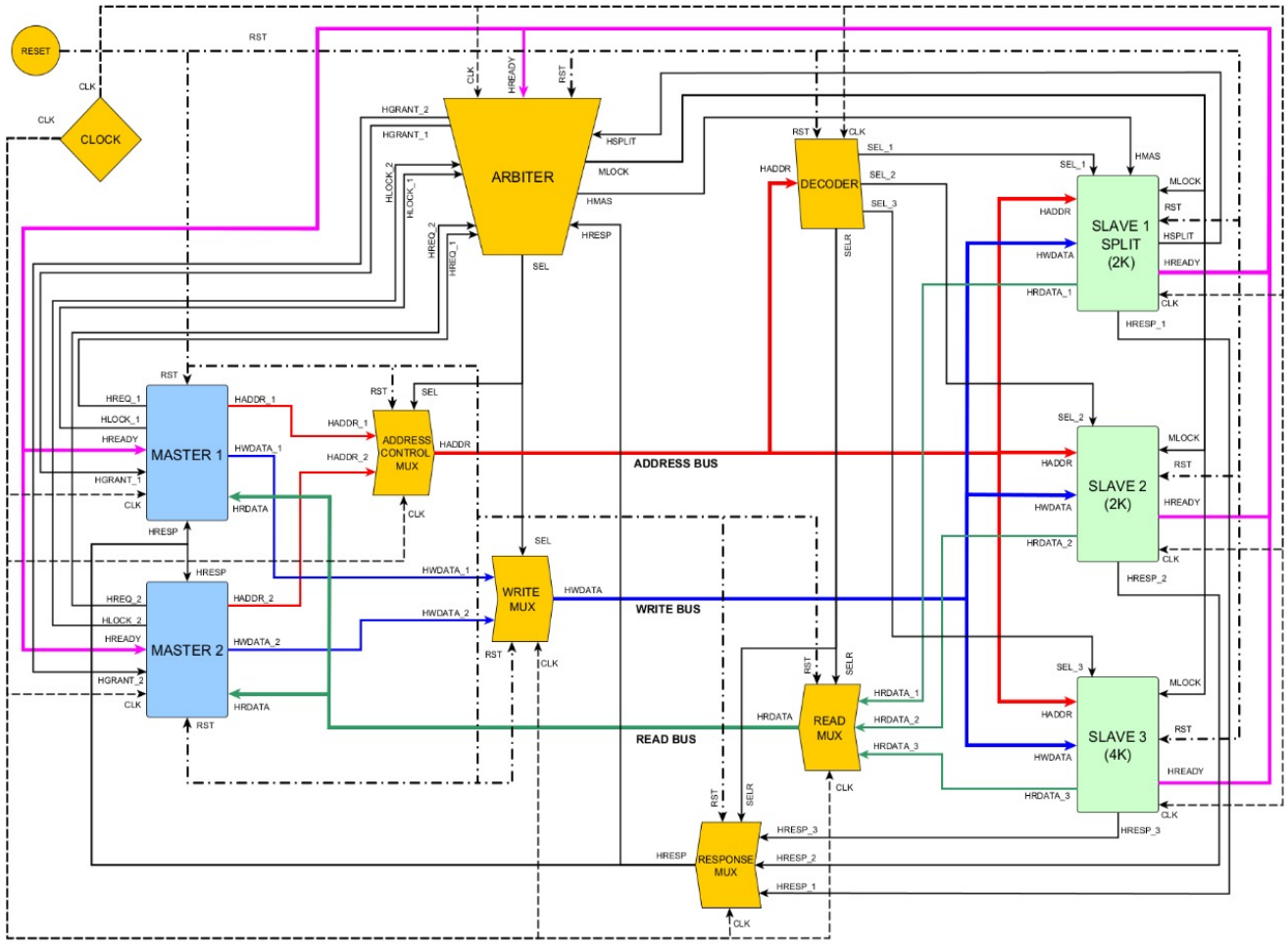

Figure 1: Implemented Architecture

2 Bus Signal List

Bus Signal List

No.	Signal	Line Width (Bits)	Source	Destination	Description
1	RST	[0]	Reset Switch	ALL	Asynchronous and active HIGH signal, used as a global reset for all modules in system
2	CLK	[0]	Clock	ALL	Synchronous clock with rising edge trigger.
3	HREADY	[0]	Slaves	Arbiter, Masters	Used to extend a transfer for an additional clock cycle if signal driven LOW. Default HIGH.
4	HADDR	[15:0]	Address Mux	Decoder, ALL Slaves	16-bit address bus lane. Transmits target device, address and control signals.
5	HADDR_1	[15:0]	Master 1	Address Mux	16-bit address lane from Master 1 to Address Mux
6	HADDR_2	[15:0]	Master 2	Address Mux	16-bit address lane from Master 2 to Address Mux
7	HWDATA	[31:0]	Write Mux	All Slaves	32-bit write bus lane for transfering WORD sized data across bus from Masters to Slaves.
8	HWDATA_1	[31:0]	Master 1	Write Mux	32-bit write bus lane from Master 1 to Write Mux
9	HWDATA_2	[31:0]	Master 2	Write Mux	32-bit write bus lane from Master 2 to Write Mux
10	HREQ_1	[0]	Master 1	Arbiter	Signal from Master 1 to Arbiter requesting access to bus. Driven HIGH for requesting access to the bus and driven LOW when bus access granted.
11	HREQ_2	[0]	Master 2	Arbiter	Signal from Master 2 to Arbiter requesting access to bus. Driven HIGH for requesting access to the bus and driven LOW when bus access granted.
12	HLOCK_1	[0]	Master 1	Arbiter	Locked access priority request signal from Master 1 to Arbiter. Higher priority over normal requests. Arbiter grants locked access to Master 1 indefinetly when signal remains HIGH. Must be driven LOW to release access to the bus.
13	HLOCK_2	[0]	Master 2	Arbiter	Locked access priority request signal from Master 2 to Arbiter. Higher priority over normal requests. Arbiter grants locked access to Master 2 indefinetly when signal remains HIGH. Must be driven LOW to release access to the bus.
14	HGRANT_1	[0]	Arbiter	Master 1	Signal from Arbiter granting bus access to Master 1. Signal remains HIGH during which, transactions between Master 1 and selected Slave devices can take place. Access revoked when signal driven LOW.
15	HGRANT_2	[0]	Arbiter	Master 2	Signal from Arbiter granting bus access to Master 2. Signal remains HIGH during which, transactions between Master 2 and selected Slave devices can take place. Access revoked when signal driven LOW.
16	SEL	[1:0]	Arbiter	Address Mux, Write Mux	2-bit signal from Arbiter to toggle selections in the Address and Write Muxes. Driven HIGH when a Master requests access to the bus and the Arbiter returns grant access. Driven LOW when Master releases control of bus or Arbiter revokes control of bus. 2'b01 selects Address and Write bus access to Master 1. 2'b10 sets accesss to Master 2.
17	SELR	[1:0]	Decoder	Read Mux, Response Mux	2-bit signal from Decorder to toggle selections in the Read and Response Muxes, that is requested by the data transmitted on the Address bus from a Master. 2'b01 selects Slave 1, 2'b10 selects Slave 2 and 2'b11 selects Slave 3. Default, 2'b00, no Slave selected.
18	SEL_1	[0]	Decoder	Slave 1	Signal from Decoder to Slave 1. Driven HIGH when a Master issues the respective Slave ID on the address bus. Must remain high in order for transactions between Master and Slave to take place. Driven low, when bus access is released by Master or Arbiter.
19	SEL_2	[0]	Decoder	Slave 2	Signal from Decoder to Slave 2. Driven HIGH when a Master issues the respective Slave ID on the address bus. Must remain high in order for transactions between Master and Slave to take place. Driven low, when bus access is released by Master or Arbiter.
20	SEL_3	[0]	Decoder	Slave 3	Signal from Decoder to Slave 3. Driven HIGH when a Master issues the respective Slave ID on the address bus. Must remain high in order for transactions between Master and Slave to take place. Driven low, when bus access is released by Master or Arbiter.
21	HRDATA	[31:0]	Read Mux	All Masters	32-bit read bus lane for transfering WORD sized data across bus from Slaves to Masters.
22	HRDATA_1	[31:0]	Slave 1	Read Mux	32-bit read bus lane from Slave 1 to Read Mux.
23	HRDATA_2	[31:0]	Slave 2	Read Mux	32-bit read bus lane from Slave 2 to Read Mux.
24	HRDATA_3	[31:0]	Slave 3	Read Mux	32-bit read bus lane from Slave 3 to Read Mux.
25	MLOCK	[0]	Arbiter	All Slaves	Signal from Arbiter to Slaves indicating current transaction is locked, hence bus access granted indefinitely till transaction successfuly completed and lock removed. When driven HIGH, Slave is forced to retry the transactions immediately until completed successfully.
26	HMAS	[1:0]	Arbiter	Slave 1	Signal transmitting the Master ID of the current Master that is connected to Slave 1. Used to send a reconnect request to Arbiter with saved Master ID to complete a split transaction.
27	HSPLIT	[1:0]	Slave 1	Arbiter	Signal transmitted from Slave 1 to Arbiter with Master ID of Master that was previously connected to Slave 1 during which a Split transaction command was invoked. Signal transmitted requesting access to bus, when Slave 1 ready to reconnect with Master and complete the Split transaction. Split transactions are considered highest priority commands here.
28	HRESP	[1:0]	Response Mux	Arbiter, Masters	2-bit signal used to transmit the transaction status post-transaction from Slaves to Masters and Arbiter. The response is monitored and Retry, Split or Error cancel decisions with Grant extension or Grant revoke taken by Master and Arbiter. 2'b00 = `OKAY, 2'b01 = `ERROR, 2'b10 = `RETRY, 2'b11 = `SPLIT.
29	HRESP_1	[1:0]	Slave 1	Response Mux	2-bit response signal from Slave 1 to Response Mux
30	HRESP_2	[1:0]	Slave 2	Response Mux	2-bit response signal from Slave 2 to Response Mux
31	HRESP_3	[1:0]	Slave 3	Response Mux	2-bit response signal from Slave 3 to Response Mux

3 Mux Modules

3.1 Address mux

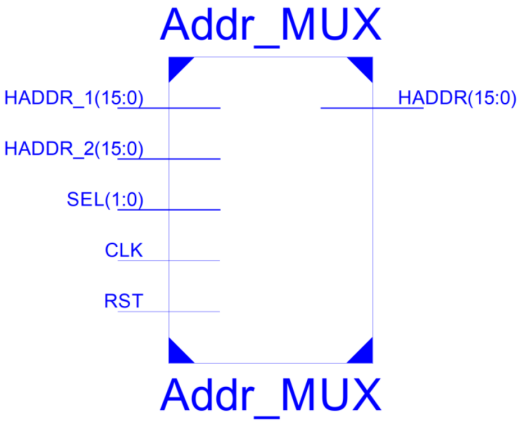

Figure 2: Address Mux Block

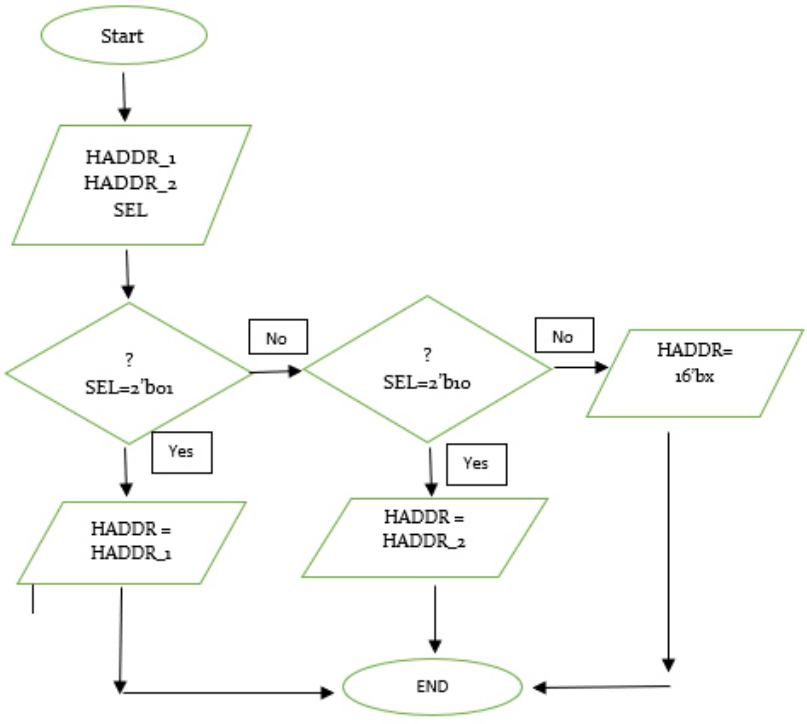

Figure 3: AddrMux Flow Chart

3.1.1 RTL

Verilog Code 1: Test_Addr_MUX.v

```
1 `timescale 1ns / 1ps
2
3 module Test_Addr_MUX;
4
```

```
  // Inputs
  reg CLK;
  reg RST;
  reg [15:0] HADDR_1;
  reg [15:0] HADDR_2;
  reg [1:0] SEL;

  // Outputs
  wire [15:0] HADDR;

  // Instantiate the Unit Under Test (UUT)
  Addr_MUX uut (
    .CLK(CLK),
    .RST(RST),
    .HADDR(HADDR),
    .HADDR_1(HADDR_1),
    .HADDR_2(HADDR_2),
    .SEL(SEL)
  );

  always #15  CLK =  ! CLK;

  initial begin
  #500 RST = 1; // Random reset stimuli
  #50 RST = 0;
  end

  initial begin
    // Initialize Inputs
    CLK = 0;
    RST = 0;
    HADDR_1 = 0;
    HADDR_2 = 0;
    SEL = 0;

    // Wait 100 ns for global reset to finish
    #15;

    // Add stimulus here

    #200 HADDR_1[15:0] = $random; HADDR_2[15:0] = $random; SEL = 2'b00; RST = 0; // no master
        selected
    #200 HADDR_1[15:0] = $random; HADDR_2[15:0] = $random; SEL = 2'b01; RST = 0; // master 1
    #200 HADDR_1[15:0] = $random; HADDR_2[15:0] = $random; SEL = 2'b10; RST = 0; // master 2
    #200 HADDR_1[15:0] = $random; HADDR_2[15:0] = $random; SEL = 2'b01; RST = 0; // master 1
    #200 HADDR_1[15:0] = $random; HADDR_2[15:0] = $random; SEL = 2'b11; RST = 0; // no master
        selected

  end

endmodule
```

3.1.2 Test cases and timing diagrams

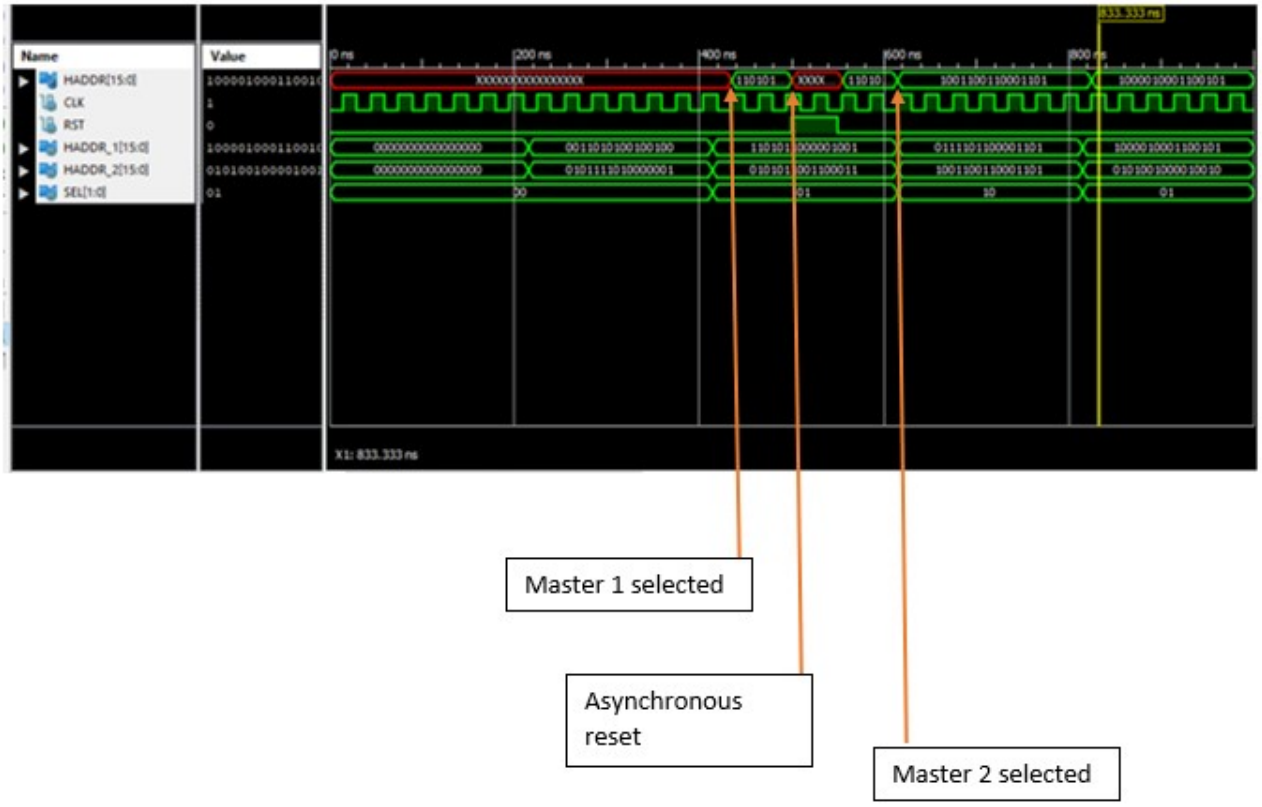

Figure 4: Address Mux Timing Diagram

3.2 Response mux

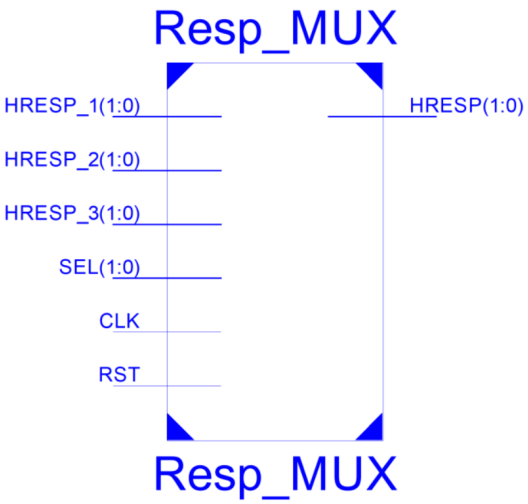

Figure 5: Response Mux Block

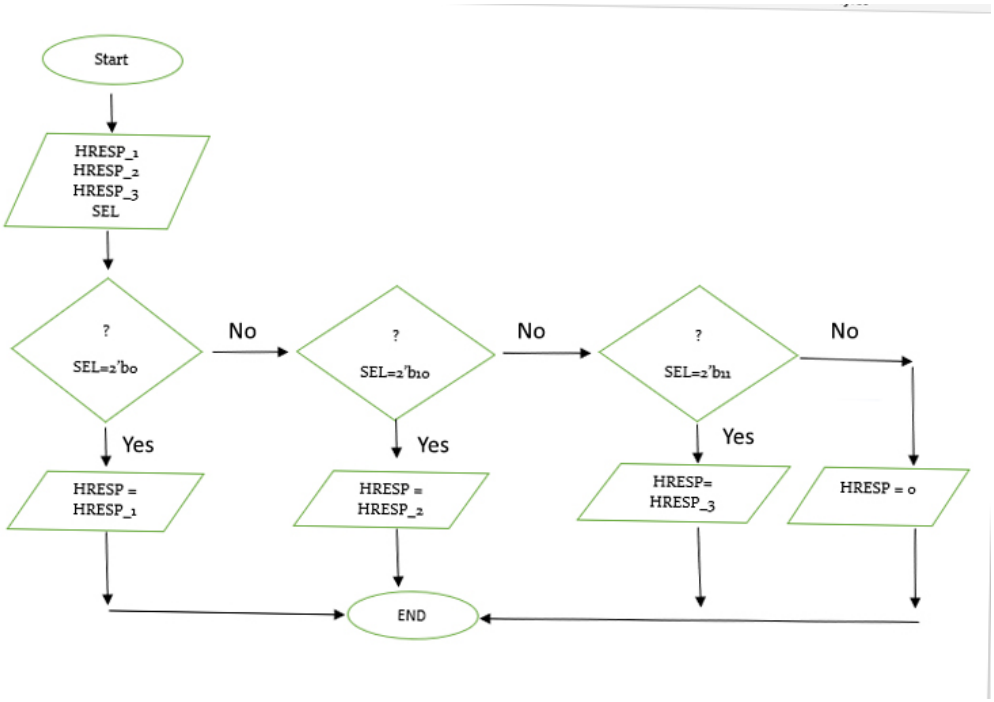

Figure 6: RespMux Flow Chart

3.2.1 RTL

Verilog Code 2: Test_Resp_MUX.v

```
`timescale 1ns / 1ps

module Test_Resp_MUX;

  // Inputs
  reg CLK;
  reg RST;
  reg [1:0] HRESP_1;
  reg [1:0] HRESP_2;
  reg [1:0] HRESP_3;
  reg [1:0] SEL;

  // Outputs
  wire [1:0] HRESP;

  // Instantiate the Unit Under Test (UUT)
  Resp_MUX uut (
    .CLK(CLK),
    .RST(RST),
    .HRESP(HRESP),
    .HRESP_1(HRESP_1),
    .HRESP_2(HRESP_2),
    .HRESP_3(HRESP_3),
    .SEL(SEL)
  );

  always #12  CLK =  ! CLK;

  initial begin
  #500 RST = 1; // Random reset stimuli
  #50 RST = 0;
  end

  initial begin
    // Initialize Inputs
    CLK = 0;
    RST = 0;
    HRESP_1 = 0;
```

```
39      HRESP_2 = 0;
40      HRESP_3 = 0;
41      SEL = 0;
42
43      // Wait 100 ns for global reset to finish
44      #15;
45
46      // Add stimulus here
47
48      #200 HRESP_1[1:0] = $random; HRESP_2[1:0] = $random; HRESP_3[1:0] = $random; SEL = 2'b00;
            RST = 0;
49      #200 HRESP_1[1:0] = $random; HRESP_2[1:0] = $random; HRESP_3[1:0] = $random; SEL = 2'b01;
            RST = 0;
50      #200 HRESP_1[1:0] = $random; HRESP_2[1:0] = $random; HRESP_3[1:0] = $random; SEL = 2'b10;
            RST = 0;
51      #200 HRESP_1[1:0] = $random; HRESP_2[1:0] = $random; HRESP_3[1:0] = $random; SEL = 2'b11;
            RST = 0;
52      #200 HRESP_1[1:0] = $random; HRESP_2[1:0] = $random; HRESP_3[1:0] = $random; SEL = 2'b00;
            RST = 0;
53
54    end
55
56  endmodule
```

