Department of Electronic and Telecommunication Engineering

University of Moratuwa

EN4020: Advanced Digital Systems

System Bus Design and Simulation Report

Group Members:

K.G.L.P. Nawarathna	160430A
N.L.Udugampola	160640R
W.W.A.T.E.Weerasinghe	160680M

24th December 2020

Table of Contents

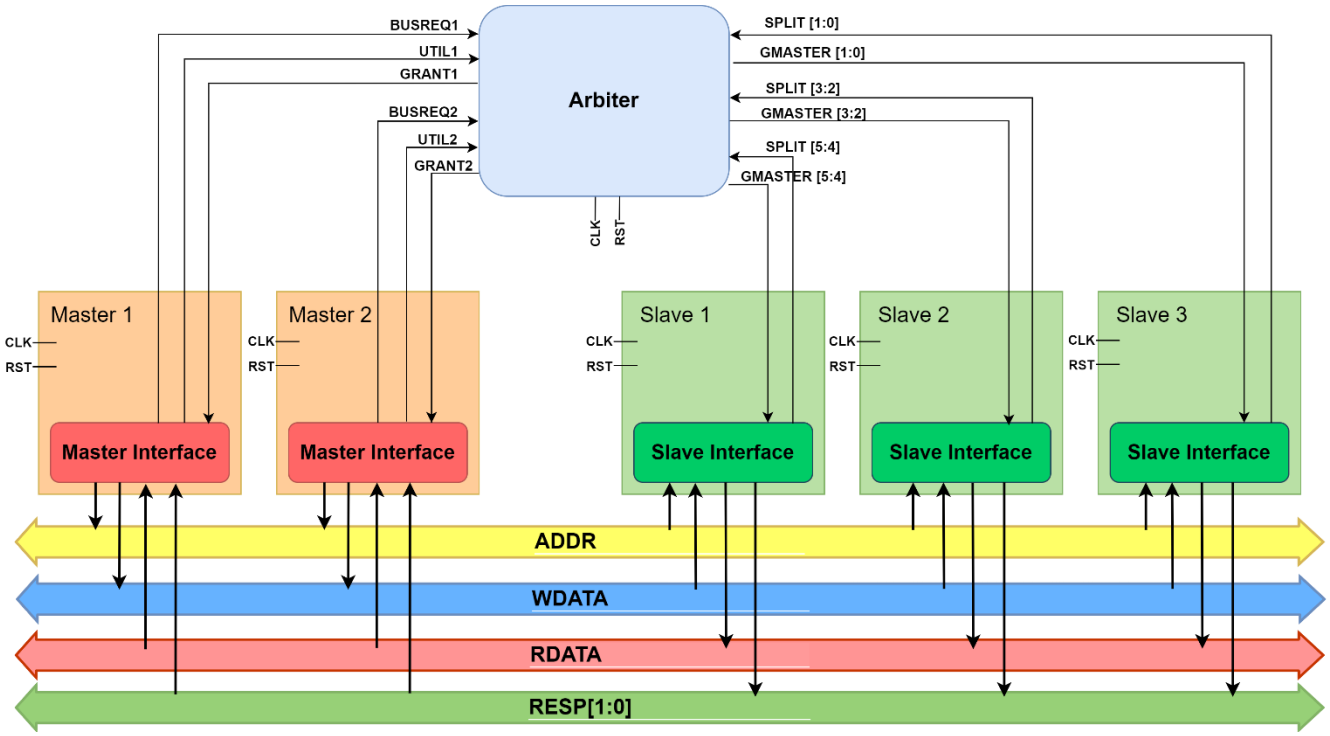
1 Introduction

This document is intended to describe the design and simulation of a system bus as a partial fulfillment for the module, EN4020: Advanced Digital Systems. 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. Quartus Prime and ModelSim are the software applications used for simulation and compilation for checking errors.

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, a **serial write-data bus** to send 8-bit data to slave devices, a **serial read-data bus** to read 8-bit data from slave devices and a **response bus** for the slave devices to respond to read or write requests from master devices. 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. Following diagram shows the architecture of the design.

Design Architecture



2 Signal List

The list below shows the signals in our system bus architecture, which are basically the main bus lines and the control signals, connecting with the arbiter.