3.2.2 Test cases and timing diagrams

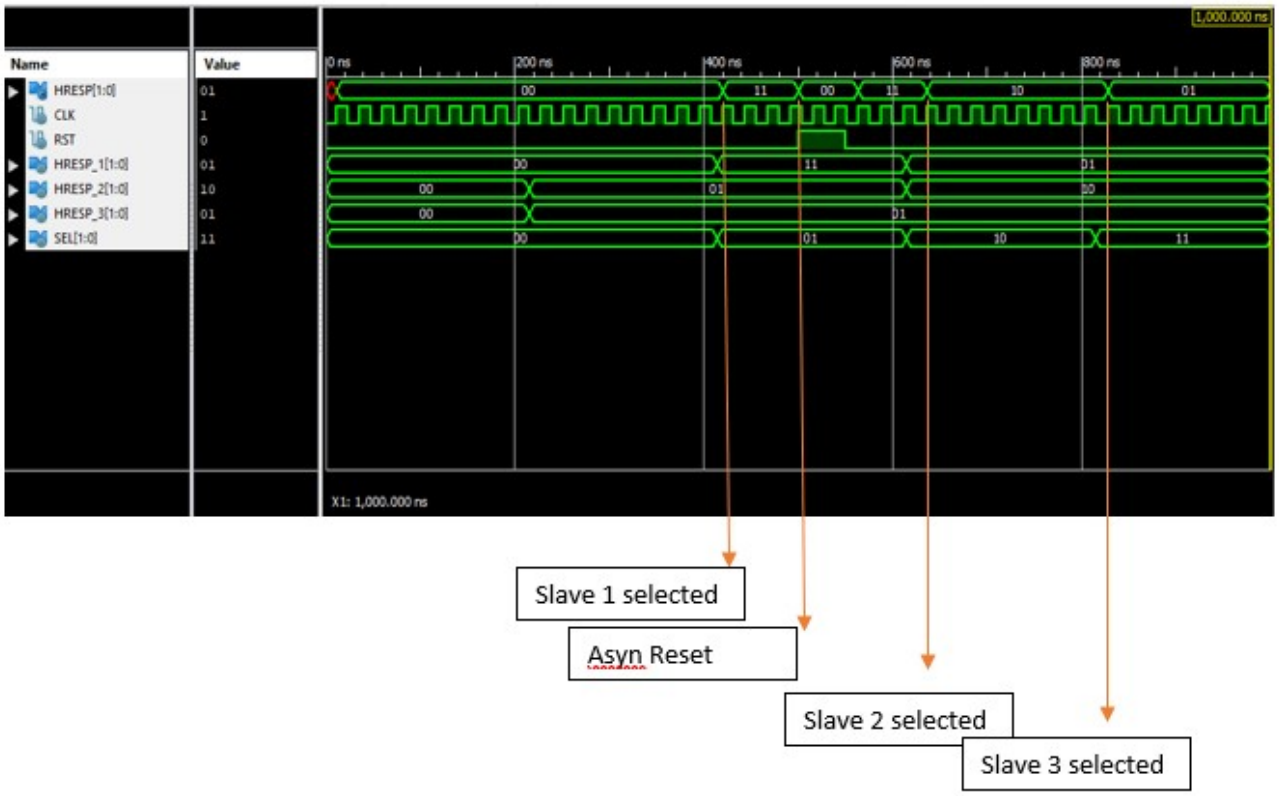

Figure 7: Response Mux Timing Diagram

3.3 Read mux

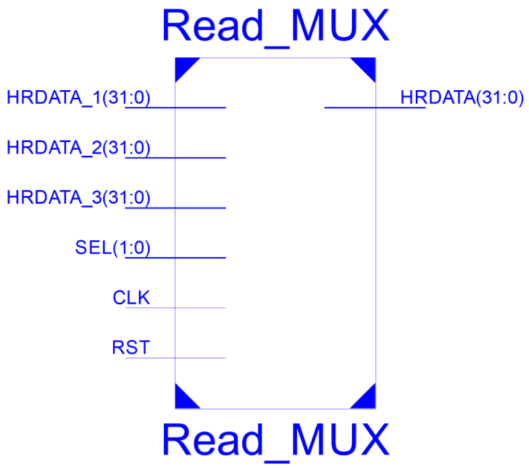

Figure 8: Read Mux Block

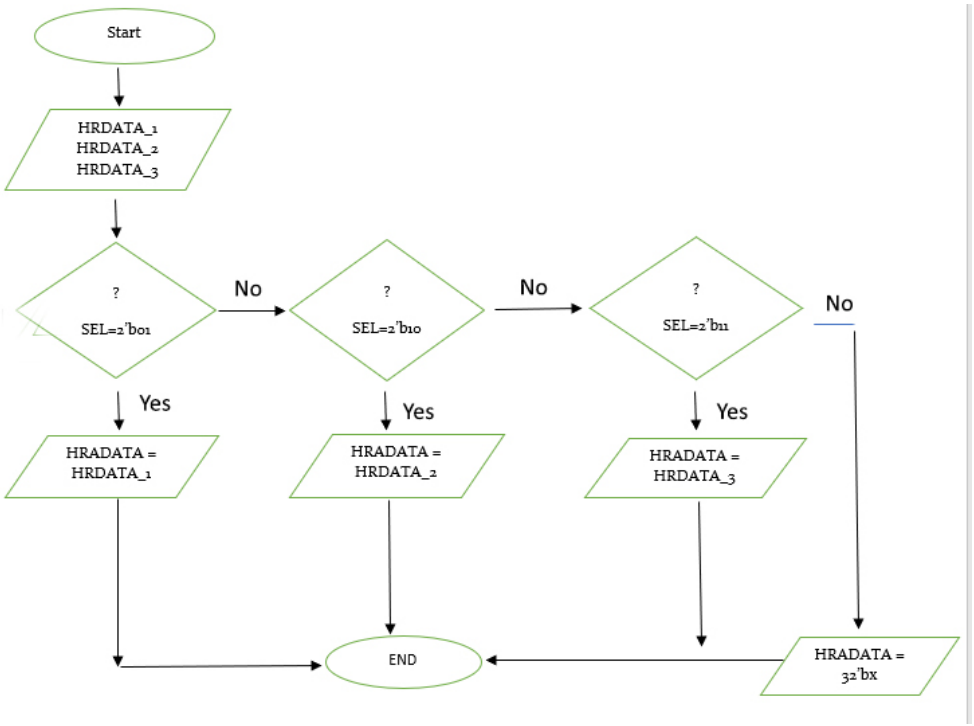

Figure 9: ReadMux Flow Chart

3.3.1 RTL

Verilog Code 3: Test_Read_MUX.v

```
1 `timescale 1ns / 1ps
2
3 module Test_Read_MUX;
4
5   // Inputs
6   reg CLK;
7   reg RST;
8   reg [31:0] HRDATA_1;
```

```
  reg [31:0] HRDATA_2;
  reg [31:0] HRDATA_3;
  reg [1:0] SEL;

  // Outputs
  wire [31:0] HRDATA;

  // Instantiate the Unit Under Test (UUT)
  Read_MUX uut (
    .CLK(CLK),
    .RST(RST),
    .HRDATA(HRDATA),
    .HRDATA_1(HRDATA_1),
    .HRDATA_2(HRDATA_2),
    .HRDATA_3(HRDATA_3),
    .SEL(SEL)
  );

  always #15  CLK =  ! CLK;

  initial begin
  #500 RST = 1; // Random reset stimuli
  #50 RST = 0;
  end

  initial begin
    // Initialize Inputs
    CLK = 0;
    RST = 0;
    HRDATA_1 = 0;
    HRDATA_2 = 0;
    HRDATA_3 = 0;
    SEL = 0;

    // Wait 100 ns for global reset to finish
    #15;

    // Add stimulus here

    #200 HRDATA_1 = $random; HRDATA_2 = $random; HRDATA_3 = $random; SEL = 2'b00; RST = 0;
    #200 /*HRDATA_1 = $random; HRDATA_2 = $random; HRDATA_3 = $random;*/ SEL = 2'b01; RST = 0;
    #200 /*HRDATA_1 = $random; HRDATA_2 = $random; HRDATA_3 = $random;*/ SEL = 2'b10; RST = 0;
    #200 /*HRDATA_1 = $random; HRDATA_2 = $random; HRDATA_3 = $random;*/ SEL = 2'b11; RST = 0;
    #200 /*HRDATA_1 = $random; HRDATA_2 = $random; HRDATA_3 = $random;*/ SEL = 2'b00; RST = 0;

  end

endmodule
```

3.3.2 Test cases and timing diagrams

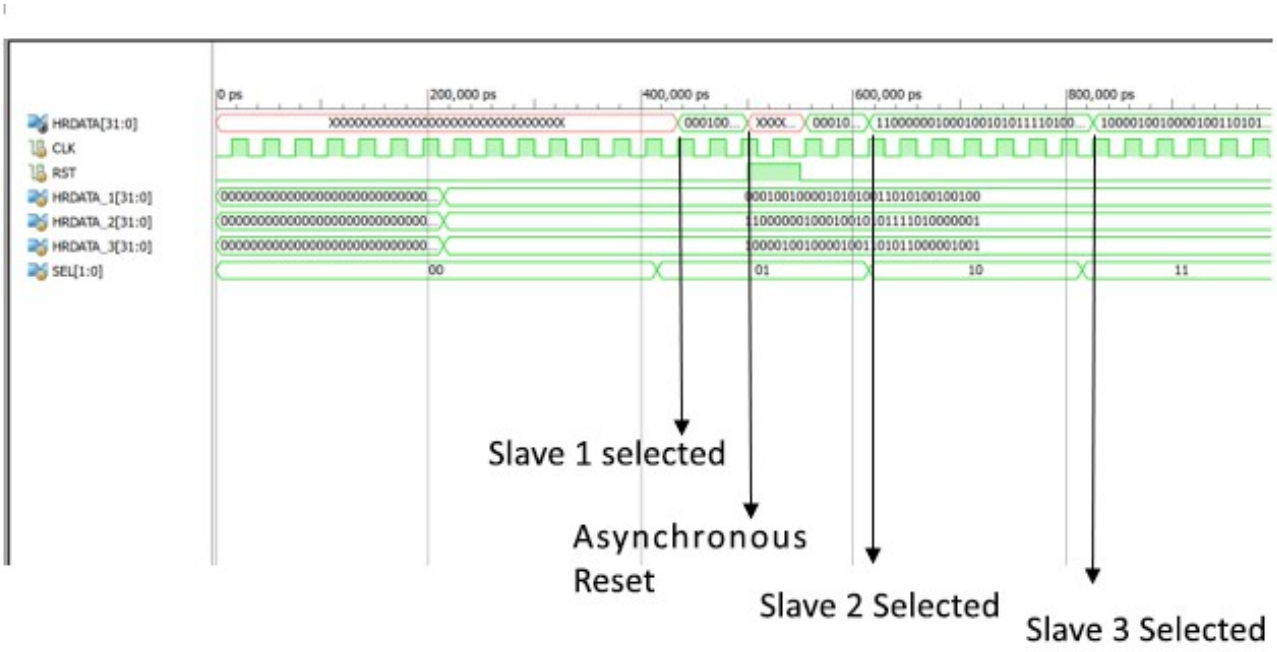

Figure 10: Read Mux Timing Diagram

3.4 Write mux

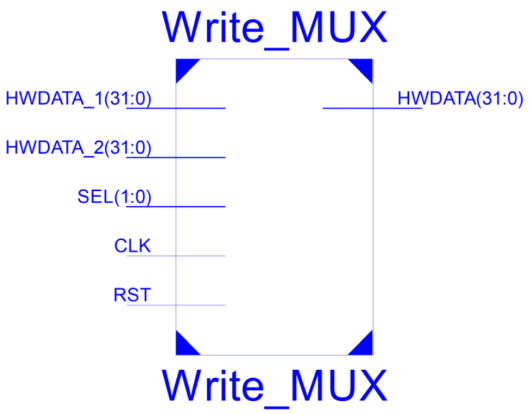

Figure 11: Write Mux Block

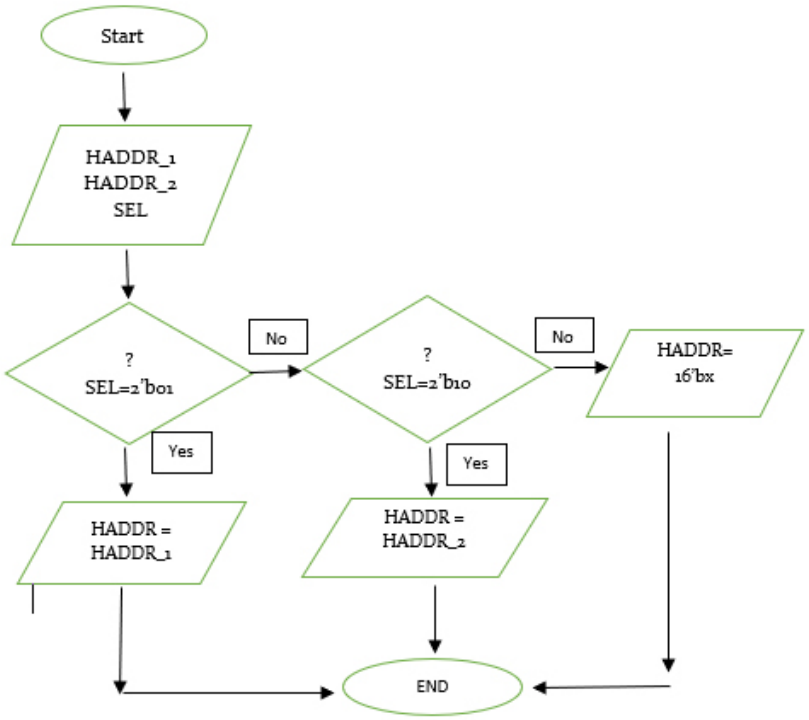

Figure 12: WriteMux Flow Chart

3.4.1 RTL

Verilog Code 4: Test_Write_MUX.v

```
`timescale 1ns / 1ps

module Test_Write_MUX;

  // Inputs
  reg CLK;
  reg RST;
  reg [31:0] HWDATA_1;
  reg [31:0] HWDATA_2;
  reg [1:0] SEL;

  // Outputs
  wire [31:0] HWDATA;

  // Instantiate the Unit Under Test (UUT)
  Write_MUX uut (
    .CLK(CLK),
    .RST(RST),
    .HWDATA(HWDATA),
    .HWDATA_1(HWDATA_1),
    .HWDATA_2(HWDATA_2),
    .SEL(SEL)
  );

  always #15  CLK =  ! CLK;

  initial begin

  #500 RST = 1; // Random reset stimuli
  #50 RST = 0;
  end

  initial begin
    // Initialize Inputs
```

```
35      CLK = 0;
36      RST = 0;
37      HWDATA_1 = 0;
38      HWDATA_2 = 0;
39      SEL = 0;
40
41      // Wait 100 ns for global reset to finish
42      #15;
43
44      // Add stimulus here
45
46      #200 HWDATA_1 = $random; HWDATA_2 = $random; SEL = 2'b00; RST = 0;
47      #200 /*HWDATA_1 = $random; HWDATA_2 = $random;*/ SEL = 2'b01; RST = 0;
48      #200 /*HWDATA_1 = $random; HWDATA_2 = $random;*/ SEL = 2'b10; RST = 0;
49      #200 /*HWDATA_1 = $random; HWDATA_2 = $random;*/ SEL = 2'b11; RST = 0;
50      #200 /*HWDATA_1 = $random; HWDATA_2 = $random;*/ SEL = 2'b00; RST = 0;
51
52    end
53
54  endmodule
```

3.4.2 Test cases and timing diagrams

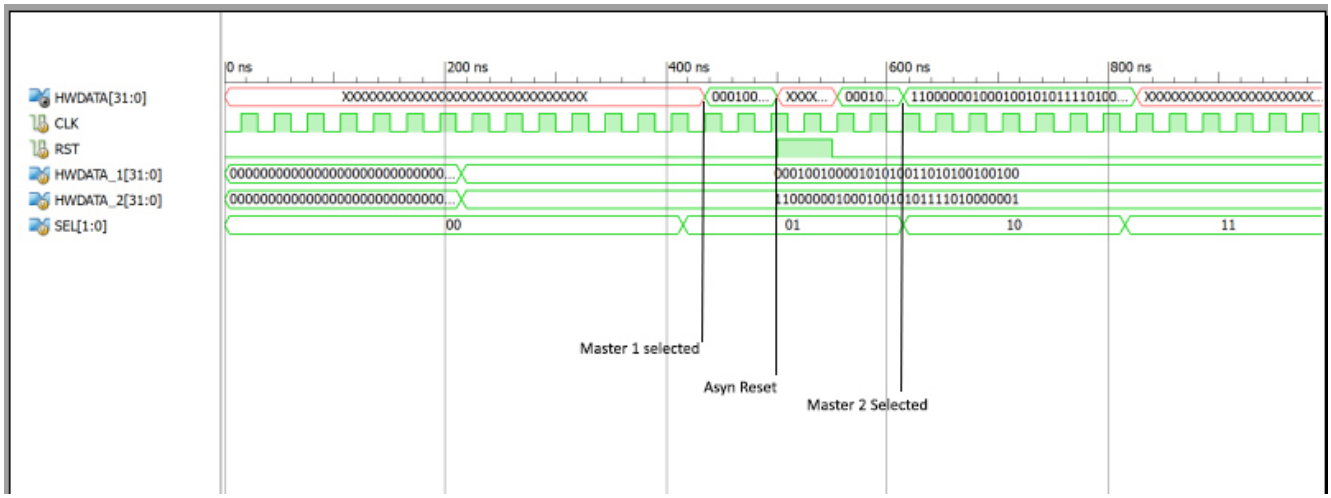

Figure 13: Write Mux Timing Diagram

4 Decoder Module

The decoder is used to decode the address of each transfer and provide a select signal for the slave that is involved in the transfer. It also provides a control signal to the read and response multiplexors. The address decoder or decoder module serve two purposes.

- Capture the 15-bit address
- Decode the slave device address, slave memory address and switch slave.

Decoder Default State: None of the Slaves Selected

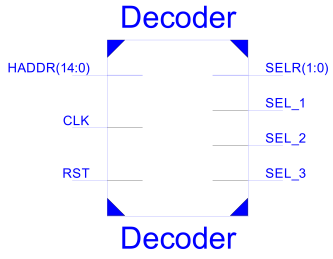

Figure 14: Decoder Block

4.1 Address bus

The 16-bit address bus, HADDR[15:0], provides the address of the transfer. All transfers are memory-mapped and therefore all memory and peripherals within the system must have an address range within which they are accessed. The decoder uses the address bus of first two MSB HADDR[15:0] to determine which bus slave is to be accessed.

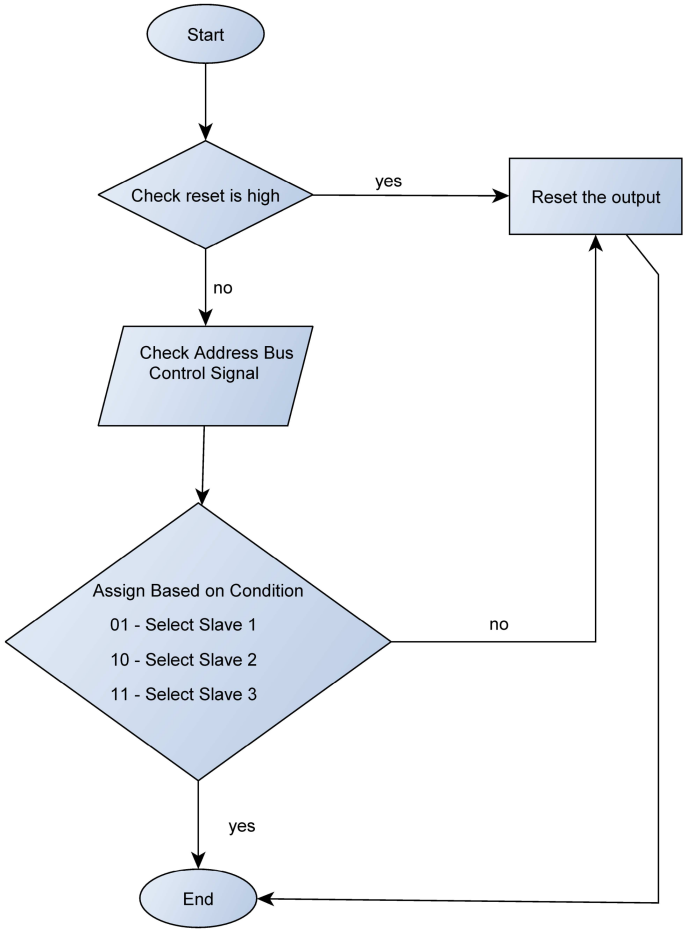

Figure 15: Decoder Flow Chart

4.2 RTL

Verilog Code 5: Test_Decoder.v

```
`timescale 1ns / 1ps

module Test_Decoder;

  // Inputs
  reg RST;
  reg CLK;
  reg [14:0] HADDR;

  // Outputs
  wire SEL_1;
  wire SEL_2;
  wire SEL_3;
  wire [1:0] SELR;

  // Instantiate the Unit Under Test (UUT)
  Decoder uut (
    .RST(RST),
    .CLK(CLK),
    .HADDR(HADDR),
    .SEL_1(SEL_1),
    .SEL_2(SEL_2),
    .SEL_3(SEL_3),
    .SELR(SELR)
  );

  always #15  CLK =  ! CLK;

  initial begin
  #500 RST = 1; // Reset test
  #50 RST = 0;
  end

  initial begin
    // Initialize Inputs
    RST = 0;
    CLK = 0;
    HADDR = 0;

    // Wait 100 ns for global reset to finish
    #100;

    // Add stimulus here
    #150 HADDR = 15'b000000000000000;RST = 0;
    #150 HADDR = 15'b010000000000000;RST = 0; // Slave 1 select
    #150 HADDR = 15'b100000000000000;RST = 0; // Slave 2 select
    #150 HADDR = 15'b110000000000000;RST = 0; // Slave 3 select
    #150 HADDR = 15'b000000000000000;RST = 0;

  end

endmodule
```

4.3 Test cases, Timing diagrams

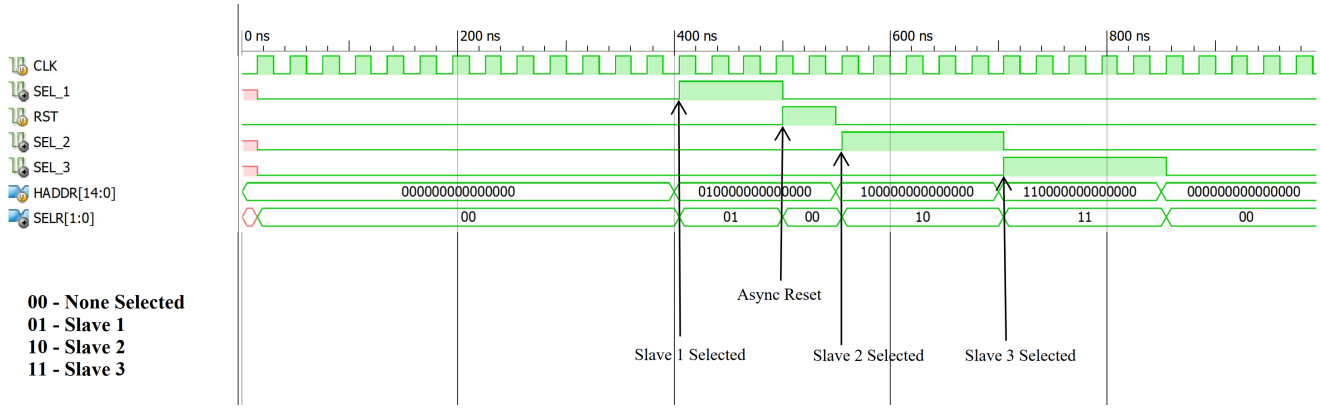

Figure 16: Decoder Timing Diagram

5 Master Design

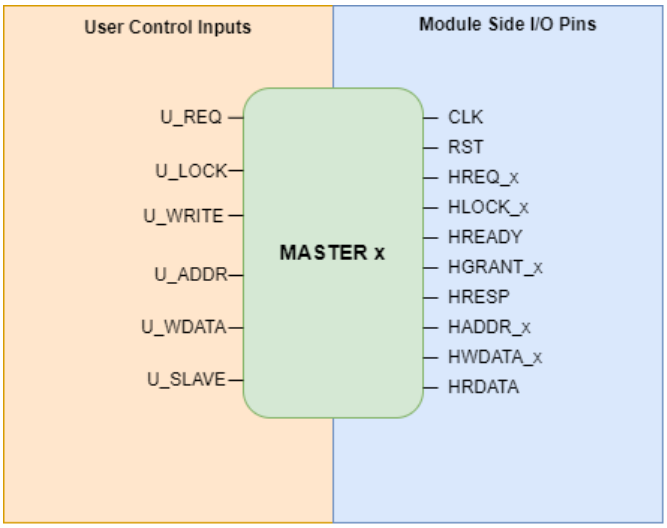

Figure 17: Master Block

The above diagram shows the master pinout structure for this designed bus architecture. To investigate the operation of the bus under certain scenarios, it is necessary for the user to provide inputs to this module so as to emulate the operation of a typical bus master. These control inputs are defined in the table below.