Name	Source	Description
CLK Bus clock	Clock source	Bus protocol is synchronous and CLK handles all timing operations in the bus. All signal timings are related to the rising edge of CLK.
RESETn Reset	External source	Reset signal is the only asynchronous and active LOW signal. It is used to reset all the states of the system and the bus.
BUSREQx BUSREQ[1:0] for 2 masters to request bus	Master	A signal from master x to the bus arbiter which is used to inform that the master x requires the bus access. Each master has a dedicated BUSREQ signal and the arbiter facilitates up to 2 masters. When requesting the bus, master must keep this signal HIGH until it is granted access.
UTILx UTIL[1:0] for 2 masters' utilized flags	Master	If master x is utilizing the bus for a data transfer, it must keep this signal HIGH throughout the time of the transaction. The master must make UTIL signal LOW at the end of every single transaction. As a result, when the master performs a burst of many transactions, the arbiter can grant the master for each of the transactions while considering master priority as well.
GRANTx GRANT[1:0] for 2 masters' bus grant	Arbiter	This signal is made HIGH by the arbiter to indicate master x that it has permission to access the bus. Only one master will be granted at a time.
GMASTERx[1:0]	Arbiter	This signal is used by the arbiter to informs the currently granted master to slave x. If this signal is made **2'b10** ,granted master is master1 and if this signal is made

GMASTER[5:0] for 3 slaves to check granted master		**2'b01**,granted master is master2. In a SPLIT transaction, slave checks this signal to know when to continue.
SPLITx [1:0] SPLIT[5:0] for 3 slaves to do split requests	Slave	This signal is used by slave x to request arbiter to grant a particular master when slave x is ready to continue a previous transaction with the master to which the slave has responded BUSY in the first time. This split request can be made for the grant of master1 by making this signal **2'b10** and for the grant of master2 by making this signal **2'b01**.
ADDR Address bus	Master	This is the serial system address bus through which a granted master can send the address of a 8-bit slave register by sending a 16-bit address packet. Its first bit is a start bit which is logic 1. Next 2 bits are for the slave device address. The bit after that is for indicating whether the master is going to do a READ or a WRITE transaction. Last 12 bits are for the address of the memory location of the slave.
WDATA Data write bus	Master	This is the serial system WDATA bus used to transfer 8-bit data from the granted master to a slave during a WRITE operation.
RDATA Data read bus	Slave	This is the serial system RDATA bus used to transfer 8-bit data from a slave to the granted master during a READ operation.
RESP[1:0] Slave response bus	Slave	This is the bus used by a slave to send a response to the granted master during a transaction. There are three types of responses named as **OK**, **BUSY**, **DONE** which can be sent by making this bus **2'b10**, **2'b01**, **2'b11** respectively. Details of these responses are written in section 3.2

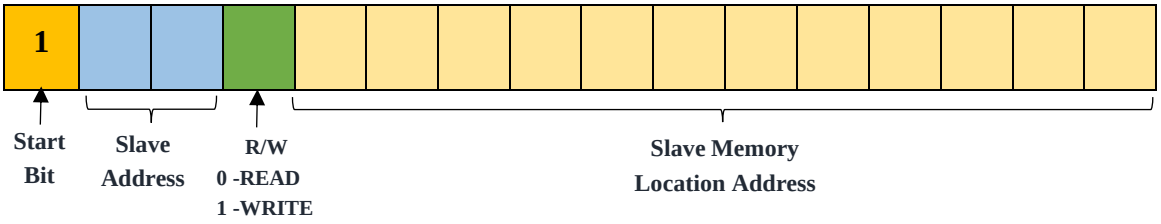
3 Functionality

3.1 Address Bus

This is the serial system address bus through which a granted master can send the address of a 8-bit slave register by sending a 16-bit address packet. It includes a start bit, the slave device address, read/write operation type and the slave memory address. When a master sends an address in the address bus, every slave connected to the bus reads 16 bits of serial data into the address decoder of slave interface (Address decoder code in APPENDIX I). After it is completed each slave decodes the address and the slave with the matching slave device address can continue the transaction. Address bus facilitates communicating with a slave device of up to 2-bit device address and up to 12-bit memory address. Connections to the address bus can be seen in the code in

APPENDIX B.