User Control Input	*Description*
U_REQ	Toggled HIGH when user wants Master to prompt arbiter for access to bus.
U_LOCK	Toggled HIGH when user wants Master to perform a locked transaction across the bus. If invoked, the arbiter considers this a priority request. Bus access remains locked until pulled LOW, to release bus.
U_WRITE	Toggled HIGH if user wants Master to perform a WRITE operation across the bus. If LOW, Master will perform a READ operation.
U_ADDR	User must define the target Slave memory address that the Master should deploy on the Address Bus in order to read / write data to Slave. (User must define 2K and 4K address lengths carefully).
U_WDATA	User must specify the 32-bit sized data that must be deployed by the Master in a WRITE operation to the Slave.
U_SLAVE	User must define the Slave ID for the Master to call for access to the target Slave.

Table 1: Master Input and Output Description

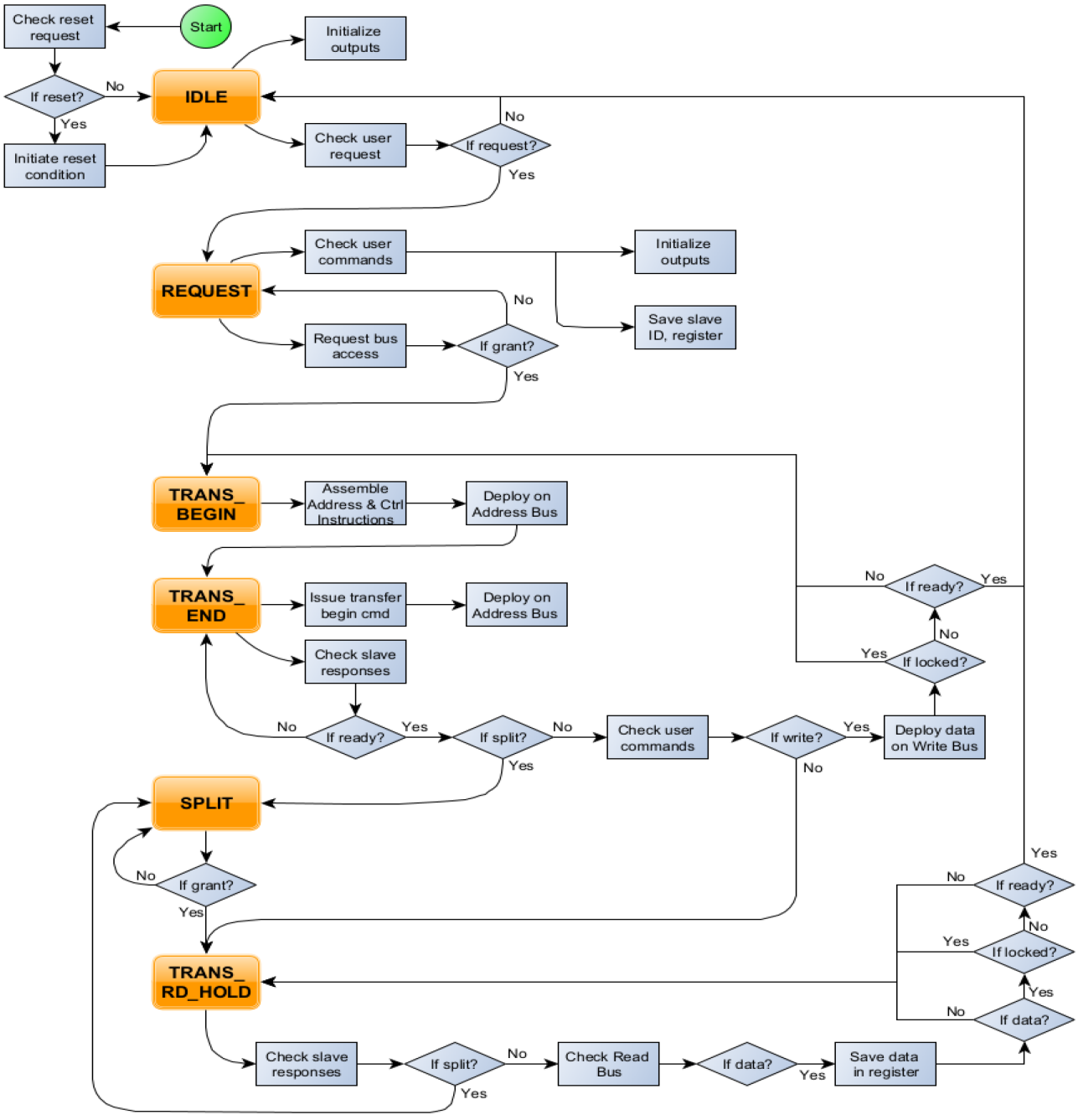

Figure 18: Master State Diagram

5.1 Master Verification

Write, Reset, Read Operation

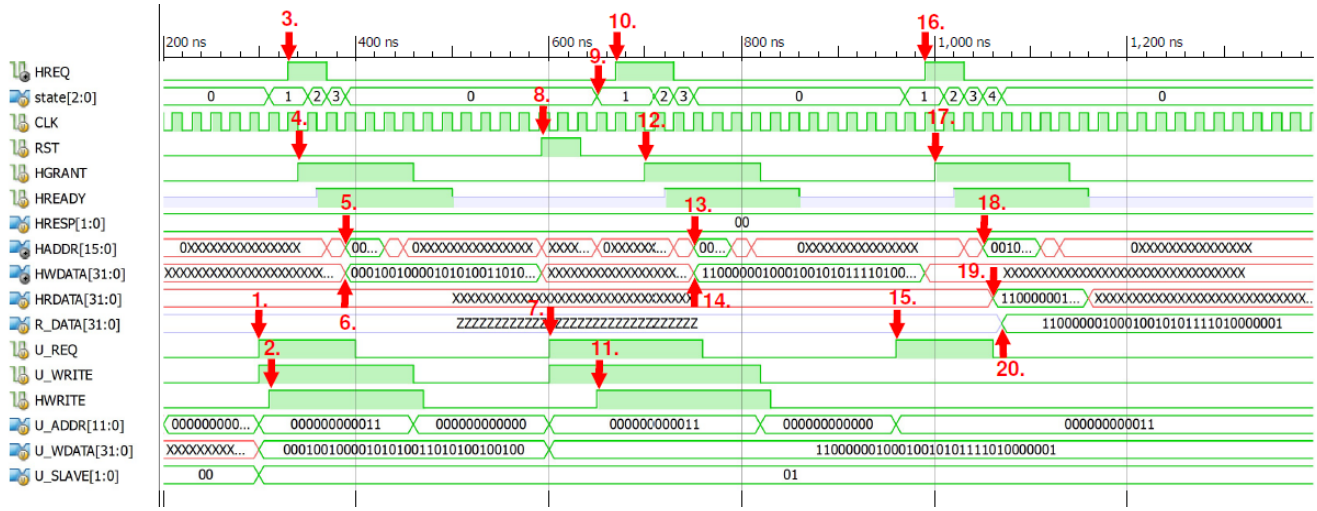

Figure 19: Write, Reset, Read Operation

No	***Operation***
1	User commands Master 1 to prompt Arbiter for bus access.
2	Master toggle HWRITE HIGH to notify of a WRITE impending operation
3	Master requests access from Arbiter, remains HIGH till access granted.
4	Arbiter provides bus access to Master 1.
5	Master deploys target slave device, memory address and control signals on Address Bus.
6	Master deploys the user defined 32-bit data to be written to slave memory on Write Bus.
7	User commands Master 1 to prompt Arbiter for bus access.
8	Reset signal asserted. All temporary registers will be cleared, and Master will return to IDLE state.
9	Reset removed. Master changes to Request state at the clock positive edge, to resume operation.
10	Master requests access from Arbiter, remains HIGH till access granted.
11	Master toggle HWRITE HIGH to notify of a WRITE impending operation
12	Arbiter provides bus access to Master 1.
13	Master deploys target slave device, memory address and control signals on Address Bus..
14	Master deploys the user defined 32-bit data to be written to slave memory on Write Bus.
15	User commands Master 1 to prompt Arbiter for bus access.
16	Master requests access from Arbiter, remains HIGH till access granted.
17	Arbiter provides bus access to Master 1.
18	Master deploys target slave device, memory address and control signals on Address Bus.
19	Master reads data arriving on the Read Bus.
20	Master successfully saves read data to a temporary storage register, R_DATA.

Table 2: Description 1

Write, Split, Read Operation

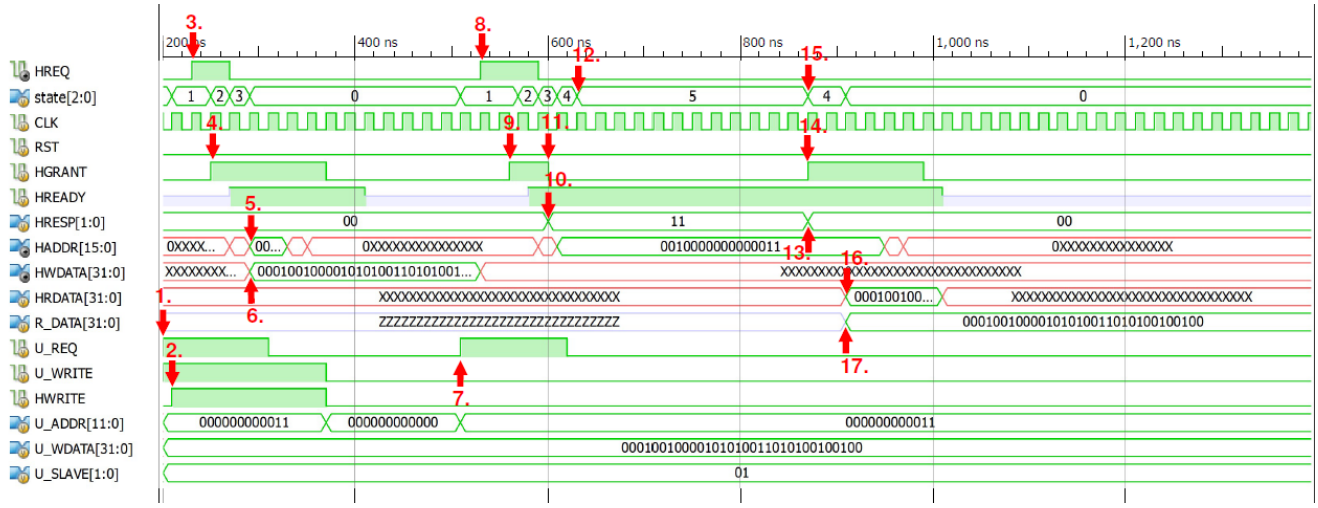

Figure 20: Write, Split, Read Operation

No	*Operation*
1	User commands Master 1 to prompt Arbiter for bus access.
2	Master toggle HWRITE HIGH to notify of a WRITE impending operation
3	Master requests access from Arbiter, remains HIGH till access granted.
4	Arbiter provides bus access to Master 1.
5	Master deploys target slave device, memory address and control signals on Address Bus.
6	Master deploys the user defined 32-bit data to be written to slave memory on Write Bus.
7	User commands Master 1 to prompt Arbiter for bus access.
8	Master requests access from Arbiter, remains HIGH till access granted.
9	Arbiter provides bus access to Master 1.
10	HRESP 2'b11 – Split operation invoked by Slave. .
11	Arbiter terminates bus access to Master 1 when split response invoked.
12	Master goes to Split state, waiting until Arbiter connects it back with Slave to complete the split transaction.
13	Split response revoked by Slave.
14	Arbiter grants access to Master 1 to continue the transaction.
15	Master changes from Split state to Trans_RD_Hold State to continue receiving the read data..
16	Read data, appears on the Read Bus.
17	Arbiter provides bus access to Master 1.
18	Master deploys target slave device, memory address and control signals on Address Bus.
19	Master reads data arriving on the Read Bus.
20	Master successfully saves read data to a temporary storage register, R_DATA. .

Table 3: Description 2

5.2 RTL

Verilog Code 6: Test_F_Master.v

```
1  `timescale 1ns / 1ps
2
3  module Test_F_Master;
4
5    // Inputs
6    reg CLK;
7    reg RST;
8    reg HGRANT;
9    reg HREADY;
10   reg [1:0] HRESP;
```

```
  reg [31:0] HRDATA;
  reg U_REQ;
  reg U_LOCK;
  reg U_WRITE;
  reg [11:0] U_ADDR;
  reg [31:0] U_WDATA;
  reg [1:0] U_SLAVE;

  // Outputs
  wire HREQ;
  wire HLOCK;
  wire [15:0] HADDR;
  wire [31:0] HWDATA;
  wire AB;

  // Instantiate the Unit Under Test (UUT)
  Master uut (
    .CLK(CLK),
    .RST(RST),
    .HGRANT(HGRANT),
    .HREADY(HREADY),
    .HRESP(HRESP),
    .HRDATA(HRDATA),
    .HREQ(HREQ),
    .HLOCK(HLOCK),
    .HADDR(HADDR),
    .HWDATA(HWDATA),
    .AB(AB),
    .U_REQ(U_REQ),
    .U_LOCK(U_LOCK),
    .U_WRITE(U_WRITE),
    .U_ADDR(U_ADDR),
    .U_WDATA(U_WDATA),
    .U_SLAVE(U_SLAVE)
  );

  always #10 CLK=~CLK;
  /*
  initial begin
  #593 RST = 1; // Random reset stimuli
  #40 RST = 0;
  end
*/
  initial begin
    // Initialize Inputs
    CLK = 0;
    RST = 0;
    HGRANT = 0;
    HREADY = 1'bz;
    HRESP = 0;
    HRDATA = 32'bx;
    U_REQ = 0;
    U_LOCK = 0;
    U_WRITE = 0;
    U_ADDR = 16'bx;
    U_WDATA = 32'bx;
    U_SLAVE = 0;

    // Wait 100 ns for global reset to finish
    #100;

    // Add stimulus here
  /*
    #100 U_REQ = 0; U_LOCK = 0; U_WRITE = 0; U_ADDR = 12'b0; U_WDATA = 32'bx; U_SLAVE = 2'b00;
        HGRANT = 0; HRESP = 2'b00; HREADY = 1'bz;

    // Write
    #100 U_REQ = 1; U_LOCK = 0; U_WRITE = 1; U_ADDR = 12'd3; U_WDATA = $random; U_SLAVE = 2'
        b01;
        HGRANT = 1; HRESP = 2'b00; #20 HREADY = 1;

    #40 U_REQ = 0; #60 U_LOCK = 0; U_WRITE = 0; U_ADDR = 12'b0; U_SLAVE = 2'b01;
         HGRANT = 0; HRESP = 2'b00; #40 HREADY = 1'bz;

     // Write with reset
    #100 U_REQ = 1; U_LOCK = 0; U_WRITE = 1; U_ADDR = 12'd3; U_WDATA = $random; U_SLAVE = 2'
```

```
        b01;
86          HGRANT = 1; HRESP = 2'b00; #20 HREADY = 1;
87
88      #40 U_REQ = 0; #60 U_LOCK = 0; U_WRITE = 0; U_ADDR = 12'b0; U_SLAVE = 2'b01;
89           HGRANT = 0; HRESP = 2'b00; #40 HREADY = 1'bz;
90
91      // Read
92      #100 U_REQ = 1; U_LOCK = 0; U_WRITE = 0; U_ADDR = 12'd3; U_SLAVE = 2'b01;
93          HRESP = 2'b00; HGRANT = 1; #20 HREADY = 1;
94
95      #40 U_REQ = 0; U_LOCK = 0; U_WRITE = 0; U_ADDR = 12'd3; U_SLAVE = 2'b01;
96          HRESP = 2'b00; HRDATA = U_WDATA;
97
98      #10 U_REQ = 0; U_LOCK = 0; U_WRITE = 0; U_ADDR = 12'd3; U_SLAVE = 2'b01;
99          HRESP = 2'b00; HREADY = 1;
100
101     #70 HGRANT = 0; #20 HRDATA = 32'bx; HREADY = 1'bz;*/
102
103     // Split
104
105       // Write
106       #100 U_REQ = 1; U_LOCK = 0; U_WRITE = 1; U_ADDR = 12'd3; U_WDATA = $random; U_SLAVE = 2'
              b01;
107           HGRANT = 1; HRESP = 2'b00; #20 HREADY = 1;
108
109       #40 U_REQ = 0; #60 U_LOCK = 0; U_WRITE = 0; U_ADDR = 12'b0; U_SLAVE = 2'b01;
110           HGRANT = 0; HRESP = 2'b00; #40 HREADY = 1'bz;
111
112       // Read
113       #100 U_REQ = 1; U_LOCK = 0; U_WRITE = 0; U_ADDR = 12'd3; U_SLAVE = 2'b01;
114           HRESP = 2'b00; HGRANT = 1; #20 HREADY = 1;
115
116         // Split invoked
117
118         #20 HREADY = 1; HRESP = 2'b11; #20 HGRANT = 0; #20  U_REQ = 0;
119
120         // Split revoked
121
122         #250 HREADY = 1; HRESP = 2'b00; HGRANT = 1;
123
124       #40 U_REQ = 0; U_LOCK = 0; U_WRITE = 0; U_ADDR = 12'd3; U_SLAVE = 2'b01;
125           HRESP = 2'b00; HRDATA = U_WDATA;
126
127       #10 U_REQ = 0; U_LOCK = 0; U_WRITE = 0; U_ADDR = 12'd3; U_SLAVE = 2'b01;
128           HRESP = 2'b00; HREADY = 1;
129
130       #70 HGRANT = 0; #20 HRDATA = 32'bx; HREADY = 1'bz;
131   end
132
133 endmodule
```

Verilog Code 7: Master.v

```
1  `timescale 1ns / 1ps
2
3  //Transfer types
4  `define IDLE 1'b0
5  `define START 1'b1
6  //`define IDLE_TRANS 2'b11
7
8  //Response types
9  `define OKAY 2'b00
10 `define ERROR 2'b01
11 `define RETRY 2'b10
12 `define SPLIT 2'b11
13
14
15 module Master (
16   input CLK,
17   input RST,
18
19   input HGRANT,
20   input HREADY,
21   input [1:0] HRESP,
22   input [31:0] HRDATA,
```

```
23
24    output reg HREQ,
25    output reg HLOCK,
26
27
28    output reg [15:0] HADDR,
29    output reg [31:0] HWDATA,
30    output reg AB,
31
32    //External triggers for testing
33    input U_REQ,
34    input U_LOCK,
35    input U_WRITE,
36    input [11:0] U_ADDR,
37    input [31:0] U_WDATA,
38    input [1:0] U_SLAVE
39
40 );
41
42    parameter [2:0] IDLE = 3'd0;
43    parameter [2:0] REQUEST = 3'd1;
44    parameter [2:0] TRANS_BEGIN = 3'd2;
45    parameter [2:0] TRANS_END = 3'd3;
46    parameter [2:0] TRANS_RD_HOLD = 3'd4;
47    parameter [2:0] SPLIT = 3'd5;
48
49    reg [15:0] ADDR;
50    reg [31:0] R_DATA;
51    reg [2:0] ADDR_INC = 12'd4; //WORD increment
52    reg [11:0] PC; // program counter
53    reg TRANS;
54
55    reg HWRITE;
56
57    reg [2:0] state;
58
59    initial begin
60      AB = 1'bz;
61      state = IDLE; //initially idle state
62      R_DATA = 32'bz;
63    end
64
65 always @ (posedge CLK or posedge RST )
66
67    begin
68      if(RST == 1)
69        begin
70
71        HREQ <= 1'b0;
72        HLOCK <= 1'b0; //master has lock
73        HADDR <= 16'bx;
74        HWDATA <= 32'bx;
75        state <= IDLE;
76
77        end
78
79    else begin
80
81      case(state)
82
83        IDLE: begin
84
85          HREQ <= 1'b0;
86          HLOCK <= 1'b0; //master has lock
87          TRANS <= `IDLE; // in idle state
88          ADDR <= 16'bx;
89          ADDR[15] <= TRANS;
90          HADDR <= ADDR;
91          HWRITE = U_WRITE;
92          state <= (U_REQ == 1)?REQUEST:IDLE;
93
94        end
95
96        REQUEST : begin
97
98          HWDATA <= 32'bx;
```

```
99          HREQ <= U_REQ;
100         HLOCK <= U_LOCK;
101           if(HGRANT) begin
102
103             ADDR[11:0] <= U_ADDR;
104             HREQ <= 1'b1;
105             HLOCK <= U_LOCK; //master has no lock
106             state <= TRANS_BEGIN;
107
108           end
109           else state <= REQUEST;
110       end
111
112       TRANS_BEGIN: begin
113
114         //if(HREADY) begin
115           HREQ <= 1'b0;
116           TRANS <= `IDLE; // non sequence state in trans
117           ADDR[15] <= TRANS;
118           ADDR[14:13] <= U_SLAVE; // Input target slave
119           HWRITE = U_WRITE; // Read / Write CTRL - EXT TRIG
120           //
121           ADDR[12] <= HWRITE;
122           HADDR <= ADDR;
123           PC <= ADDR[11:0] + ADDR_INC; // Program Counter register increment
124           state <= TRANS_END;
125           //#50;
126         //end
127       end
128
129       TRANS_END: begin
130
131         HLOCK <= U_LOCK;
132         TRANS <= `START;
133         ADDR[15] <= TRANS;
134         HADDR <= ADDR;
135         //R_DATA <= 0;
136
137         if(HREADY == 1) begin
138
139             if(U_WRITE == 1)
140               begin
141                 HWDATA <= U_WDATA; //write the data
142                 state <= (HLOCK == 1 || HREADY != 1)?TRANS_BEGIN:IDLE;
143               end
144             else if(U_WRITE == 0)
145               begin
146                 state <= TRANS_RD_HOLD;
147               end
148
149             /*else
150               begin
151                 if(HLOCK == 1)
152                   begin
153                     state <= TRANS_BEGIN;
154                   end
155                 else
156                   begin
157                     state <= (HREADY != 1)?TRANS_BEGIN:IDLE;
158                   end
159               end*/
160         end
161
162         else if(HRESP == `SPLIT) state <= SPLIT;
163
164         else state <= TRANS_END;
165
166         //end
167         //else if(HRESP == `ERROR) state <= IDLE;
168         //else if(HRESP == `RETRY) state <= TRANS_BEGIN;
169         //else if(HRESP == `SPLIT) state <= SPLIT;
170       end
171
172       TRANS_RD_HOLD: begin
173
174         if (HRESP == `SPLIT) state <= SPLIT;
```

```
175          else
176            begin
177              HLOCK <= U_LOCK;
178              R_DATA <= (HRDATA !== 32'bx)?HRDATA:32'bz; //read the data
179              state <= (HLOCK == 1 || HRDATA === 32'bx || HREADY != 1)?TRANS_RD_HOLD:IDLE;
180            end
181        end
182
183        SPLIT: begin
184          if(HGRANT) state <= TRANS_RD_HOLD;
185        end
186      endcase
187   end
188
189 end
190
191 endmodule
```

6 Slave Design

6.1 Slave Address Mapping

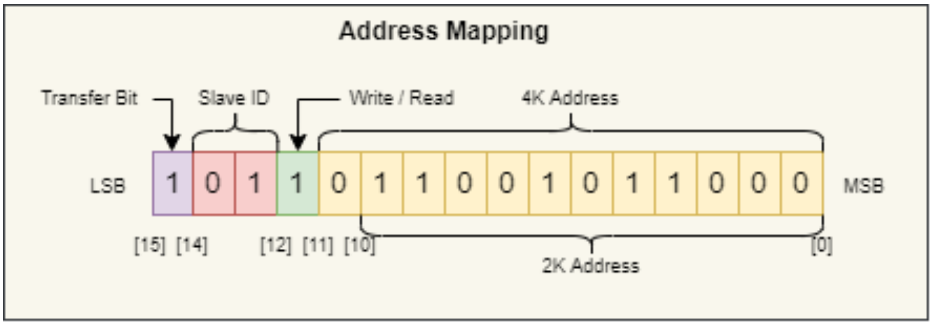

Figure 21: Address Break Down

The above diagram shows the address mapping for the 2K and 4K address mapping. Each 2K slave reads this entire 16-bit address and control data packet and slices the first 11-bits to decode the target memory location. The 4K slave slices the first 12-bits to target its memory location.

6.2 Split Slave

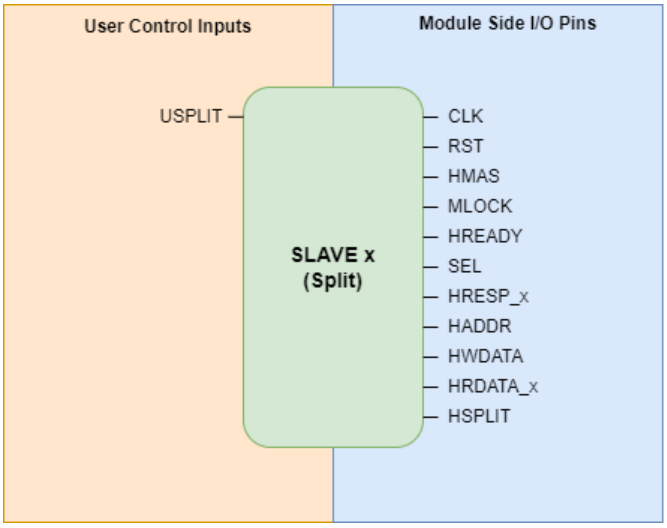

Figure 22: Split Slave Block

The split slave pinout with control inputs is shown above. This module has only a single user control input, used to triggering a split request, to emulate the behaviour of transaction that takes too long to complete in the allocated time. In this bus architecture, the split behaviour is modeled for read operations only, since in a typical scenario, split requests for read operations have higher occurrences due to read operations typically requiring more time to complete than write operations.

User Control Input	***Description***
USPLIT	Toggled HIGH when user wants Slave to issue split request to Arbiter to split current read operation. When toggled LOW, the appropriate Master and Slave are connected back by the Arbiter to resume completion of transaction.

Table 4: Description 3

6.3 Slave (2K / 4K)

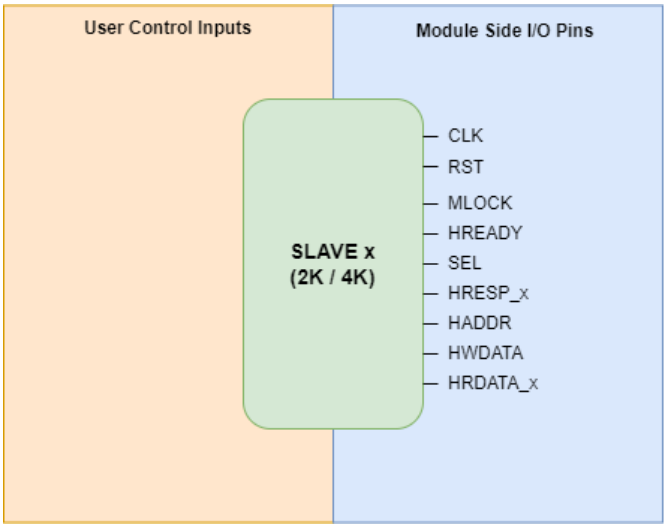

Figure 23: Slave (2K / 4K) Block

The Non-split capable slave modules do not have any user controllable inputs. They have all the functionality of the split slave without the additional split capability.

6.4 Split Slave State Transition Diagram

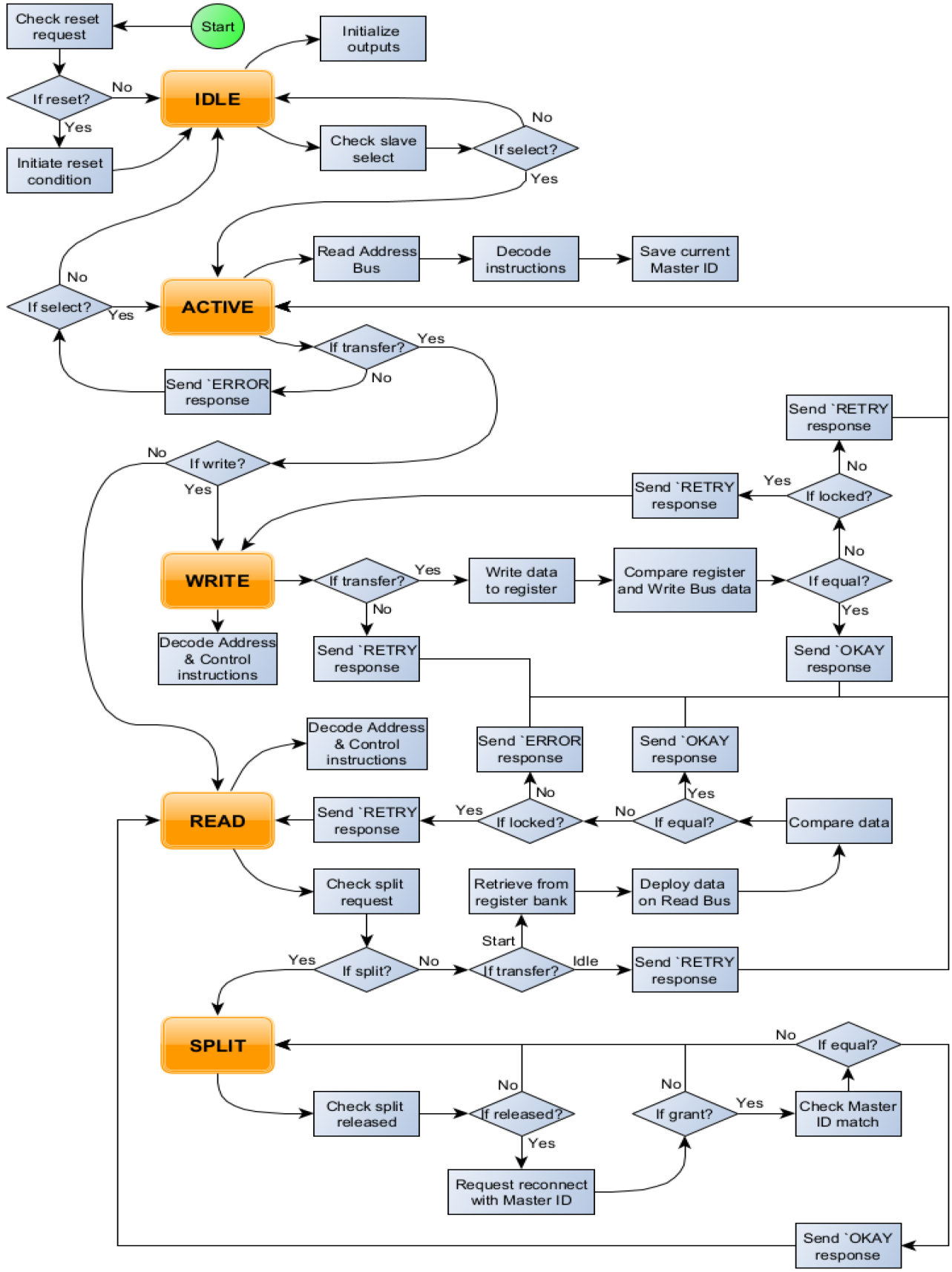

Figure 24: Slave State Diagram

6.5 Slave Verification

Write, Reset, Read Operation

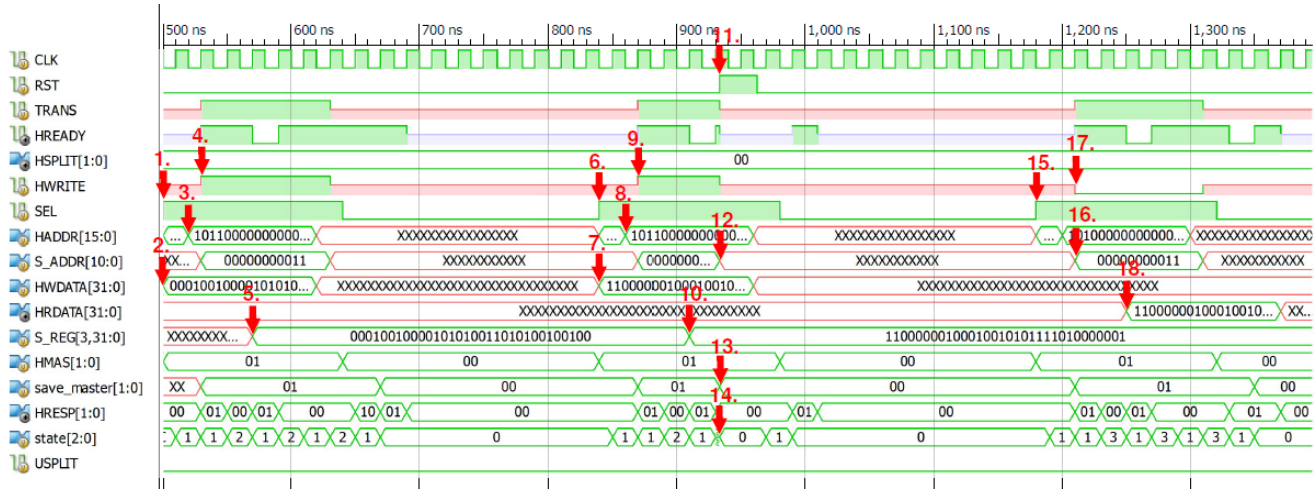

Figure 25: Write, Reset, Read Operation

No	***Operation***
1	SEL pin toggled HIGH to enable Slave.
2	32-bit data arrives on Write Bus.
3	Memory address and control instructions arrive on Address Bus, transfer start signal given.
4	HWRITE toggled HIGH in this operation to specify an impending WRITE operation.
5	Data successfully written to target memory address in Slave.
6	SEL pin toggled HIGH to enable Slave.
7	32-bit data arrives on Write Bus.
8	Memory address and control instructions arrive on Address Bus, transfer start signal given.
9	HWRITE toggled HIGH in this operation to specify an impending WRITE operation.
10	Data successfully written to target memory address in Slave.
11	Reset pin triggered.
12	Temporary register storing target address in Slave cleared on reset. Main memory not cleared.
13	Temporary register storing Master ID cleared on reset.
14	Slave transitions back to IDLE state on reset.
15	SEL pin toggled HIGH to enable Slave.
16	Memory address and control instructions arrive on Address Bus, transfer start signal given.
17	HWRITE toggled LOW in this operation to specify an impending READ operation.
18	Target register data fetched and deployed successfully on Read Bus.

Table 5: Description 4

Write, Split, Read Operation

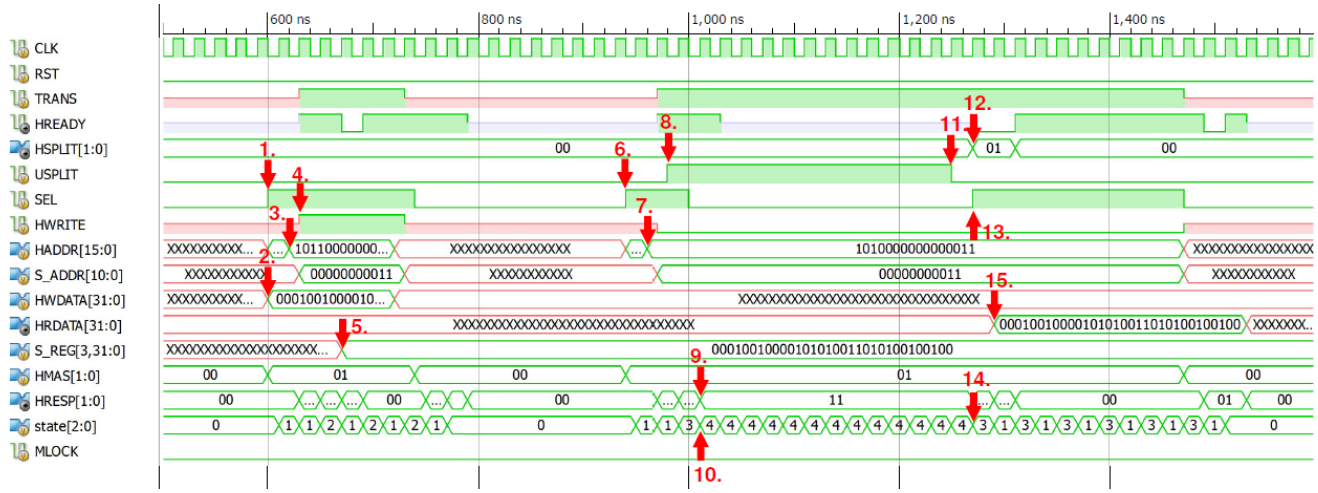

Figure 26: Write, Split, Read Operation

No	*Operation*
1	SEL pin toggled HIGH to enable Slave.
2	32-bit data arrives on Write Bus.
3	Memory address and control instructions arrive on Address Bus, transfer start signal given.
4	HWRITE toggled HIGH in this operation to specify an impending WRITE operation.
5	Data successfully written to target memory address in Slave.
6	SEL pin toggled HIGH to enable Slave.
7	Memory address and control instructions arrive on Address Bus, transfer start signal given.
8	User triggers split command HIGH to slave.
9	Slave response changes to Split response – 2'b11. This line is monitored by Arbiter and Masters.
10	Slave transitions to Split State.
11	User toggles split command LOW to Slave, specifying to resume completion of split transaction.
12	Slave issues Master ID of master that was terminated when performing the split to Arbiter requesting that it be connected back. Split resume requests are given the highest priority in this bus architecture.
13	SEL pin toggled HIGH to enable Slave.
14	Slave transitions back to Read state to resume transaction.
15	Target register data fetched and deployed successfully on Read Bus.

Table 6: Description 5

7 Arbiter Module

The arbiter module ensures that only one bus master at a time is allowed to initiate data transfers. Even though the arbitration protocol is fixed, any arbitration algorithm, such as highest priority or fair access can be implemented depending on the application requirements. The proposed architecture includes only one arbiter, although this would be trivial in single bus master systems.

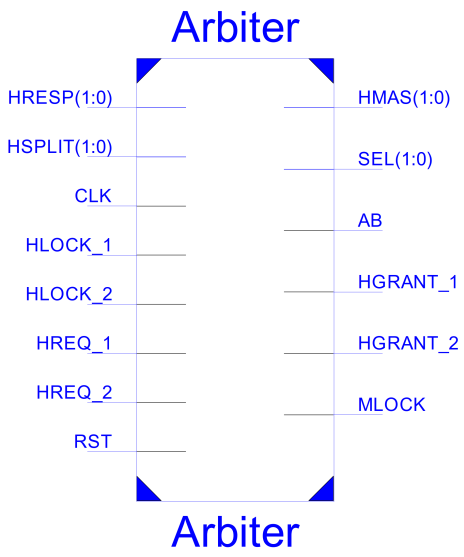

Figure 27: Arbiter Block

7.1 Arbitration

To communicate Master 1 and 2 with Slaves, It should go through the arbiter, Arbiter going to select which master going to connect which slave, write access (not read access) and priority selection.

Each master also generates an HLOCK_x signal which is used to indicate that the master requires exclusive access to the bus. Arbiter in proposed architecture uses HREQ_1, HREQ_2 for normal read write operations, To demostrate priority based, It uses HLOCK_1, HLOCK_2 signals by Master 1 and Master 2 respectively. In case if HREQ_1 & HLOCK_1 is set high at first Master 1 has higher priority than Master 2, As a result HGRANT_1 will be set high.

Priority selection for split slave: **Split request > Locked request > Normal request**

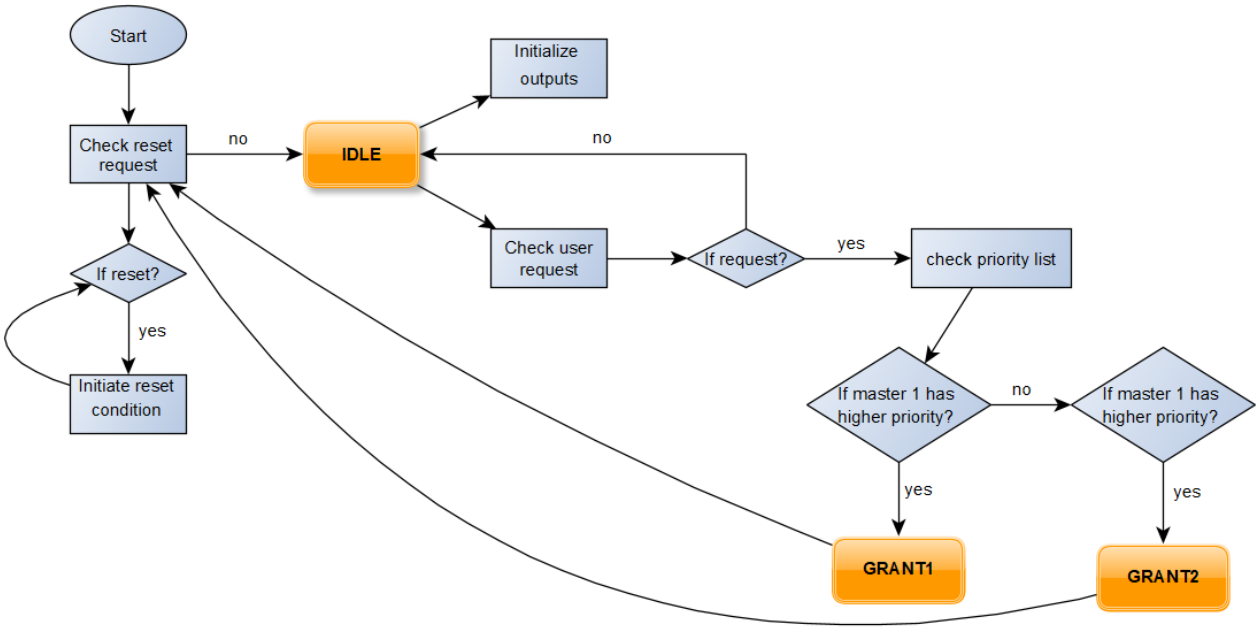

Figure 28: Arbiter State Diagram

7.2 RTL

Verilog Code 8: Arbiter.v

```
1  `timescale 1ns / 1ps
2
3  module Arbiter(
4
5   CLK,
6   RST,
7   HREQ_1,
8   HLOCK_1,
9   HREQ_2,
10  HLOCK_2,
11  HSPLIT,
12  HRESP,
13  HGRANT_1,
14  HGRANT_2,
15  HMAS,
16  MLOCK,
17  SEL,AB,
18  HREADY
19     );
20
21    input CLK;
22    input RST;
23    input HREQ_1;
24    input HLOCK_1;
25    input HREQ_2;
26    input HLOCK_2;
27    input [1:0] HSPLIT;
28    input [1:0] HRESP;
29
30    input HREADY;
31
32    output reg HGRANT_1;
33    output reg HGRANT_2;
34    output reg [1:0] HMAS;
35    output reg MLOCK;
36    output reg [1:0] SEL;
37    output reg AB;
38
39  //-----------------------------
40    parameter SPLIT = 2'b11;
41    parameter IDLE = 2'b00;
42    parameter GRANT1 = 2'b01;
43    parameter GRANT2 = 2'b10;
44
45    reg [1:0]  state;
46    reg g1;
47    reg g2;
48    reg [1:0] grant_save;
49
50
51  initial begin
52    AB = 1'bz;
53    state = IDLE;
54  end
55
56  always@(posedge CLK or posedge RST)
57    begin
58      if(RST == 1)
59        begin
60          HGRANT_1 <= 0;
61          HGRANT_2 <= 0;
62          HMAS <= 0;
63          MLOCK <= 0;
64          SEL <= 2'b00;
65          grant_save <= 0; // saves info about the previous master granted access to the bus
66          state <= IDLE;
67
68        end
69    else
70      begin
71
72        state <= IDLE;
```

```
73
74      case(state)
75
76        IDLE:begin
77
78          HGRANT_1 <= 0;
79          HGRANT_2 <= 0;
80          HMAS <= 0;
81          MLOCK <= 0;
82          SEL <= 2'b00;
83
84  // Priority order = Split request > Locked request > normal request
85  // Master 1 - 01, Master 2 - 10
86
87          if(HSPLIT == 2'b01) // Split request from slave 1 asking master 1 to connect
88            begin
89              state <= GRANT1;
90            end
91          else if(HSPLIT == 2'b10) // split request from slave 1 asking master 2 to connect
92            begin
93              state <= GRANT2;
94            end
95          else if(HREQ_1 == 1 && HREQ_2 == 0) // normal request only from master 1
96            begin
97              state <= GRANT1;
98            end
99          else if(HREQ_1 == 0 && HREQ_2 == 1) // normal request only from master 2
100           begin
101             state <= GRANT2;
102           end
103         else if(HREQ_1 == 1 && HREQ_2 == 1) //Priority selection of masters
104           begin
105             if(HSPLIT == 2'b01 || HLOCK_1) // checks for split request from slave 1 asking
                    master 1 to connect OR a locked request from master 1
106               begin
107                 state <= GRANT1;
108               end
109             else if(HSPLIT == 2'b10 || HLOCK_2) // checks for split request from slave 1
                    asking master 2 to connect OR a locked request from master 2
110               begin
111                 state <= GRANT2;
112               end
113             else // signals with equal normal priority
114               begin
115                 if (grant_save == 2'b01) // variable storing previous master ID
116                   begin
117                     state <= GRANT2;
118                   end
119                 else if(grant_save == 2'b10)
120                   begin
121                     state <= GRANT1;
122                   end
123                 else
124                   begin
125                     state <= GRANT1; // Default grant to master 1 on start
126                   end
127               end
128           end
129         else
130           begin
131             state <= IDLE;
132           end
133         end
134
135
136       GRANT1:begin // access given to Master 1
137
138         HGRANT_1 <= 1;
139         HGRANT_2 <= 0;
140         HMAS <= 2'b01; // unique ID of Master 1 - 01
141         MLOCK <= HLOCK_1;
142         SEL <= 2'b01; // goes to the address and write muxes
143         grant_save <= 2'b01;
144
145
146         if(HRESP == SPLIT) // If split response issued, grant switched to other master
```

```
147             begin
148               state <= GRANT2;
149             end
150           else
151             begin
152               state <= (HLOCK_1 || HRESP == 2'b00 || HREADY == 1'b0)?GRANT1:IDLE; // (some
                      condition)?if_TRUE:if_FALSE Ternery operator
153             end
154           end
155
156
157       GRANT2:begin
158
159         HGRANT_1 <= 0;
160         HGRANT_2 <= 1;
161         HMAS <= 2'b10;
162         MLOCK <= HLOCK_2;
163         SEL <= 2'b10;
164         grant_save <= 2'b10;
165
166         if(HRESP == SPLIT)
167           begin
168             state <= GRANT1;
169           end
170         else
171           begin
172             state <= (HLOCK_2 || HRESP == 2'b00 || HREADY == 1'b0)?GRANT2:IDLE;
173           end
174         end
175     endcase
176   end
177 end
178
179 endmodule
```

Below code is the testbench for the arbiter verification. With that arbiter was verified using Xilinx ISE for following cases.