Address Format



3.2 Response Bus

This is the bus used by a slave to send a response to the granted master during a transaction. It is a 2-bit parallel bus which only maintains two logic levels in the two electrical lines during sending a response. When the master sends an address in the address bus, the relevant slave with the matching slave device address has to respond with a OK or BUSY response to continue with the data transaction and at the end of the transaction it has to send a DONE response to indicate successful completion. These three responses are further elaborated in the below table. Connections to the response bus can be seen in the code in

APPENDIX B and the response origination occurs in the slave interface, which is in APPENDIX I.

Response Types

RESP[1:0]	Type	Description

10	OK	OK response indicates that the address matched slave is free to start the data read/write transfer in the data bus. A data read/write transfer always happens immediately after an OK response.
01	BUSY	After the relevant slave receives the 16-bit packet sent in the address bus, the slave can reply with a BUSY response if it is not ready to receive or send data at the moment. This will lead to a split situation where the transaction is postponed, and the bus is released for other transactions of other parties.
11	DONE	After the slave sending or receiving an 8-bit data in data write/read bus, a DONE response is sent to notify the master that the transaction completed successfully.
00	NAK	Both lines in logic LOW correspond to no response.

3.3 WDATA Bus & RDATA Bus

The granted master can write 8-bit data to the address matched slave using serial WDATA bus and it can read 8-bit data from the slave using serial RDATA bus. In writing data right after the address is sent, the master must wait till it gets an OK response to start writing 8 bits of data to the WDATA bus. Also, in reading data right after the address is sent, the master must wait till it gets an OK response to start sampling the RDATA bus to receive the 8-bit data sent by the slave. WDATA and RDATA bus is capable of doing ***single read***, ***burst read***, ***single write***, ***burst write***. Data bus connections are in

APPENDIX B code.

SPLIT State

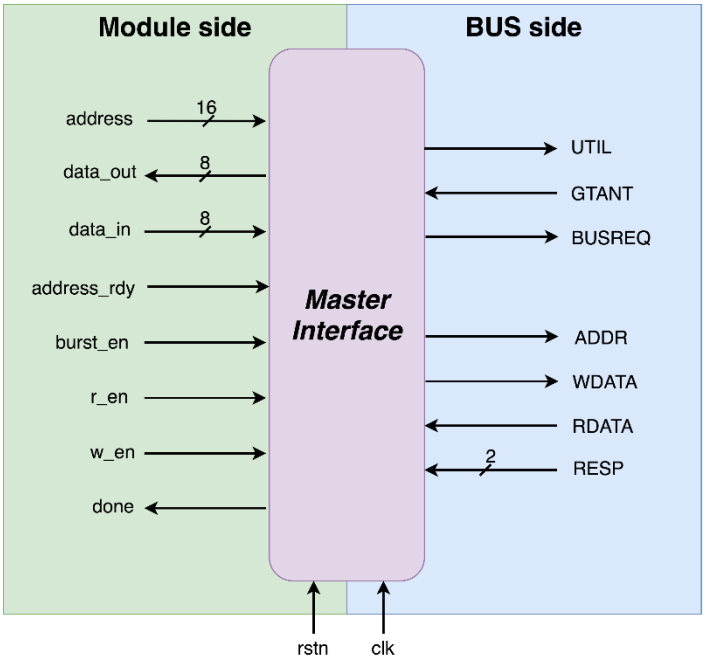
If the master gets a BUSY response instead of the above mentioned OK response, it must go to a SPLIT state by releasing the bus by making UTIL signal LOW, without writing anything to the WDATA BUS or sampling the RDATA bus. BUSY response is sent by the slave to the master as a way of telling that it is not ready yet and the master has to wait till the slave is not busy. In SPLIT state, the master must wait till it is granted again because of a SPLIT request made by the slave to the arbiter on behalf of the master. As soon as the master is granted again, the next response is going to be an OK response and with that the transaction can be completed using WDATA bus or RDATA bus.

Priority Control

If the bus is not utilized by any master and if there are several bus requests from several masters, the arbiter will grant the master with the highest priority and in our case, it is the master 1. This is actually very useful in burst reads and burst write.

Example: If master2 is doing a burst transaction of writing 100 of 8-bit data to slave2 and if master1 tries to read some data from slave3 when master2 is writing data in the 10th transaction of its 100 write burst, master1 will not have to wait till master2 completes 90 more transactions. Since master1 has the highest priority it will get to do its transactions after the 10th transaction of master2. Then master2 can continue its 90 remaining transactions when master1 is done.