- Single master request
- Two master requests
- Two master priority requests
- Asynchronous reset test

Verilog Code 9: TEST_ARBITER.v

```
1  `timescale 1ns / 1ps
2
3  module Test_Arbiter;
4
5    // Inputs
6    reg CLK;
7    reg RST;
8    reg HREQ_1;
9    reg HLOCK_1;
10   reg HREQ_2;
11   reg HLOCK_2;
12   reg [1:0] HSPLIT;
13   reg [1:0] HRESP;
14
15   // Outputs
16   wire HGRANT_1;
17   wire HGRANT_2;
18   wire [1:0] HMAS;
19   wire MLOCK;
20   wire [1:0] SEL;
21   wire AB;
22
23   // Instantiate the Unit Under Test (UUT)
24   Arbiter uut (
25     .CLK(CLK),
26     .RST(RST),
```

```
27      .HREQ_1(HREQ_1),
28      .HLOCK_1(HLOCK_1),
29      .HREQ_2(HREQ_2),
30      .HLOCK_2(HLOCK_2),
31      .HSPLIT(HSPLIT),
32      .HRESP(HRESP),
33      .HGRANT_1(HGRANT_1),
34      .HGRANT_2(HGRANT_2),
35      .HMAS(HMAS),
36      .MLOCK(MLOCK),
37      .SEL(SEL),
38      .AB(AB),
39      .HREADY(HREADY)
40    );
41
42    always #15 CLK = !CLK;
43
44    initial begin
45    #400 RST = 1;
46    #50  RST = 0;
47    end
48
49    initial begin
50      // Initialize Inputs
51      CLK = 0;
52      RST = 0;
53
54      // Wait 100 ns for global reset to finish
55
56        #100 HREQ_1 <= 1; HREQ_2 <= 0; HRESP <= 0; HSPLIT <= 0;HLOCK_1 <= 0;HLOCK_2 <= 0; //
              Master 1 request
57
58      #100 HREQ_1 <= 0; HREQ_2 <= 0; HRESP <= 2'b01; HSPLIT <= 0;HLOCK_1 <= 0;HLOCK_2 <= 0;
59
60      #100 HREQ_1 <= 1; HREQ_2 <= 1; HRESP <= 0; HSPLIT <= 0;HLOCK_1 <= 0;HLOCK_2 <= 0; //
            Master 1 & 2 request
61
62      #200 HREQ_1 <= 0; HREQ_2 <= 0; HRESP <= 2'b01; HSPLIT <= 0;HLOCK_1 <= 0;HLOCK_2 <= 0;
63
64      #100 HREQ_1 <= 1; HREQ_2 <= 1; HRESP <= 0; HSPLIT <= 0;HLOCK_1 <= 1;HLOCK_2 <= 0; //
            Master 1 priority request
65
66      #100 HREQ_1 <= 0; HREQ_2 <= 0; HRESP <= 2'b01; HSPLIT <= 0;HLOCK_1 <= 0;HLOCK_2 <= 0;
67
68    end
69
70  endmodule
```

7.3 Test cases, Timing diagrams

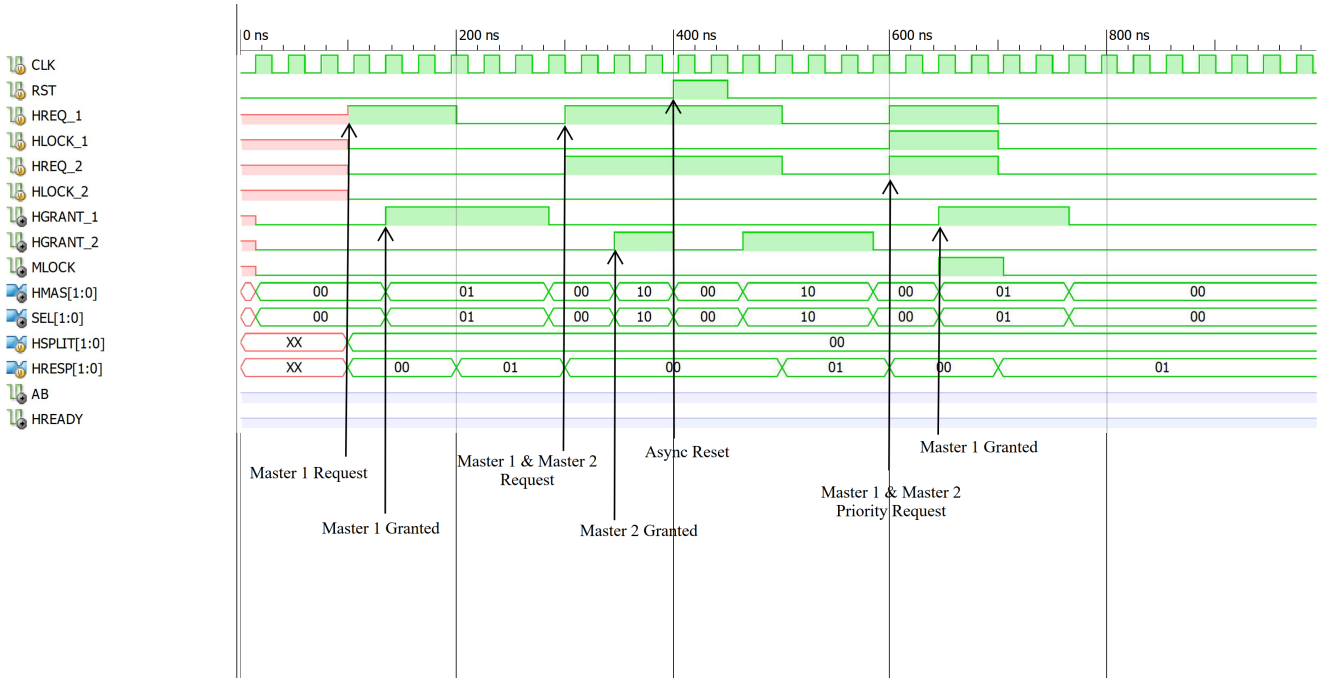

Figure 29: Arbiter Timing Diagram

8 Top Level Verification

8.1 Single Master Request

Write, Reset, Read Operation

Arbiter

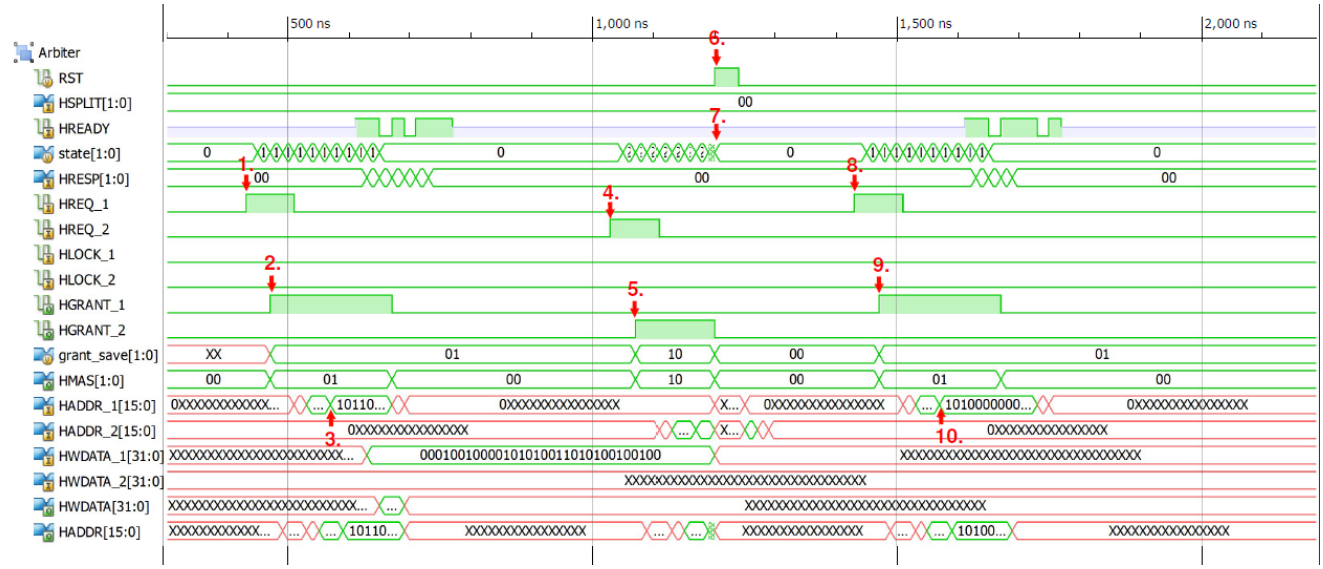

Figure 30: Write, Reset, Read Operation

No	*Operation*
1	Master 1 requests for bus access.
2	Master 1 granted bus access.
3	Master 1 deploys address and control data on Address Bus.
4	Master 2 requests for bus access..
5	Master 2 granted bus access..
6	Reset pin triggered.
7	Arbiter transitions to IDLE state upon reset.
8	Master 1 requests for bus access.
9	Master 1 granted bus access.
10	Master 1 deploys address and control data on Address Bus.

Table 7: Description 6

Master 1

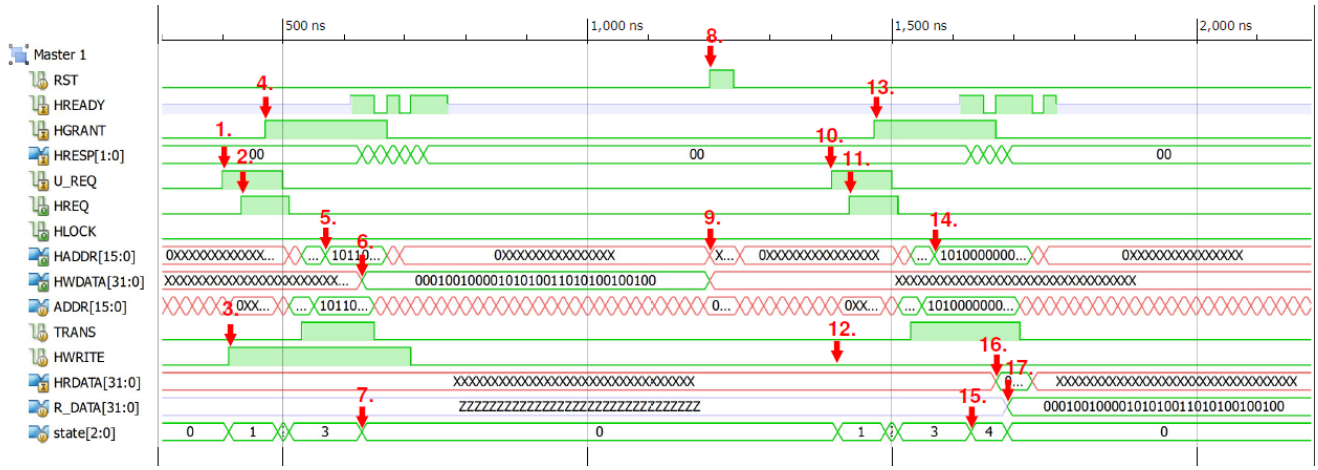

Figure 31: Write, Reset, Read Operation

No	*Operation*
1	User commands Master 1 to prompt Arbiter for bus access.
2	Master 1 requests access from Arbiter, remains HIGH till access granted.
3	Master toggle HWRITE HIGH to notify of a WRITE impending operation
4	Arbiter provides bus access to Master 1.
5	Master deploys target slave device, memory address and control signals on Address Bus.
6	Master deploys the user defined 32-bit data to be written to slave memory on Write Bus.
7	Master state transitions back to IDLE after successful transaction.
8	Reset signal asserted. All temporary registers will be cleared, and Master will return to IDLE state.
9	Address Bus data cleared.
10	User commands Master 1 to prompt Arbiter for bus access.
11	Master requests access from Arbiter, remains HIGH till access granted.
12	Master toggle HWRITE LOW to notify of a READ impending operation
13	Arbiter provides bus access to Master 1.
14	Master deploys target slave device, memory address and control signals on Address Bus.
15	Master state transitions to Trans_RD_HOLD state to wait for read data to show on the Read Bus.
16	Master reads data arriving on the Read Bus.
17	Master successfully saves read data to a temporary storage register, R_DATA.

Table 8: Description 7

Master 2

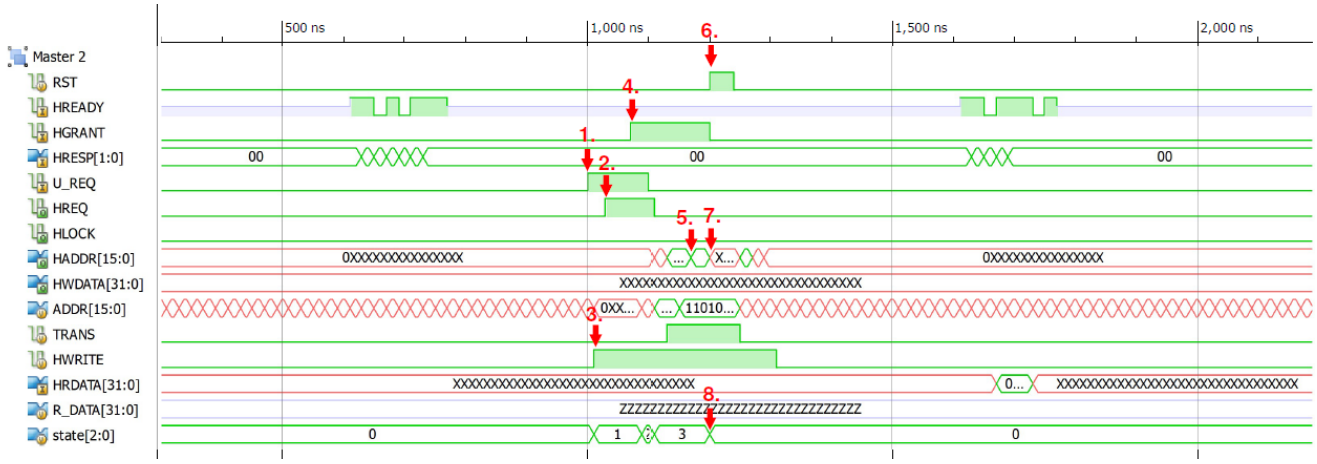

Figure 32: Write, Reset, Read Operation

No	*Operation*
1	User commands Master 2 to prompt Arbiter for bus access.
2	Master 2 requests access from Arbiter, remains HIGH till access granted.
3	Master toggle HWRITE HIGH to notify of a WRITE impending operation
4	Master toggle HWRITE HIGH to notify of a WRITE impending operation
5	Master deploys target slave device, memory address and control signals on Address Bus.
6	Reset signal asserted. All temporary registers will be cleared, and Master will return to IDLE state.
7	Address Bus data cleared.
8	Reset signal asserted. All temporary registers will be cleared, and Master will return to IDLE state.

Table 9: Description 8

Slave 1

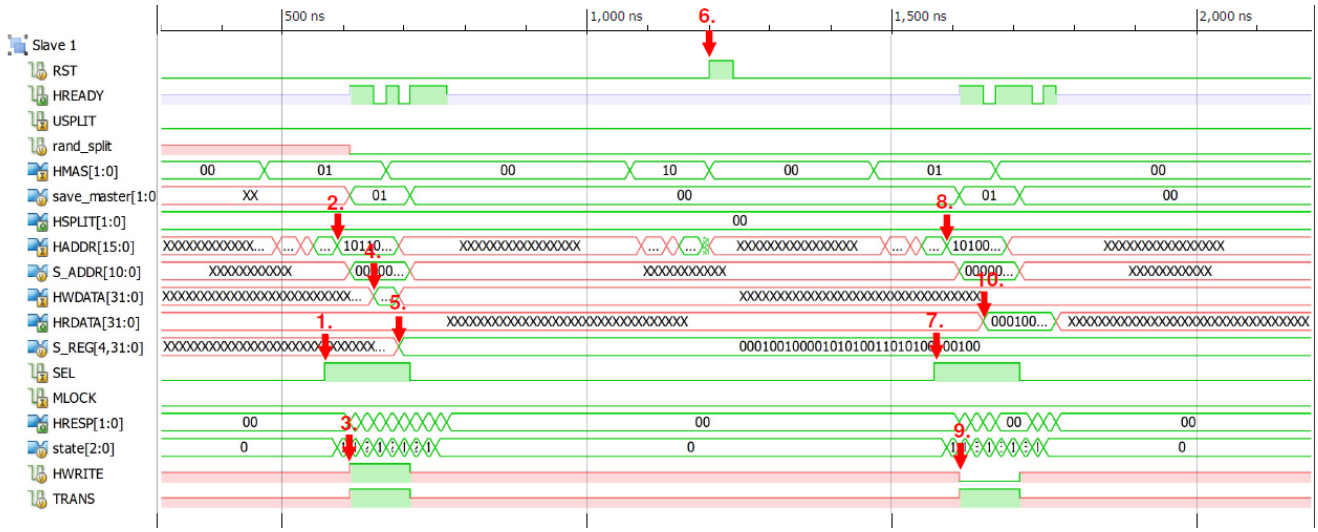

Figure 33: Write, Reset, Read Operation

No	*Operation*
1	SEL pin toggled HIGH to enable Slave.
2	Memory address and control instructions arrive on Address Bus, transfer start signal given.
3	HWRITE toggled HIGH in this operation to specify an impending WRITE operation.
4	32-bit data arrives on Write Bus.
5	Data successfully written to target memory address in Slave.
6	Reset pin triggered.
7	SEL pin toggled HIGH to enable Slave.
8	Memory address and control instructions arrive on Address Bus, transfer start signal given.
9	HWRITE toggled LOW in this operation to specify an impending READ operation.
10	Target register data fetched and deployed successfully on Read Bus.

Table 10: Description 9

Slave 2

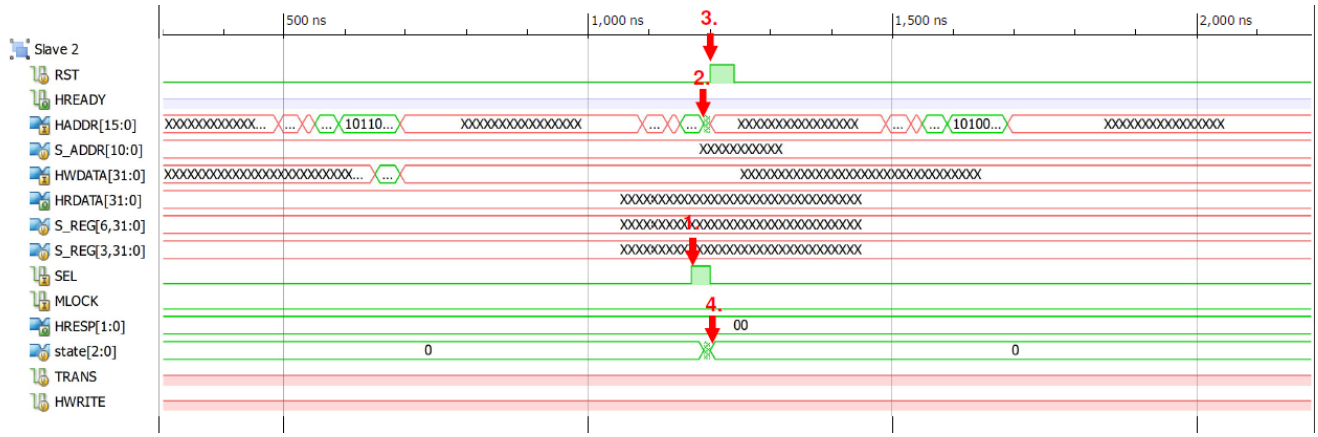

Figure 34: Write, Reset, Read Operation

No	*Operation*
1	SEL pin toggled HIGH to enable Slave.
2	Memory address and control instructions arrive on Address Bus, transfer start signal given.
3	Reset pin triggered.
4	Reset signal asserted. All temporary registers will be cleared, and Slave will return to IDLE state.

Table 11: Description 10

8.2 Two Master Requests, with Priority

Write Operation

Arbiter

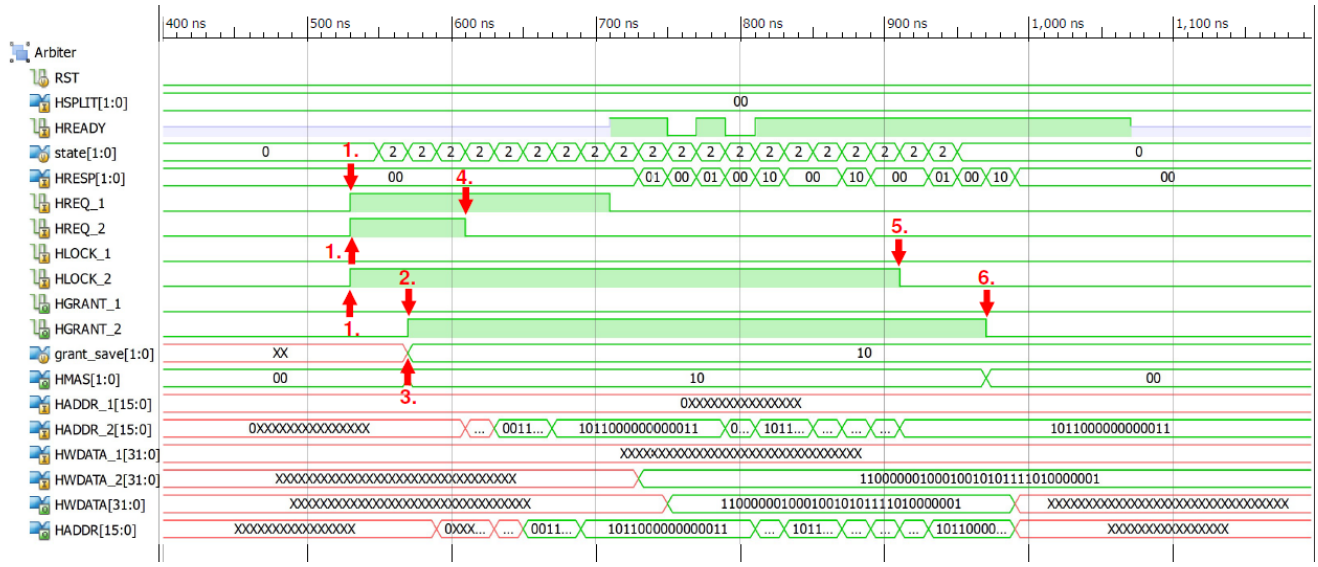

Figure 35: Write Operation

No	*Operation*
1	Master 1, Master 2 requests for bus access. Master 2 requests locked priority access.
2	Master 2 granted bus access.
3	Temporary register stores Master ID of current master given access to the bus.
4	Master 2 request signal pulled LOW once Arbiter grants access..
5	Master 2 releases lock of bus.
6	Arbiter removes bus access to Master 2 once lock pulled LOW.

Table 12: Description 11

Master 1

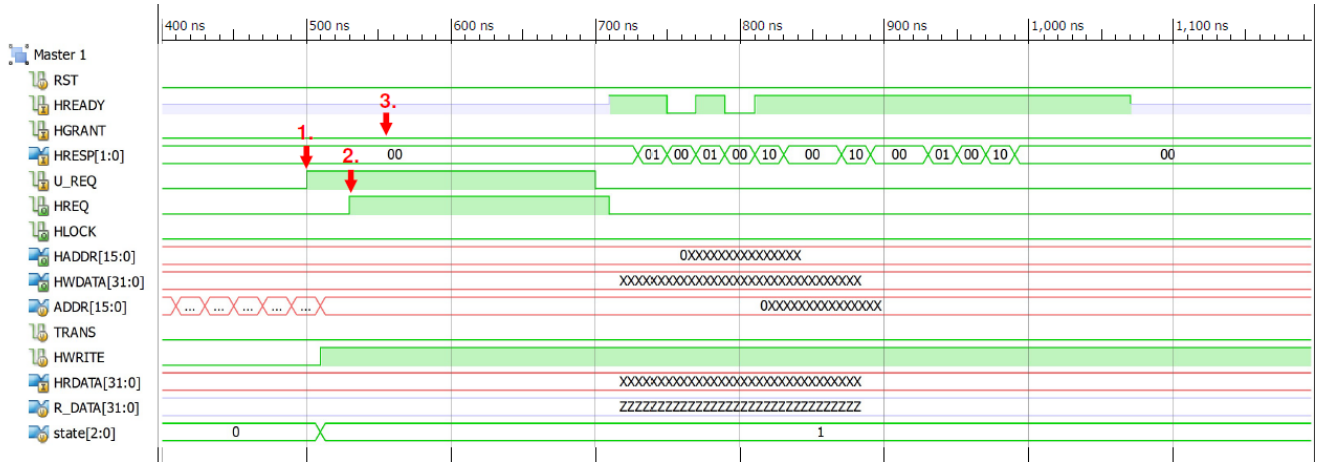

Figure 36: Write Operation

No	*Operation*
1	User commands Master 1 to request access for bus from Arbiter.
2	Master 1 requests for bus access.
3	Arbiter denies granting bus access to Master 1.

Table 13: Description 12

Master 2

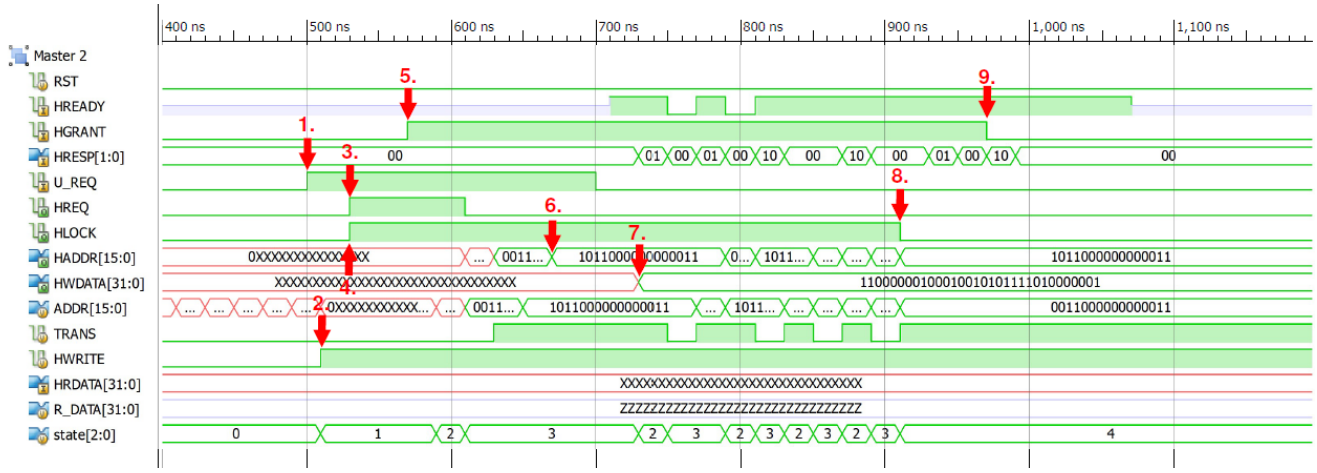

Figure 37: Write Operation

No	*Operation*
1	User commands Master 2 to prompt Arbiter for bus access.
2	Master toggle HWRITE HIGH to notify of a WRITE impending operation
3	Master 2 requests access from Arbiter, remains HIGH till access granted.
4	Master 2 requests for locked access to bus. Lock toggled HIGH.
5	Arbiter provides bus access to Master 2.
6	Master deploys target slave device, memory address and control signals on Address Bus.
7	Master deploys the user defined 32-bit data to be written to slave memory on Write Bus.
8	Lock toggled LOW, releasing access to bus.
9	Arbiter removes bus access to Master 2 once lock pulled LOW.

Table 14: Description 13

Slave 1

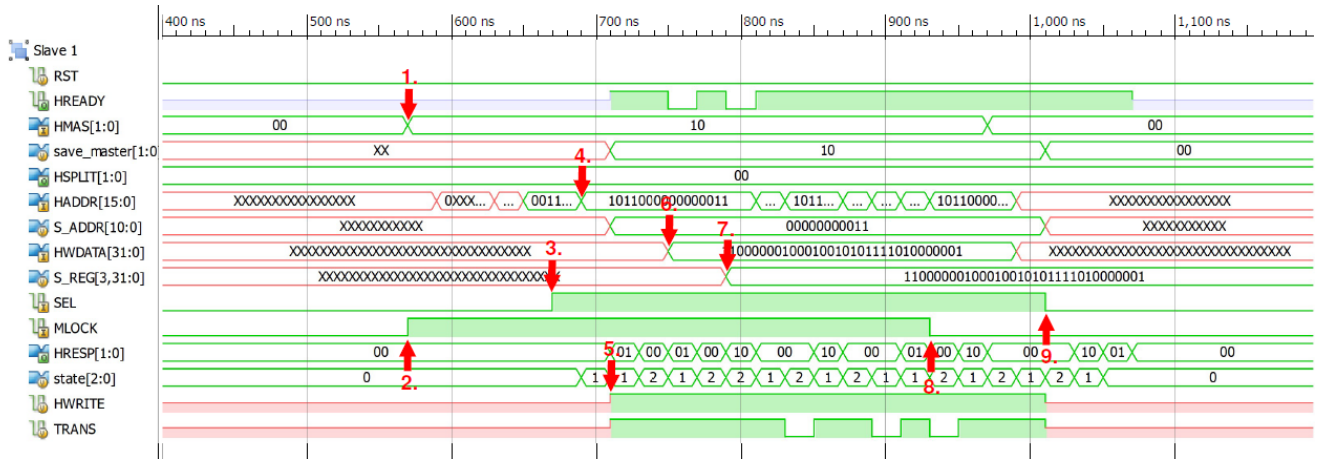

Figure 38: Write Operation

No	*Operation*
1	Master 2 given control of bus. Arbiter send Master ID to Slave 1.
2	Master 2 locks the bus for transaction. Arbiter tells Slave that the bus is currently performing a locked operation.
3	SEL pin toggled HIGH to enable Slave.
4	Memory address and control instructions arrive on Address Bus, transfer start signal given.
5	HWRITE toggled HIGH in this operation to specify an impending WRITE operation.
6	32-bit data arrives on Write Bus.
7	Data successfully written to target memory address in Slave.
8	Master 2 releases lock on bus.
9	Arbiter removes grant access to Master 2. Slave 1 is disabled by toggling SEL, LOW.

Table 15: Description 14

8.3 Split Transaction

Write, Split, Write, Read Operation

Arbiter

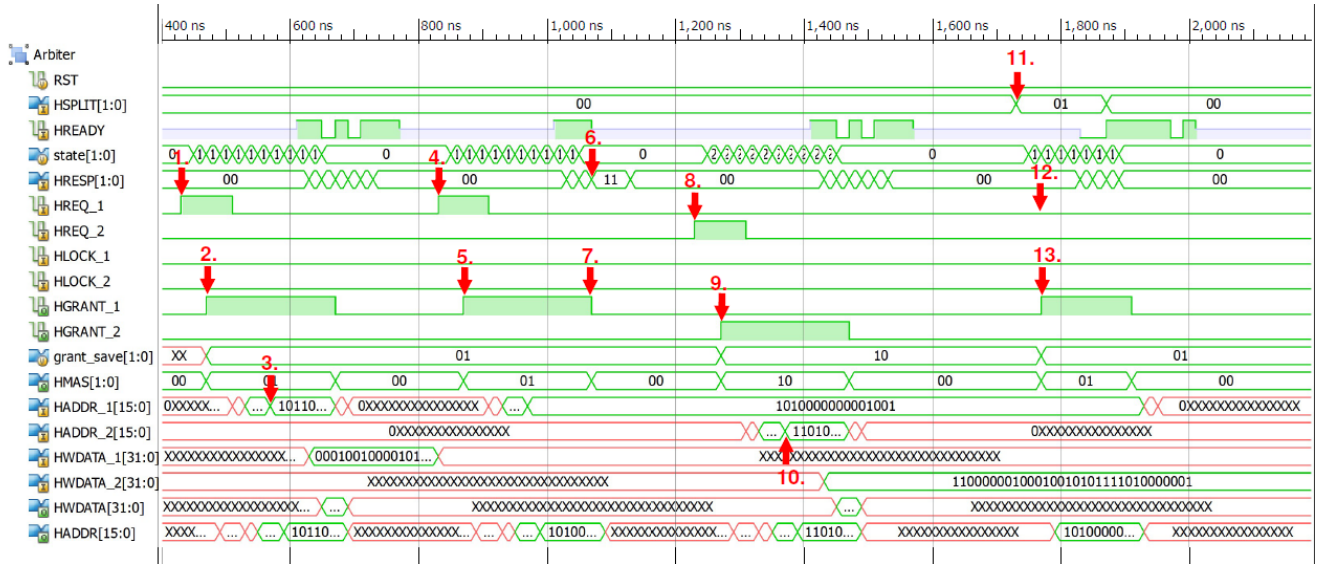

Figure 39: Write, Split, Write, Read Operation

No	*Operation*
1	Master 1 requests for bus access.
2	Master 1 granted bus access.
3	Master 1 deploys address and control data on Address Bus.
4	Master 1 requests for bus access.
5	Master 1 granted bus access.
6	Split response triggered
7	Arbiter revokes bus access to Master 1.
8	Master 2 requests for bus access.
9	Master 2 granted bus access.
10	Master 1 deploys address and control data on Address Bus.
11	Split Slave 1 sends reconnect request with Master ID to Arbiter, to complete split transaction..
12	Master 1 does not request for bus access.
13	Master 1 granted bus access as prompted by Slave 1 to complete split transaction.

Table 16: Description 15

Master 1

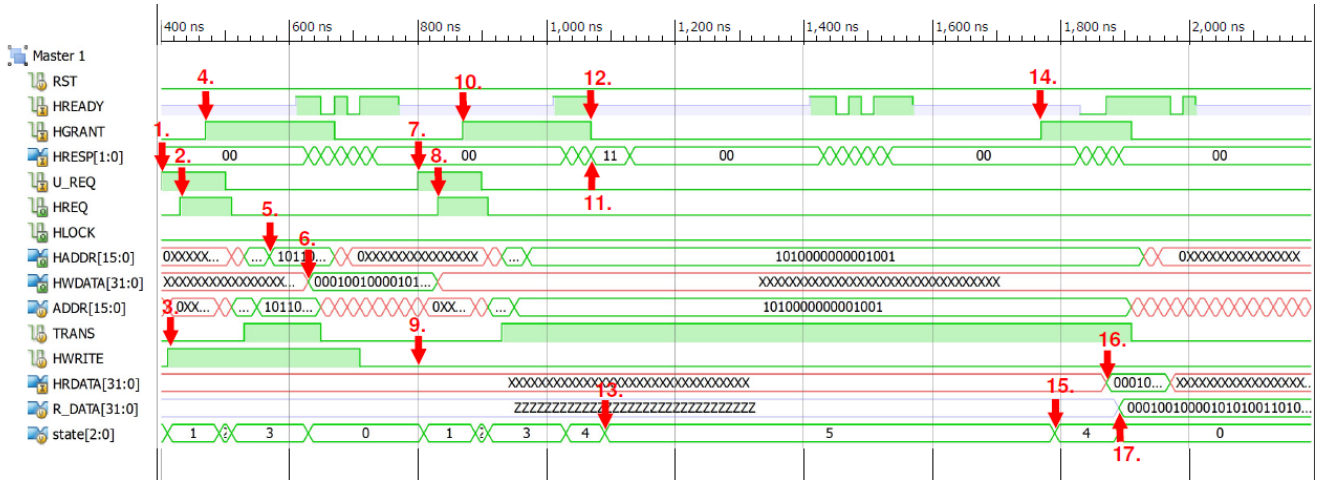

Figure 40: Write, Split, Write, Read Operation

No	*Operation*
1	User commands Master 1 to prompt Arbiter for bus access.
2	Master 1 requests access from Arbiter, remains HIGH till access granted.
3	Master toggle HWRITE HIGH to notify of a WRITE impending operation
4	Arbiter provides bus access to Master 1.
5	Master deploys target slave device, memory address and control signals on Address Bus.
6	Master deploys the user defined 32-bit data to be written to slave memory on Write Bus.
7	User commands Master 1 to prompt Arbiter for bus access.
8	Master 1 requests access from Arbiter, remains HIGH till access granted.
9	Master toggle HWRITE HIGH to notify of a WRITE impending operation
10	Arbiter provides bus access to Master 1.
11	Split response triggered
12	Arbiter revokes bus access to Master 1.
13	Master 1 transitions states to Split state.
14	Master 1 granted bus access as prompted by Slave 1 to complete split transaction.
15	Master 1 transitions back to Trans_RD_HOLD state to await read data arriving from Slave 1.
16	Master reads data arriving on the Read Bus.
17	Master successfully saves read data to a temporary storage register, R_DATA.

Table 17: Description 16

Master 2

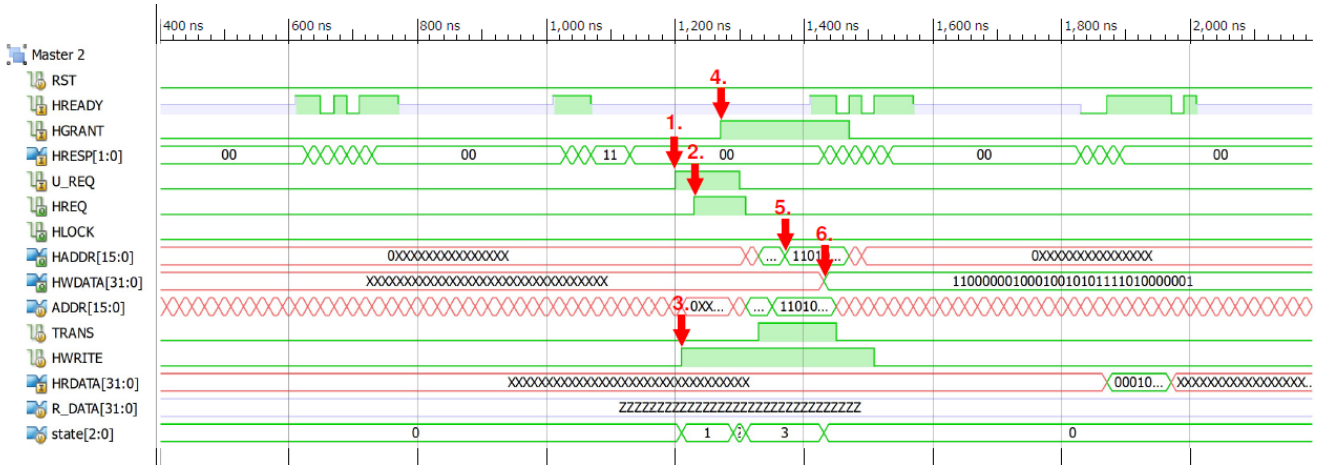

Figure 41: Write, Split, Write, Read Operation

No	*Operation*
1	User commands Master 2 to prompt Arbiter for bus access.
2	Master 2 requests access from Arbiter, remains HIGH till access granted.
3	Master toggle HWRITE HIGH to notify of a WRITE impending operation
4	Arbiter provides bus access to Master 2.
5	Master deploys target slave device, memory address and control signals on Address Bus.
6	Master deploys the user defined 32-bit data to be written to slave memory on Write Bus.

Table 18: Description 17

Slave 1

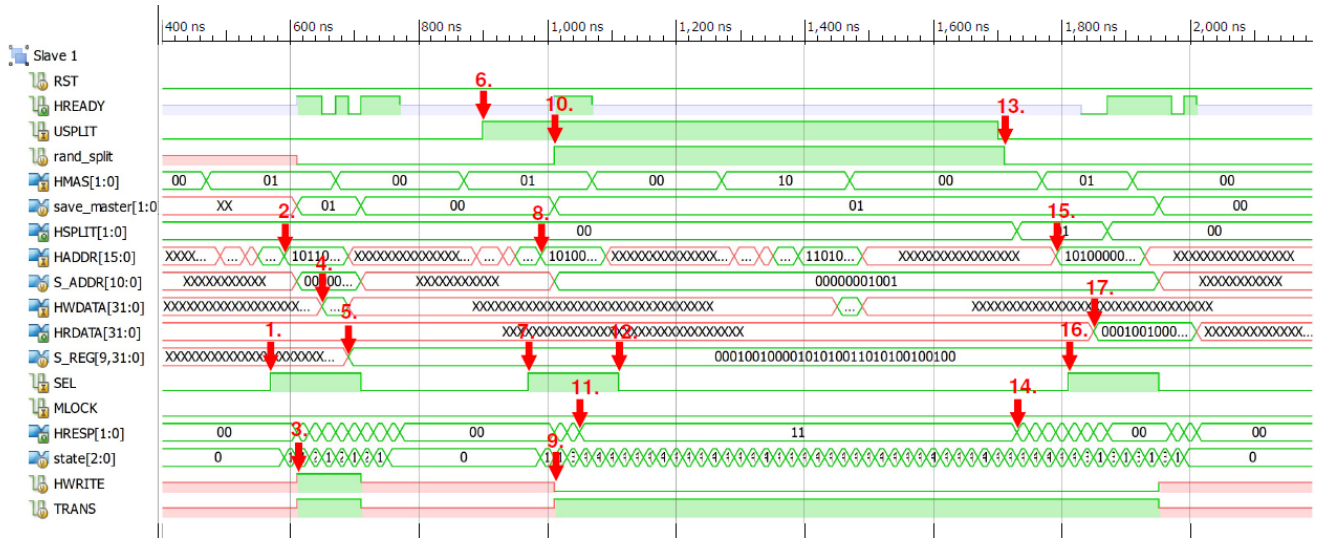

Figure 42: Write, Split, Write, Read Operation

No	*Operation*
1	SEL pin toggled HIGH to enable Slave.
2	Memory address and control instructions arrive on Address Bus, transfer start signal given.
3	HWRITE toggled HIGH in this operation to specify an impending WRITE operation.
4	32-bit data arrives on Write Bus.
5	Data successfully written to target memory address in Slave.
6	User triggers split command HIGH to slave.
7	SEL pin toggled HIGH to enable Slave.
8	Memory address and control instructions arrive on Address Bus, transfer start signal given.
9	HWRITE toggled LOW in this operation to specify an impending READ operation.
10	User invoked split command registered in Slave temporary address.
11	Slave response changes to Split response – 2'b11. This line is monitored by Arbiter and Masters.
12	Arbiter revokes bus access to Slave 1. Slave disabled as SEL pulled LOW.
13	User toggles split command LOW to Slave, specifying to resume completion of split transaction.
14	Slave response changes from split, as it changes states from Split state.
15	Memory address and control instructions arrive on Address Bus, transfer start signal given.
16	SEL pin toggled HIGH to enable Slave.
17	Target register data fetched and deployed successfully on Read Bus.

Table 19: Description 18

Slave 2

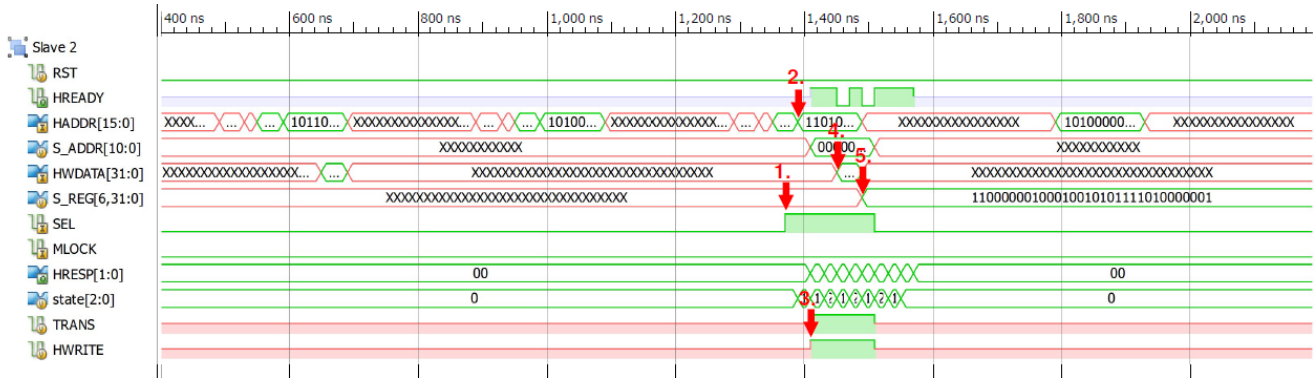

Figure 43: Write, Split, Write, Read Operation

No	*Operation*
1	SEL pin toggled HIGH to enable Slave.
2	Memory address and control instructions arrive on Address Bus, transfer start signal given.
3	HWRITE toggled HIGH in this operation to specify an impending WRITE operation.
4	32-bit data arrives on Write Bus.
5	Data successfully written to target memory address in Slave.