3.4 Master Interface



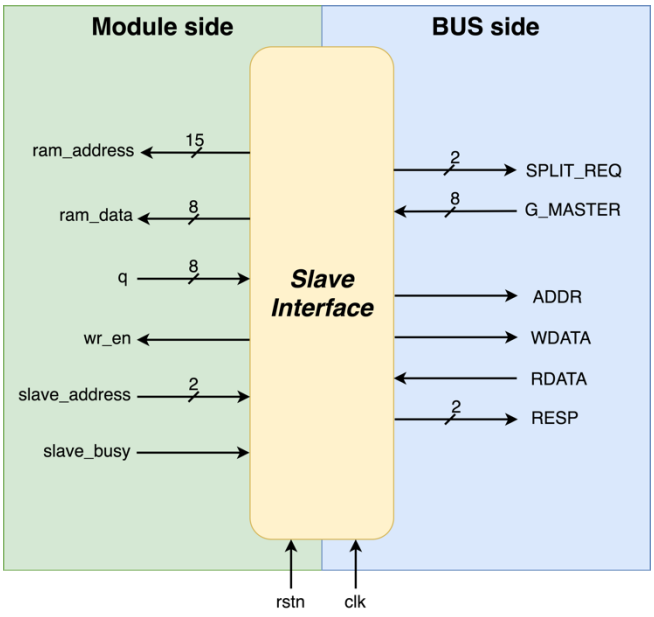
In our system bus design, we made a separate module named master interface which can be used to connect any master device to the system bus through its master module side ports which are listed below. Its bus side ports are simply connected to the main bus lines and the dedicated control signal lines with the arbiter. Verilog code for the master interface module is in APPENDIX G.

Master Module Side Ports

Port	I/O	Description
Address [15:0]	INPUT	Address of the slave for read write operations provided by the sequence in Master Core

data_out [7:0]	OUTPUT	Data received by slave is sent to Master Core from this signal
data_in [7:0]	INPUT	8-bit data which should be sent to slave provided by the sequence in Master Core
address_rdy	INPUT	Request from master core to write the provided address to the address bus
burst_en	INPUT	Become high when burst transactions are taken place
r_en	INPUT	Enable read from slave
w_en	INPUT	Enable write to slave
done	OUTPUT	When a read or write transaction completes, Master Core is notified through this signal.

3.5 Slave Interface



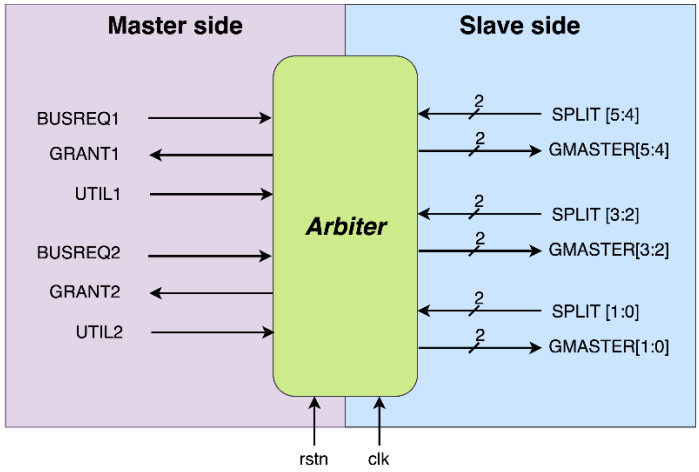
In our system bus design, we made a separate module named slave interface which can be used to connect any slave device to the system bus through its slave module side ports which are listed below. Some of its

bus side ports are connected to the main bus lines to make the slave accessible through the system bus and some are connected to the dedicated signal lines with the arbiter to make the slave capable of requesting a particular master in a split transaction while monitoring which master has access to the bus. Both of its module side and bus side ports consist of ports for address, data in and data out. As a result, this slave interface module acts as an address decoder which can be used to access any slave device with an address space up to 12bit (4k) from our system bus with a standard 16bit address format. Verilog code for the slave interface module is in APPENDIX I.