Table 20: Description 19

8.4 RTL

Verilog Code 10: TEST_TOP_MODULE.v

```
1  `timescale 1ns / 1ps
2
3  module Test_TOP_MODULE;
4
5    // Inputs
6    reg RST;
7    reg CLK;
8    reg U1_REQ;
```

```
  reg U1_LOCK;
  reg U1_WRITE;
  reg [11:0] U1_ADDR;
  reg [31:0] U1_WDATA;
  reg [1:0] U1_SLAVE;
  reg U2_REQ;
  reg U2_LOCK;
  reg U2_WRITE;
  reg [11:0] U2_ADDR;
  reg [31:0] U2_WDATA;
  reg [1:0] U2_SLAVE;
  reg U_SPLIT;

  // Outputs
  wire AB;

  // Instantiate the Unit Under Test (UUT)
  TOP_MODULE uut (
    .RST(RST),
    .CLK(CLK),
    .U1_REQ(U1_REQ),
    .U1_LOCK(U1_LOCK),
    .U1_WRITE(U1_WRITE),
    .U1_ADDR(U1_ADDR),
    .U1_WDATA(U1_WDATA),
    .U1_SLAVE(U1_SLAVE),
    .U2_REQ(U2_REQ),
    .U2_LOCK(U2_LOCK),
    .U2_WRITE(U2_WRITE),
    .U2_ADDR(U2_ADDR),
    .U2_WDATA(U2_WDATA),
    .U2_SLAVE(U2_SLAVE),
    .U_SPLIT(U_SPLIT),
    .AB(AB)
  );

  always #10 CLK=~CLK;
/*
  initial begin
  #1200 RST = 1; // Random reset stimuli
  #40 RST = 0;
  end
*/

  initial begin
    // Initialize Inputs
    RST = 0;
    CLK = 0;
    U1_REQ = 0;
    U1_LOCK = 0;
    U1_WRITE = 0;
    U1_ADDR = 0;
    U1_WDATA = 0;
    U1_SLAVE = 0;
    U2_REQ = 0;
    U2_LOCK = 0;
    U2_WRITE = 0;
    U2_ADDR = 0;
    U2_WDATA = 0;
    U2_SLAVE = 0;
    U_SPLIT = 0;

    // Wait 100 ns for global reset to finish
    #100;

    // Add stimulus here

    //#200;

//---One Master Request with Write/Read-------

  /*  // WRITE from Master 1 to Slave 1
    #200 U1_REQ = 0; U1_LOCK = 0; U1_WRITE = 0; U1_ADDR = 12'b0; U1_WDATA = 32'b0; U1_SLAVE =
        2'b00;

    #100 U1_REQ = 1; U1_LOCK = 0; U1_WRITE = 1; U1_ADDR = 12'd4; U1_WDATA = $random; U1_SLAVE
```

```
        = 2'b01;

    #100 U1_REQ = 0; U1_LOCK = 0; U1_WRITE = 1; U1_ADDR = 12'd4; U1_SLAVE = 2'b01;

    #200 U1_REQ = 0; U1_LOCK = 0; U1_WRITE = 0; U1_ADDR = 12'd0; U1_WDATA = 32'b0; U1_SLAVE =
        2'b00;

    // WRITE from Master 2 to Slave 2
    #200 U2_REQ = 0; U2_LOCK = 0; U2_WRITE = 0; U2_ADDR = 12'b0; U2_WDATA = 32'b0; U2_SLAVE =
        2'b00;

    #100 U2_REQ = 1; U2_LOCK = 0; U2_WRITE = 1; U2_ADDR = 12'd4; U2_WDATA = $random; U2_SLAVE
        = 2'b10;

    #100 U2_REQ = 0; U2_LOCK = 0; U2_WRITE = 1; U2_ADDR = 12'd4; U2_SLAVE = 2'b10;

    #200 U2_REQ = 0; U2_LOCK = 0; U2_WRITE = 0; U2_ADDR = 12'd0; U2_WDATA = 32'b0; U2_SLAVE =
        2'b00;

    // READ from Slave 1 to Master 1
    #100 U1_REQ = 1; U1_LOCK = 0; U1_WRITE = 0; U1_ADDR = 12'd4;  U1_SLAVE = 2'b01;

    #100 U1_REQ = 0; U1_LOCK = 0; U1_WRITE = 0; U1_ADDR = 12'd4;  U1_SLAVE = 2'b01;

    #100 U1_REQ = 0; U1_LOCK = 0; U1_WRITE = 0; U1_ADDR = 12'd4;  U1_SLAVE = 2'b00;
  */
//---Two Master Request with equal priorities---------

  /*  #200 U1_REQ = 0; U2_REQ = 0; U1_LOCK = 0; U2_LOCK = 0; U1_WRITE = 0; U2_WRITE = 0;
      U1_ADDR = 12'b0;
        U2_ADDR = 12'b0; U1_WDATA = 32'b0; U2_WDATA = 32'b0; U1_SLAVE = 2'b00; U2_SLAVE = 2'
            b00;

    // Master 1 + Master 2 request and write to Slave 1 (Master 1 granted access)
    #200 U1_REQ = 1; U2_REQ = 1; U1_LOCK = 0; U2_LOCK = 0; U1_WRITE = 1; U2_WRITE = 1; U1_ADDR
         = 12'd1;
        U2_ADDR = 12'd3; U1_WDATA = $random; U2_WDATA = $random; U1_SLAVE = 2'b01; U2_SLAVE =
            2'b01;

    #200 U1_REQ = 0; U2_REQ = 0; U1_LOCK = 0; U2_LOCK = 0; U1_WRITE = 1; U2_WRITE = 0; U1_ADDR
         = 12'd1;
        U2_ADDR = 12'b0; U1_SLAVE = 2'b01; U2_SLAVE = 2'b01;

     #200 U1_REQ = 0; U2_REQ = 0; U1_LOCK = 0; U2_LOCK = 0; U1_WRITE = 0; U2_WRITE = 0;
         U1_ADDR = 12'b0;
        U2_ADDR = 12'b0; U1_WDATA = 32'b0; U2_WDATA = 32'b0; U1_SLAVE = 2'b00; U2_SLAVE = 2'
            b00;

    // Master 1 + Master 2 request and write to Slave 1 (Master 2 granted access)
    #200 U1_REQ = 1; #20 U2_REQ = 1; U1_LOCK = 0; U2_LOCK = 0; U1_WRITE = 1; U2_WRITE = 1;
        U1_ADDR = 12'd1;
        U2_ADDR = 12'd3; U1_WDATA = $random; U2_WDATA = $random; U1_SLAVE = 2'b01; U2_SLAVE =
            2'b01;

     #200 U1_REQ = 0; U2_REQ = 0; U1_LOCK = 0; U2_LOCK = 0; U1_WRITE = 0; U2_WRITE = 1;
         U1_ADDR = 12'd1;
        U2_ADDR = 12'b0; U1_SLAVE = 2'b01; U2_SLAVE = 2'b01;

    #200 U1_REQ = 0; U2_REQ = 0; U1_LOCK = 0; U2_LOCK = 0; U1_WRITE = 0; U2_WRITE = 0; U1_ADDR
         = 12'b0;
        U2_ADDR = 12'b0; U1_WDATA = 32'b0; U2_WDATA = 32'b0; U1_SLAVE = 2'b00; U2_SLAVE = 2'
            b00;
  */
//---Two Master Request with different priorities---------

  /*  #200 U1_REQ = 0; U2_REQ = 0; U1_LOCK = 0; U2_LOCK = 0; U1_WRITE = 0; U2_WRITE = 0;
      U1_ADDR = 12'b0;
        U2_ADDR = 12'b0; U1_WDATA = 32'b0; U2_WDATA = 32'b0; U1_SLAVE = 2'b00; U2_SLAVE = 2'
            b00;

    // Master 1 + Master 2 request and write to Slave 1 (Master 2 HLOCK invoked priority)
    #200 U1_REQ = 1; U2_REQ = 1; U1_LOCK = 0; U2_LOCK = 1; U1_WRITE = 1; U2_WRITE = 1; U1_ADDR
         = 12'd1;
        U2_ADDR = 12'd3; U1_WDATA = $random; U2_WDATA = $random; U1_SLAVE = 2'b01; U2_SLAVE =
            2'b01;

```

```
139      #200 U1_REQ = 0; U2_REQ = 0; U1_LOCK = 0; U2_LOCK = 1; U1_WRITE = 0; U2_WRITE = 1;
             U1_ADDR = 12'd1;
140          U2_ADDR = 12'b0; U1_SLAVE = 2'b01; U2_SLAVE = 2'b01;
141
142      #200 U1_REQ = 0; U2_REQ = 0; U1_LOCK = 0; U2_LOCK = 0; U1_WRITE = 0; U2_WRITE = 0; U1_ADDR
             = 12'b0;
143          U2_ADDR = 12'b0; U1_WDATA = 32'b0; U2_WDATA = 32'b0; U1_SLAVE = 2'b00; U2_SLAVE = 2'
                 b00;
144    */
145  //---Split Transaction-----
146
147      // WRITE to Master 1
148      #200 U1_REQ = 0; U1_LOCK = 0; U1_WRITE = 0; U1_ADDR = 12'b0; U1_WDATA = 32'b0; U1_SLAVE =
             2'b00;
149
150      #100 U1_REQ = 1; U1_LOCK = 0; U1_WRITE = 1; U1_ADDR = 12'd9; U1_WDATA = $random; U1_SLAVE
             = 2'b01;
151
152      #100 U1_REQ = 0; U1_LOCK = 0; U1_WRITE = 1; U1_ADDR = 12'd9; U1_SLAVE = 2'b01;
153
154      #200 U1_REQ = 0; U1_LOCK = 0; U1_WRITE = 0; U1_ADDR = 12'd0; U1_WDATA = 32'b0; U1_SLAVE =
             2'b00;
155
156      // READ from Master 1
157      #100 U1_REQ = 1; U1_LOCK = 0; U1_WRITE = 0; U1_ADDR = 12'd9;  U1_SLAVE = 2'b01;
158
159      #100 U1_REQ = 0; U1_LOCK = 0; U1_WRITE = 0; U1_ADDR = 12'd9;  U1_SLAVE = 2'b01; U_SPLIT =
             1;
160
161      //---- SPLIT invoked
162
163        // WRITE from Master 2 to Slave 2
164        #200 U2_REQ = 0; U2_LOCK = 0; U2_WRITE = 0; U2_ADDR = 12'b0; U2_WDATA = 32'b0; U2_SLAVE
               = 2'b00;
165
166        #100 U2_REQ = 1; U2_LOCK = 0; U2_WRITE = 1; U2_ADDR = 12'd6; U2_WDATA = $random;
               U2_SLAVE = 2'b10;
167
168        #100 U2_REQ = 0; U2_LOCK = 0; U2_WRITE = 1; U2_ADDR = 12'd6; U2_SLAVE = 2'b10;
169
170        #200 U2_REQ = 0; U2_LOCK = 0; U2_WRITE = 0; U2_ADDR = 12'd0; U2_WDATA = 32'b0; U2_SLAVE
               = 2'b00;
171
172      //---- SPLIT resumed
173
174      #200 U1_REQ = 0; U1_LOCK = 0; U1_WRITE = 0; U1_ADDR = 12'd9;  U1_SLAVE = 2'b01; U_SPLIT =
             0;
175
176      #100 U1_REQ = 0; U1_LOCK = 0; U1_WRITE = 0; U1_ADDR = 12'd9;  U1_SLAVE = 2'b00;
177
178      #200 U1_REQ = 0; U2_REQ = 0; U1_LOCK = 0; U2_LOCK = 0; U1_WRITE = 0; U2_WRITE = 0; U1_ADDR
             = 12'b0;
179          U2_ADDR = 12'b0; U1_WDATA = 32'b0; U2_WDATA = 32'b0; U1_SLAVE = 2'b00; U2_SLAVE = 2'
                 b00;
180
181    /*
182      // Master 1 + Master 2 request and write to Slave 1 (Master 2 granted access)
183      #200 U1_REQ = 1; U2_REQ = 1; U1_LOCK = 0; U2_LOCK = 0; U1_WRITE = 1; U2_WRITE = 1; U1_ADDR
             = 12'd1;
184          U2_ADDR = 12'd3; U1_WDATA = $random; U2_WDATA = $random; U1_SLAVE = 2'b01; U2_SLAVE =
                 2'b10;
185
186      #200 U1_REQ = 0; U2_REQ = 0; U1_LOCK = 0; U2_LOCK = 0; U1_WRITE = 1; U2_WRITE = 1; U1_ADDR
             = 12'd1;
187          U2_ADDR = 12'b0; U1_SLAVE = 2'b01; U2_SLAVE = 2'b10;
188
189       #200 U1_REQ = 0; U2_REQ = 0; U1_LOCK = 0; U2_LOCK = 0; U1_WRITE = 0; U2_WRITE = 0;
             U1_ADDR = 12'b0;
190          U2_ADDR = 12'b0; U1_WDATA = 32'b0; U2_WDATA = 32'b0; U1_SLAVE = 2'b00; U2_SLAVE = 2'
                 b00;
191
192       // Master 1 + Master 2 request and write to Slave 1 (Master 1 granted access)
193      #200 U1_REQ = 1; U2_REQ = 1; U1_LOCK = 0; U2_LOCK = 0; U1_WRITE = 1; U2_WRITE = 1; U1_ADDR
             = 12'd1;
194          U2_ADDR = 12'd3; U1_WDATA = $random; U2_WDATA = $random; U1_SLAVE = 2'b01; U2_SLAVE =
                 2'b10;
```

```
195
196     #200 U1_REQ = 0; U2_REQ = 0; U1_LOCK = 0; U2_LOCK = 0; U1_WRITE = 1; U2_WRITE = 1; U1_ADDR
            = 12'd1;
197         U2_ADDR = 12'b0; U1_SLAVE = 2'b01; U2_SLAVE = 2'b10;
198
199      #200 U1_REQ = 0; U2_REQ = 0; U1_LOCK = 0; U2_LOCK = 0; U1_WRITE = 0; U2_WRITE = 0;
            U1_ADDR = 12'b0;
200         U2_ADDR = 12'b0; U1_WDATA = 32'b0; U2_WDATA = 32'b0; U1_SLAVE = 2'b00; U2_SLAVE = 2'
               b00;
201   */
202 //---Additional priority verification
203 /*
204     #200 U1_REQ = 0; U2_REQ = 1; U1_LOCK = 0; U2_LOCK = 1; U1_WRITE = 0; U2_WRITE = 1;
205         U1_ADDR = 12'd9; U2_ADDR = 12'd9;  U1_SLAVE = 2'b01; U2_SLAVE = 2'b10; U2_WDATA =
               $random;
206         U_SPLIT = 0;
207
208     #100 U1_REQ = 0;U2_REQ = 0; U1_LOCK = 0; U2_LOCK = 1; U1_WRITE = 0; U2_WRITE = 1;  U1_ADDR
            = 12'd9;
209         U2_ADDR = 12'd9; U1_SLAVE = 2'b00; U2_SLAVE = 2'b10;
210
211   //  #200 U1_REQ = 0; U2_REQ = 0; U1_LOCK = 0; U2_LOCK = 0; U1_WRITE = 0; U2_WRITE = 0;
          U1_ADDR = 12'b0;
212   //       U2_ADDR = 12'b0; U1_WDATA = 32'b0; U2_WDATA = 32'b0; U1_SLAVE = 2'b00; U2_SLAVE = 2'
          b00;
213
214     #200 U1_REQ = 0; U2_REQ = 0; U1_LOCK = 0; U2_LOCK = 0; U1_WRITE = 0; U2_WRITE = 0; U1_ADDR
            = 12'b0;
215         U2_ADDR = 12'b0; U1_WDATA = 32'b0; U2_WDATA = 32'b0; U1_SLAVE = 2'b00; U2_SLAVE = 2'
               b00;
216
217     // Master 1 + Master 2 request and write to Slave 1 (Master 1 granted access)
218     #200 U1_REQ = 1; U2_REQ = 1; U1_LOCK = 0; U2_LOCK = 0; U1_WRITE = 1; U2_WRITE = 1; U1_ADDR
            = 12'd1;
219         U2_ADDR = 12'd3; U1_WDATA = $random; U2_WDATA = $random; U1_SLAVE = 2'b01; U2_SLAVE =
               2'b01;
220
221     #200 U1_REQ = 0; U2_REQ = 0; U1_LOCK = 0; U2_LOCK = 0; U1_WRITE = 1; U2_WRITE = 0; U1_ADDR
            = 12'd1;
222         U2_ADDR = 12'b0; U1_SLAVE = 2'b01; U2_SLAVE = 2'b01;
223
224      #200 U1_REQ = 0; U2_REQ = 0; U1_LOCK = 0; U2_LOCK = 0; U1_WRITE = 0; U2_WRITE = 0;
            U1_ADDR = 12'b0;
225         U2_ADDR = 12'b0; U1_WDATA = 32'b0; U2_WDATA = 32'b0; U1_SLAVE = 2'b00; U2_SLAVE = 2'
               b00;
226
227      // Master 1 + Master 2 request and write to Slave 1 (Master 1 granted access)
228     #200 U1_REQ = 1; U2_REQ = 1; U1_LOCK = 0; U2_LOCK = 0; U1_WRITE = 1; U2_WRITE = 1; U1_ADDR
            = 12'd1;
229         U2_ADDR = 12'd3; U1_WDATA = $random; U2_WDATA = $random; U1_SLAVE = 2'b01; U2_SLAVE =
               2'b01;
230
231     #200 U1_REQ = 0; U2_REQ = 0; U1_LOCK = 0; U2_LOCK = 0; U1_WRITE = 1; U2_WRITE = 0; U1_ADDR
            = 12'd1;
232         U2_ADDR = 12'b0; U1_SLAVE = 2'b01; U2_SLAVE = 2'b01;
233
234      #200 U1_REQ = 0; U2_REQ = 0; U1_LOCK = 0; U2_LOCK = 0; U1_WRITE = 0; U2_WRITE = 0;
            U1_ADDR = 12'b0;
235         U2_ADDR = 12'b0; U1_WDATA = 32'b0; U2_WDATA = 32'b0; U1_SLAVE = 2'b00; U2_SLAVE = 2'
               b00;
236   */
237   end
238
239 endmodule
```

9 Limitations

This bus architecture is inspired by the functionality and features of the AMBA 2.0 AHB and APB buses, hence it consists of a fusion of the design considerations and protocols used in both.
The design limitations of this architecture consist of the following.

- Data transfer width limited to 32-bits, WORD sized packets.
- No program counter implemented within the Slave modules.
- No burst transfer capability. Only isolated individual write/read requests are possible.
- Split transactions are only possible when reading data from Slaves.
- HREADY pin is implemented as a uni-directional pin from Slaves instead of bi-directional.

10 Bugs

Some of the known bugs at the time of initial debugging are as follows.

- The user control requests from the Master module must be carefully executed in a certain timed sequence in order to avoid unpredictable responses on the top-level integration.
- The user control requests must also be toggled for a certain minimum / maximum period before returning to a non-request state in order to avoid unpredictable responses.
- The change in the clock frequency of the top-level integration, can most likely lead to bus malfunctions as this system cannot tolerate different static clock speeds.
- State bouncing between transitions may occur sometimes during certain combinations of control inputs. Signal delays play a prominent role here.

Appendix - Source Code

All RTL, Test bench source code available at https://github.com/kaushanr/System-Bus-Design

- ## Mux modules

Verilog Code A.1: Addr_MUX.v

```
`timescale 1ns / 1ps
module Addr_MUX(CLK,RST,HADDR,HADDR_1,HADDR_2,SEL);

input [1:0]SEL;
input RST,CLK;
input [15:0]HADDR_1, HADDR_2;

output reg [15:0]HADDR;

always@(posedge CLK or posedge RST)
  begin
  if(SEL == 2'b01 && RST != 1) // Master 1 selected
    begin
      HADDR = HADDR_1;
    end
  else if(SEL == 2'b10 && RST != 1) // Master 2 selected
    begin
    HADDR = HADDR_2;
    end
  else if(RST == 1) // Async reset
    begin
      HADDR = 16'bx;
    end
  else
    begin
      HADDR = 16'bx;
    end

  end

endmodule
```

Verilog Code A.2: Read_MUX.v

```
`timescale 1ns / 1ps

module Read_MUX(CLK,RST,HRDATA,HRDATA_1,HRDATA_2,HRDATA_3,SEL);

input [1:0]SEL;
input RST,CLK;
input [31:0]HRDATA_1, HRDATA_2, HRDATA_3;

output reg [31:0]HRDATA;

always@(posedge CLK or posedge RST)
  begin
  if(SEL == 2'b01 && RST != 1) // Slave 1 selected
    begin
      HRDATA = HRDATA_1;
    end
  else if(SEL == 2'b10 && RST != 1) // Slave 2 selected
    begin
      HRDATA = HRDATA_2;
    end
  else if(SEL == 2'b11 && RST != 1) // Slave 3 selected
    begin
      HRDATA = HRDATA_3;
    end
  else if(RST == 1) // Async reset
    begin
      HRDATA = 32'bx;
    end
```

```
  else
    begin
      HRDATA = 32'bx;
    end

  end

endmodule
```

Verilog Code A.3: Write_MUX.v

```
`timescale 1ns / 1ps

module Write_MUX(CLK,RST,HWDATA,HWDATA_1,HWDATA_2,SEL);

input [1:0]SEL;
input RST,CLK;
input [31:0]HWDATA_1, HWDATA_2;

output reg [31:0]HWDATA;

always@(posedge CLK or posedge RST)
  begin
  if(SEL == 2'b01 && RST != 1) // Master 1 selected
    begin
      HWDATA = HWDATA_1;
    end
  else if(SEL == 2'b10 && RST != 1) // Master 2 selected
    begin
      HWDATA = HWDATA_2;
    end
  else if(RST == 1) // Async reset
    begin
      HWDATA = 32'bx;
    end
  else
    begin
      HWDATA = 32'bx;
    end

  end

endmodule
```

Verilog Code A.4: Resp_MUX.v

```
`timescale 1ns / 1ps

module Resp_MUX(CLK,RST,HRESP,HRESP_1,HRESP_2,HRESP_3,SEL);

input [1:0]SEL;
input RST,CLK;
input [1:0]HRESP_1, HRESP_2, HRESP_3;

output reg [1:0]HRESP;

always@(posedge CLK or posedge RST)
  begin
  if(SEL == 2'b01 && RST != 1) // Slave 1 selected
    begin
      HRESP = HRESP_1;
    end
  else if(SEL == 2'b10 && RST != 1) // Slave 2 selected
    begin
      HRESP = HRESP_2;
    end
  else if(SEL == 2'b11 && RST != 1) // Slave 3 selected
    begin
      HRESP = HRESP_3;
    end
  else if(RST == 1) // Async reset
    begin
      HRESP = 0;
    end
```

```
29    else
30      begin
31        HRESP = 0;
32      end
33
34    end
35
36
37  endmodule
```

- # Decoder module

Verilog Code A.5: Decoder.v

```
1   `timescale 1ns / 1ps
2
3   module Decoder(RST,CLK,HADDR,SEL_1,SEL_2,SEL_3,SELR);
4
5   input RST,CLK;
6   input [14:0]HADDR; // looks at first 15 bits  only, 16th  bit not needed
7
8   output reg SEL_1,SEL_2,SEL_3;
9   output reg [1:0]SELR;
10
11  always@(posedge CLK or posedge RST)
12    begin
13    if(HADDR[14:13] == 2'b01 && RST != 1) // Slave 1 selected
14      begin
15        SEL_1 = 1;
16        SEL_2 = 0;
17        SEL_3 = 0;
18        SELR = 2'b01;
19      end
20    else if(HADDR[14:13] == 2'b10 && RST != 1) // Slave 2 selected
21      begin
22        SEL_1 = 0;
23        SEL_2 = 1;
24        SEL_3 = 0;
25        SELR = 2'b10;
26      end
27    else if(HADDR[14:13] == 2'b11 && RST != 1) // Slave 3 selected
28      begin
29        SEL_1 = 0;
30        SEL_2 = 0;
31        SEL_3 = 1;
32        SELR = 2'b11;
33      end
34    else if(RST == 1) // Async reset
35      begin
36        SEL_1 = 0;
37        SEL_2 = 0;
38        SEL_3 = 0;
39        SELR = 2'b00;
40      end
41    else
42      begin
43        SEL_1 = 0;
44        SEL_2 = 0;
45        SEL_3 = 0;
46        SELR = 2'b00;
47      end
48    end
49
50  endmodule
```

- # Master module

Included in Master Design section in above document

- # Slave module

Verilog Code A.6: Slave_2K.v

```
`timescale 1ns / 1ps

`define IDLE 1'b0
`define START 1'b1

// Responses
`define OKAY 2'b00
`define ERROR 2'b01
`define RETRY 2'b10

module Slave_2K(SEL,HADDR,HWDATA,HRDATA,HRESP,CLK,HREADY,RST,MLOCK,AB);

input [15:0] HADDR;
input [31:0] HWDATA;
input SEL;
input RST,CLK;
input MLOCK;

//input HWRITE;

output reg [31:0] HRDATA;
output reg [1:0] HRESP;
output reg HREADY;
output reg AB;

// Slave States
parameter IDLE = 3'd0;
parameter ACTIVE = 3'd1;
parameter WRITE = 3'd2;
parameter READ = 3'd3;


parameter MSB_ADDR = 10; // MSB for 2K Address

reg [2:0] state;
reg [10:0] S_ADDR;
reg HWRITE;
reg TRANS;

// 2K Memory Addresses

reg [31:0] S_REG [2047:0]; //2K, 32-bit registers

initial begin
 HREADY <= 1'bz;
 AB = 1'bz;
end

always @ (posedge CLK or posedge RST )

  begin
    if(RST == 1)
      begin
        HRDATA <= 32'bx;
        HRESP <= `OKAY;
        HREADY <= 1'bz;
        S_ADDR <= 11'bx;
        HWRITE <= 1'bx;
        TRANS <= 1'bx;
        state <= IDLE;
      end
  else
    begin

      state <= IDLE;

      case(state)

        IDLE:begin

          //HRDATA <= 0;
          HRESP <= `OKAY;
          //HSPLIT <= 0;
          HREADY <= 1'bz;
          state <= (SEL == 1)?ACTIVE:IDLE;

```

```
77          end
78
79          ACTIVE:begin
80
81            HRESP <= `OKAY;
82            HREADY <= 1'b1;
83            S_ADDR <= HADDR[MSB_ADDR:0];
84            HWRITE <= HADDR[12];
85            TRANS <= HADDR[15];
86              if(TRANS == 1'b1)
87                begin
88                  HREADY <= 1'b1; // toggles the HREADY line only if selected as active device
89                  HRESP <= `OKAY;
90
91                  //HSPLIT <= 0;
92
93                  state <= (HWRITE == 1)?WRITE:READ;
94
95                end
96              else
97                begin
98                  HRESP <= `ERROR;
99                  state <= (SEL == 1)?ACTIVE:IDLE;
100               end
101         end
102
103         WRITE:begin
104
105           S_ADDR <= HADDR[MSB_ADDR:0];
106           HWRITE <= HADDR[12];
107           TRANS <= HADDR[15];
108
109           if(TRANS == `START)
110             begin
111               S_REG[S_ADDR] <= HWDATA;
112               if(HWDATA == S_REG[S_ADDR])
113                 begin
114                   HRESP <= `OKAY;
115                   HREADY <= 1'b1;
116                   state <= ACTIVE;
117                 end
118               else if(HWDATA !== S_REG[S_ADDR] && MLOCK == 1)
119                 begin
120                   HRESP <= `RETRY;
121                   HREADY <= 1'b0;
122                   state <= WRITE;
123                 end
124               else
125                 begin
126                   HRESP <= `ERROR;
127                   HREADY <= 1'b0;
128                   state <= ACTIVE;
129                 end
130             end
131           else
132             begin
133               HRESP <= `RETRY;
134               state <= ACTIVE;
135             end
136         end
137
138         READ:begin
139
140           S_ADDR <= HADDR[MSB_ADDR:0];
141           HWRITE <= HADDR[12];
142           TRANS <= HADDR[15];
143
144           if(TRANS == `START)
145             begin
146
147               HRDATA <= S_REG[S_ADDR];
148
149             end
150           else
151             begin
152               HRESP <= `RETRY;
```

```
153                 state <= ACTIVE;
154               end
155
156             if(HRDATA == S_REG[S_ADDR])
157               begin
158                 HRESP <= `OKAY;
159                 HREADY <= 1'b1;
160                 state <= ACTIVE;
161               end
162             else if(HRDATA != S_REG[S_ADDR] && MLOCK == 1)
163               begin
164                 HRESP <= `RETRY;
165                 HREADY <= 1'b0;
166                 state <= READ;
167               end
168             else
169               begin
170                 HRESP <= `ERROR;
171                 HREADY <= 1'b0;
172                 state <= ACTIVE;
173               end
174           end
175         endcase
176       end
177     end
178 endmodule
```

Verilog Code A.7: Slave_4K.v

```
1  `timescale 1ns / 1ps
2
3  `define IDLE 1'b0
4  `define START 1'b1
5
6  // Responses
7  `define OKAY 2'b00
8  `define ERROR 2'b01
9  `define RETRY 2'b10
10
11 module Slave_4K(SEL,HADDR,HWDATA,HRDATA,HRESP,CLK,HREADY,RST,MLOCK,AB);
12
13 input [15:0] HADDR;
14 input [31:0] HWDATA;
15 input SEL;
16 input RST,CLK;
17 input MLOCK;
18
19 //input HWRITE;
20
21 output reg [31:0] HRDATA;
22 output reg [1:0] HRESP;
23 output reg HREADY;
24 output reg AB;
25
26 // Slave States
27 parameter IDLE = 3'd0;
28 parameter ACTIVE = 3'd1;
29 parameter WRITE = 3'd2;
30 parameter READ = 3'd3;
31
32
33 parameter MSB_ADDR = 11; // MSB for 2K Address
34
35 reg [2:0] state;
36 reg [10:0] S_ADDR;
37 reg HWRITE;
38 reg TRANS;
39
40 // 4K Memory Addresses
41
42 reg [31:0] S_REG [4095:0]; //4K, 32-bit registers
43
44 initial begin
45   AB = 1'bz;
46   HREADY <= 1'bz;
```

```
47 end
48
49
50 always @ (posedge CLK or posedge RST )
51
52   begin
53     if(RST == 1)
54       begin
55         HRDATA <= 32'bx;
56         HRESP <= `OKAY;
57         HREADY <= 1'bz;
58         S_ADDR <= 11'bx;
59         HWRITE <= 1'bx;
60         TRANS <= 1'bx;
61         state <= IDLE;
62       end
63   else
64     begin
65
66       state <= IDLE;
67
68       case(state)
69
70         IDLE:begin
71
72           //HRDATA <= 0;
73           HRESP <= `OKAY;
74           //HSPLIT <= 0;
75           HREADY <= 1'bz;
76           state <= (SEL == 1)?ACTIVE:IDLE;
77
78         end
79
80         ACTIVE:begin
81
82           HRESP <= `OKAY;
83           HREADY <= 1'b1;
84           S_ADDR <= HADDR[MSB_ADDR:0];
85           HWRITE <= HADDR[12];
86           TRANS <= HADDR[15];
87             if(TRANS == 1'b1)
88               begin
89                 HREADY <= 1'b1; // toggles the HREADY line only if selected as active device
90                 HRESP <= `OKAY;
91
92                 //HSPLIT <= 0;
93
94                 state <= (HWRITE == 1)?WRITE:READ;
95
96               end
97             else
98               begin
99                 HRESP <= `ERROR;
100                state <= (SEL == 1)?ACTIVE:IDLE;
101              end
102        end
103
104        WRITE:begin
105
106          S_ADDR <= HADDR[MSB_ADDR:0];
107          HWRITE <= HADDR[12];
108          TRANS <= HADDR[15];
109
110          if(TRANS == `START)
111            begin
112              S_REG[S_ADDR] <= HWDATA;
113              if(HWDATA == S_REG[S_ADDR])
114                begin
115                  HRESP <= `OKAY;
116                  HREADY <= 1'b1;
117                  state <= ACTIVE;
118                end
119              else if(HWDATA !== S_REG[S_ADDR] && MLOCK == 1)
120                begin
121                  HRESP <= `RETRY;
122                  HREADY <= 1'b0;
```

```
                    state <= WRITE;
                  end
                else
                  begin
                    HRESP <= `ERROR;
                    HREADY <= 1'b0;
                    state <= ACTIVE;
                  end
              end
            else
              begin
                HRESP <= `RETRY;
                state <= ACTIVE;
              end
          end

          READ:begin

            S_ADDR <= HADDR[MSB_ADDR:0];
            HWRITE <= HADDR[12];
            TRANS <= HADDR[15];


            if(TRANS == `START)
              begin
                HRDATA <= S_REG[S_ADDR];

              end
            else
              begin
                HRESP <= `RETRY;
                state <= ACTIVE;
              end

            if(HRDATA == S_REG[S_ADDR] /*&& MLOCK == 1 && rand_split == 0*/)
              begin
                HRESP <= `OKAY;
                HREADY <= 1'b1;
                state <= ACTIVE;
              end
            else if(HRDATA != S_REG[S_ADDR] && MLOCK == 1)
              begin
                HRESP <= `RETRY;
                HREADY <= 1'b0;
                state <= READ;
              end
            else
              begin
                HRESP <= `ERROR;
                HREADY <= 1'b0;
                state <= ACTIVE;
              end
          end
        endcase
      end
    end
endmodule
```

Verilog Code A.8: Slave_2K_Split.v

```
`timescale 1ns / 1ps

`define IDLE 1'b0
`define START 1'b1

// Responses
`define OKAY 2'b00
`define ERROR 2'b01
`define RETRY 2'b10
`define SPLIT 2'b11

module Slave_2K_Split(SEL,HADDR,HWDATA,HRDATA,HRESP,CLK,HREADY,HSPLIT,RST,HMAS,MLOCK,USPLIT,AB
    );

input [15:0] HADDR;
```

```
15 input [31:0] HWDATA;
16 input SEL;
17 input RST,CLK;
18 input [1:0] HMAS;
19 input MLOCK;
20 input USPLIT;
21 //input HWRITE;
22
23 output reg [31:0] HRDATA;
24 output reg [1:0] HRESP;
25 output reg [1:0] HSPLIT;
26 output reg HREADY;
27 output reg AB;
28
29 // Slave States
30 parameter IDLE = 3'd0;
31 parameter ACTIVE = 3'd1;
32 parameter WRITE = 3'd2;
33 parameter READ = 3'd3;
34 parameter SPLITX = 3'd4;
35
36 parameter MSB_ADDR = 10; // MSB for 2K Address
37
38 reg [2:0] state;
39 reg [10:0] S_ADDR;
40 reg HWRITE;
41 reg rand_split;
42 reg [1:0] save_master;
43 reg TRANS;
44
45 // 2K Memory Addresses
46
47 reg [31:0] S_REG [2047:0]; //2K, 32-bit registers
48
49 initial begin
50   HREADY = 1'bz;
51   AB = 1'bz;
52   state = IDLE;
53 end
54
55
56 always @ (posedge CLK or posedge RST )
57
58   begin
59     if(RST == 1)
60       begin
61         HRDATA <= 32'bx;
62         HRESP <= `OKAY;
63         HSPLIT <= 0;
64         HREADY <= 1'bz;
65         S_ADDR <= 11'bx;
66         HWRITE <= 1'bx;
67         save_master <= 0;
68         TRANS <= 1'bx;
69         state <= IDLE;
70       end
71   else
72     begin
73
74       state <= IDLE;
75
76       case(state)
77
78         IDLE:begin
79
80           HRDATA <= 32'bx;
81           HRESP <= `OKAY;
82           HSPLIT <= 0;
83           HREADY <= 1'bz;
84           state <= (SEL == 1)?ACTIVE:IDLE;
85
86         end
87
88         ACTIVE:begin
89
90           rand_split <= USPLIT;
```

```
91            //HRDATA <= 32'bx;
92            save_master <= HMAS;
93            HRESP <= `OKAY;
94            HREADY <= 1'b1;
95            S_ADDR <= HADDR[MSB_ADDR:0];
96            HWRITE <= HADDR[12];
97            TRANS <= HADDR[15];
98              if(TRANS == 1'b1)
99                begin
100                 HREADY <= 1'b1; // toggles the HREADY line only if selected as active device
101                 HRESP <= `OKAY;
102
103                 HSPLIT <= 0;
104
105                 state <= (HWRITE == 1)?WRITE:READ;
106
107               end
108             else
109               begin
110                 HRESP <= `ERROR;
111                 state <= (SEL == 1)?ACTIVE:IDLE;
112               end
113         end
114
115         WRITE:begin
116
117           S_ADDR <= HADDR[MSB_ADDR:0];
118           HWRITE <= HADDR[12];
119           TRANS <= HADDR[15];
120
121           if(TRANS == `START)
122             begin
123               S_REG[S_ADDR] <= HWDATA;
124               if(HWDATA == S_REG[S_ADDR])
125                 begin
126                   HRESP <= `OKAY;
127                   HREADY <= 1'b1;
128                   state <= ACTIVE;
129                 end
130               else if(HWDATA !== S_REG[S_ADDR] && MLOCK == 1)
131                 begin
132                   HRESP <= `RETRY;
133                   HREADY <= 1'b0;
134                   state <= WRITE;
135                 end
136               else
137                 begin
138                   HRESP <= `ERROR;
139                   HREADY <= 1'b0;
140                   state <= ACTIVE;
141                 end
142             end
143           else
144             begin
145               HRESP <= `RETRY;
146               state <= ACTIVE;
147             end
148         end
149
150         READ:begin
151
152           S_ADDR <= HADDR[MSB_ADDR:0];
153           HWRITE <= HADDR[12];
154           TRANS <= HADDR[15];
155
156           rand_split <= USPLIT;
157
158
159           if(TRANS == `START && rand_split != 1'b1)
160             begin
161               HRDATA <= S_REG[S_ADDR];
162             end
163           else
164             begin
165               HRESP <= `RETRY;
166               state <= ACTIVE;
```

```
167             end
168
169
170           if(HRDATA == S_REG[S_ADDR] && rand_split != 1'b1)
171             begin
172               HRESP <= `OKAY;
173               HREADY <= 1'b1;
174               state <= ACTIVE;
175             end
176           else if(HRDATA != S_REG[S_ADDR] && MLOCK == 1 && rand_split != 1'b1)
177             begin
178               HRESP <= `RETRY;
179               HREADY <= 1'b0;
180               state <= READ;
181             end
182           else if(rand_split == 1'b1)
183             begin
184               //save_master <= HMAS;
185               HRESP <= `SPLIT;
186               //HREADY <= 1'bz;
187               state <= SPLITX;
188             end
189           else
190             begin
191               HRESP <= `ERROR;
192               HREADY <= 1'b0;
193               state <= ACTIVE;
194             end
195         end
196
197         SPLITX:begin
198
199           //save_master <= HMAS;
200           //HRESP <= `SPLIT;
201           HREADY <= 1'bz;
202
203           rand_split <= USPLIT;
204           if (rand_split == 0)
205             begin
206               HSPLIT <= save_master;
207               HRESP <= `OKAY;
208               if(/*HREADY == 1 &&*/ SEL == 1 && HMAS == save_master)
209                 begin
210                   HREADY <= 1'b0;
211                   HRESP <= `OKAY;
212                   state <= READ;
213                 end
214
215               else
216                 begin
217                   state <= SPLITX;
218                   HRESP <= `ERROR;
219                 end
220             end
221
222           else
223             begin
224               state <= SPLITX;
225             end
226
227         end
228
229       endcase
230     end
231   end
232 endmodule
```

- **Arbiter module**

Included in Arbiter Module section in above document

- **Bus module**

Verilog Code A.9: BUS.v

```
`timescale 1ns / 1ps

module BUS(

//Common inputs

input wire RST, CLK,

//Address and Control MUX I/O

//input wire [1:0]SEL,
input wire [15:0]HADDR_1,HADDR_2,
output wire [15:0]HADDR,

//Write MUX I/O

input wire [1:0]SEL,
input wire [31:0]HWDATA_1,HWDATA_2,
output wire [31:0]HWDATA,

//Read MUX I/O

input wire [31:0]HRDATA_1, HRDATA_2,
input wire [31:0]HRDATA_3,
output wire [31:0]HRDATA,

//Response MUX I/O

input wire [1:0]HRESP_1, HRESP_2,
input wire [1:0]HRESP_3,
output wire [1:0]HRESP,

//Decoder MUX I/O

output wire SEL_1,
output wire SEL_2,
output wire SEL_3,

//Blank

output wire AB

);

wire [1:0]SELR;


Decoder Decoder(.CLK(CLK),.RST(RST),.HADDR(HADDR),.SEL_1(SEL_1),.SEL_2(SEL_2),.SEL_3(SEL_3),.
    SELR(SELR));

Addr_MUX Addr_MUX(.CLK(CLK),.RST(RST),.HADDR(HADDR),.HADDR_1(HADDR_1),.HADDR_2(HADDR_2),.SEL(
    SEL));
Write_MUX Write_MUX(.CLK(CLK),.RST(RST),.HWDATA(HWDATA),.HWDATA_1(HWDATA_1),.HWDATA_2(HWDATA_2
    ),.SEL(SEL));
Read_MUX Read_MUX(.CLK(CLK),.RST(RST),.HRDATA(HRDATA),.HRDATA_1(HRDATA_1),.HRDATA_2(HRDATA_2)
    ,.HRDATA_3(HRDATA_3),.SEL(SELR));
Resp_MUX Resp_MUX(.CLK(CLK),.RST(RST),.HRESP(HRESP),.HRESP_1(HRESP_1),.HRESP_2(HRESP_2),.
    HRESP_3(HRESP_3),.SEL(SELR));


endmodule
```

Verilog Code A.10: TOP_BUS_ARB.v

```
`timescale 1ns / 1ps

module TOP_BUS_ARB(

//Common I/O

input wire RST,CLK,
input wire HREADY,

```

```
//Arbiter I/O

input wire HLOCK_1,HLOCK_2,
input wire HREQ_1,HREQ_2,
input wire [1:0]HSPLIT,

output wire HGRANT_1,HGRANT_2,
output wire [1:0]HMAS,
output wire MLOCK,


//BUS I/O


input wire [15:0]HADDR_1,HADDR_2,
input wire [31:0]HWDATA_1,HWDATA_2,
input wire [31:0]HRDATA_1,HRDATA_2,
input wire [31:0]HRDATA_3,
input wire [1:0]HRESP_1,HRESP_2,HRESP_3,

output wire [31:0]HWDATA,
output wire [31:0]HRDATA,
output wire [15:0]HADDR,
output wire [1:0] HRESP,
output wire SEL_1,SEL_2,SEL_3,

//Blank wire

output wire AB

);


wire [1:0]SEL;




Arbiter Arbiter(.CLK(CLK),.RST(RST),.HLOCK_1(HLOCK_1),
             .HLOCK_2(HLOCK_2),.HREQ_1(HREQ_1),.HREQ_2(HREQ_2),
             .HRESP(HRESP),.HSPLIT(HSPLIT),.SEL(SEL),.HMAS(HMAS),
             .HGRANT_1(HGRANT_1),.HGRANT_2(HGRANT_2),
             .MLOCK(MLOCK),.AB(AB),.HREADY(HREADY));

BUS BUS(.CLK(CLK),.RST(RST),.SEL(SEL),.HADDR_1(HADDR_1),.HADDR_2(HADDR_2),
      .HADDR(HADDR),.HWDATA_1(HWDATA_1),.HWDATA_2(HWDATA_2),.HWDATA(HWDATA),
      .HRDATA_1(HRDATA_1),.HRDATA_2(HRDATA_2),.HRDATA_3(HRDATA_3),.HRDATA(HRDATA),
      .HRESP_1(HRESP_1),.HRESP_2(HRESP_2),.HRESP_3(HRESP_3),.HRESP(HRESP),.SEL_1(SEL_1),
      .SEL_2(SEL_2),.SEL_3(SEL_3),.AB(AB));


endmodule
```

- **Top module**

Verilog Code A.11: TOP_MODULE.v

```
`timescale 1ns / 1ps

module TOP_MODULE(

//Common Top Module I/O

input wire RST,
input wire CLK,

// Master 1 Control I/O

input wire U1_REQ,
input wire U1_LOCK,
input wire U1_WRITE,
input wire [11:0]U1_ADDR,
input wire [31:0]U1_WDATA,
input wire [1:0]U1_SLAVE,

```

```
19 //Master 2 Control I/O
20
21 input wire U2_REQ,
22 input wire U2_LOCK,
23 input wire U2_WRITE,
24 input wire [11:0]U2_ADDR,
25 input wire [31:0]U2_WDATA,
26 input wire [1:0]U2_SLAVE,
27
28
29 //Slave 1 Control I/O
30
31 input wire U_SPLIT,
32
33 // Blanks
34
35 output wire AB
36
37
38 );
39
40    wire [31:0]s11,m12,m22,s21,s31;
41    wire [15:0]m11,m21;
42    wire [1:0]s12,s13,HMAS,s22,s32;
43    wire m13,m23,m14,m24,s14,MLOCK,m15,m25,s23,s33;
44
45    //Common wires
46
47    wire HREADY;
48    wire [1:0]HRESP;
49    wire [15:0]HADDR;
50    wire [31:0]HWDATA, HRDATA;
51
52
53
54
55    TOP_BUS_ARB BUS_ARB(.HADDR_1(m11),.HADDR_2(m21),.HRDATA_1(s11),.HRDATA_2(s21),.HRDATA_3(s31
          ),.HRESP_1(s12),
56                  .HRESP_2(s22),.HRESP_3(s32),.HSPLIT(s13),.HWDATA_1(m12),.HWDATA_2(m22),.CLK(
                        CLK),.HLOCK_1(m13),
57                  .HLOCK_2(m23),.HREQ_1(m14),.HREQ_2(m24),.RST(RST),.HREADY(HREADY),.SEL_1(s14)
                        ,.SEL_2(s23),.SEL_3(s33),
58                  .MLOCK(MLOCK),.HGRANT_1(m15),.HGRANT_2(m25),.HWDATA(HWDATA),.HRESP(HRESP),.
                        HRDATA(HRDATA),.HMAS(HMAS),
59                  .HADDR(HADDR),.AB(AB));
60
61    Master MASTER_1(.CLK(CLK),.RST(RST),.HGRANT(m15),.HREADY(HREADY),.HRESP(HRESP),.HRDATA(
          HRDATA),.HREQ(m14),.HLOCK(m13),.HADDR(m11),
62                  .HWDATA(m12),.U_REQ(U1_REQ),.U_LOCK(U1_LOCK),.U_WRITE(U1_WRITE),.U_ADDR(U1_ADDR)
                        ,.U_WDATA(U1_WDATA),
63                  .U_SLAVE(U1_SLAVE),.AB(AB));
64
65    Master MASTER_2(.CLK(CLK),.RST(RST),.HGRANT(m25),.HREADY(HREADY),.HRESP(HRESP),.HRDATA(
          HRDATA),.HREQ(m24),.HLOCK(m23),.HADDR(m21),
66                  .HWDATA(m22),.U_REQ(U2_REQ),.U_LOCK(U2_LOCK),.U_WRITE(U2_WRITE),.U_ADDR(U2_ADDR)
                        ,.U_WDATA(U2_WDATA),
67                  .U_SLAVE(U2_SLAVE),.AB(AB));
68
69    Slave_2K_Split SLAVE_1(.HADDR(HADDR),.HWDATA(HWDATA),.SEL(s14),.CLK(CLK),.RST(RST),.HMAS(
          HMAS),.MLOCK(MLOCK),.USPLIT(U_SPLIT),.HRDATA(s11),
70                      .HRESP(s12),.HSPLIT(s13),.HREADY(HREADY),.AB(AB));
71
72    Slave_2K SLAVE_2(.HADDR(HADDR),.HWDATA(HWDATA),.SEL(s23),.CLK(CLK),.RST(RST),.MLOCK(MLOCK)
          ,.HRDATA(s21),
73                  .HRESP(s22),.HREADY(HREADY),.AB(AB));
74
75    Slave_4K SLAVE_3(.HADDR(HADDR),.HWDATA(HWDATA),.SEL(s33),.CLK(CLK),.RST(RST),.MLOCK(MLOCK)
          ,.HRDATA(s31),
76                  .HRESP(s32),.HREADY(HREADY),.AB(AB));
77
78
79 endmodule
```

References

[1] AMBA Specification (Rev 2.0) - Arm