Slave Module Side Ports

Signal	Source	Description
ram_address [11:0]	Slave interface	Address of the memory location inside the connected slave device (RAM).
ram_data [7:0]	Slave interface	Data which is to be written to a memory location specified by the ram_address
q [7:0]	Slave	Output data of the connected slave which represent a value stored in a memory location specified by the ram_address.
wr_en	Slave interface	Write enable signal of the connected slave.
slave_address[1:0]	Slave	Sets the device address of the connected slave. This address is used to distinguish between slave devices connected to the system bus.
slave_busy	Slave	Enable read from slave

3.6 Arbiter



In this system bus design, a separate module named arbiter is created. It acts as the controller for slaves and masters. The ports of arbiter are divided among masters and slaves. Arbiter accepts requests from masters, grant permission for masters to utilize the bus lines and monitor the utilization status of masters. It communicates with slaves for split transactions. Slave notifies the arbiter that they are ready for split transaction with a particular master. Arbiter informs the currently granted master to the corresponding slave, which helps slave to continue the split transaction later by contacting the arbiter. Apart from that arbiter prioritize the requests from masters when there are multiple requests. The Verilog implementation of master interface can be viewed in APPENDIX K.

4 Arbiter Design

As the first step of arbiter design, it is important to understand the ports of the module.

```
1   module arbiter (
2       input clk,
3       input reset,
4       input [1:0]req_from_master,     //BUSREQ signals in format:[m1,m2]
5
6       //SPLIT signals in format:[s1->m1,s1->m2,s2->m1,s2->m2,s3->m1,s3->m2]
7       input [5:0]split_req_from_slave,
8
9       input [1:0]bus_utilization,     //UTIL signals in format:[m1,m2]
10      output reg [1:0]grant_to_master,//GRANT signals in format:[m1,m2]
11
12      //GMASTER signals in format:[s1<-m1,s1<-m2,s2<-m1,s2<-m2,s3<-m1,s3<-m2]
13      output reg [5:0]notify_granted_master_to_slave
14  ); //NOTE: m1,m2,s1,s2,s3 stands for masters and slaves
```

req_from_master indicates the requests from each master to utilize the bus line. *split_req_from_slave* refers to the request from slave to grant corresponding master. If the master is involved in memory transaction, the *bus_utilization* signal becomes high. Otherwise, it is low. Granting permission to master to write the address is accomplished by *grant_master*. Currently working master is informed to slave through *notify_granted_master_to_slave*.

In the next step the requests from masters and slaves are processed by a combinational circuit.

```
16  wire [1:0]current_requests;//[m1 requested?, m2 requested ?]
17
18  /*either get responses from masters or get request from slaves waiting in
19  the split states to grant permission for corresponding master*/
20  assign current_requests[1] = req_from_master[1] || split_req_from_slave[5]
21                      || split_req_from_slave[3] || split_req_from_slave[1];
22  assign current_requests[0] = req_from_master[0] || split_req_from_slave[4]
23                      || split_req_from_slave[2] || split_req_from_slave[0];
```

current_requests become high whenever there is a request from a master or a split request from any of the slaves to corresponding master. This signal is 2bit wide to facilitate 2 masters.

Initially no masters are granted.

```
25  // initially no master is granted. no master is notified to slave
26  initial begin
27      grant_to_master               <= 2'b00;
28      notify_granted_master_to_slave <= 6'b000000;
29  end
```

In each clock cycle the requests are processed as follows.

```
31  always @(posedge clk or negedge reset) begin
32      //asynchronous reset
33      if(~reset)begin
34          //reset to no grants and no granted notifications
35          grant_to_master               <= 2'b00;
36          notify_granted_master_to_slave <= 6'b000000;
37      end
```

```
38
39      //bus requests are checked and granted only when bus is not utilized
40      else if(bus_utilization == 2'b00) begin
41          case(current_requests) //process the requests from masters and slaves
              ...
              ...
```

If reset is applied, it initializes grant and notify signals. Otherwise, it checks the status of bus utilization and only grant when no master is utilized. As mentioned earlier in master interface, at the end of each memory transaction, the utilization line is released, which allows arbiter to process requests. If only master 1 is requested, grant is provided to master 1. If only master 2 is requested grant will be provided to master 2. In both instances the granted master is notified to all the slaves.

```
39      //bus requests are checked and granted only when bus is not utilized
40      else if(bus_utilization == 2'b00) begin
41          case(current_requests) //process the requests from masters and slaves
42              2'b00:begin         //no requests
43                  grant_to_master                 <= 2'b00;
44                  notify_granted_master_to_slave  <= 6'b000000;
45                  end
46              2'b10:begin         //request to grant master 1
47                  //grant master 1
48                  grant_to_master                 <= 2'b10;
49                  //notify 3 slaves that master 1 is granted
50                  notify_granted_master_to_slave  <= 6'b101010;
51                  end
52              2'b01:begin         //request to grant master 2
53                  //grant master 2
54                  grant_to_master                 <= 2'b01;
55                  //notify 3 slaves that master 2 is granted
56                  notify_granted_master_to_slave  <= 6'b010101;
57                  end
58              //high priority goes to master 1
59              2'b11:begin       //request to grant both master 1 and 2
60                  //master 1 is granted (high priority)
61                  grant_to_master                 <= 2'b10;
62                  //notify 3 slaves that master 1 is granted
63                  notify_granted_master_to_slave  <= 6'b101010;
64                  end
65          endcase
66      end
67  end
68
69  endmodule
```

What will happen if there is a request from both masters? In this scenario a priority-based decision should be made. Priority is given to master 1. It is depicted below.

The complete content of the arbiter.v file is given in APPENDIX K.

5 Arbiter Verification

5.1 Arbiter Testbench

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

- single master request
- two master requests
- split transaction viable scenario
- reset test

arbiter_tb.v

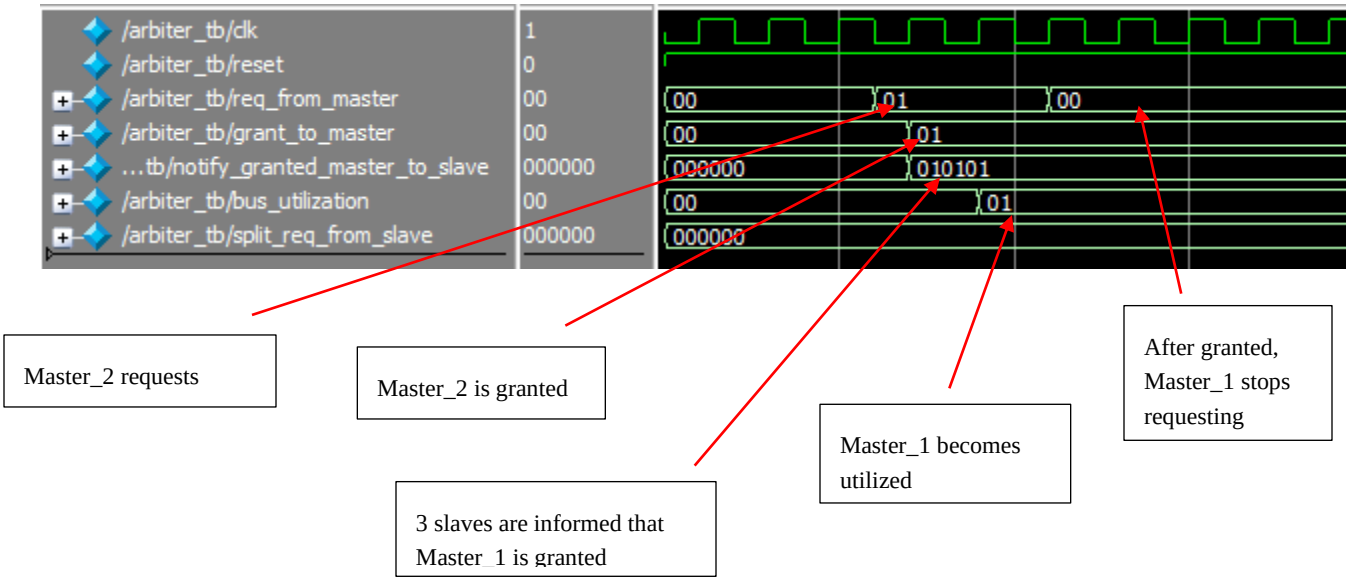
```
1   `timescale 1ps/1ps
2   module arbiter_tb();
3
4   //TB INPUTS
5   reg clk;
6   reg reset;
7   reg [1:0] req_from_master;
8   reg [5:0] split_req_from_slave;
9   reg [1:0] bus_utilization;
10  wire [1:0] grant_to_master;
11  wire [5:0] notify_granted_master_to_slave;
12
13  //RESET
14  reg reset_release_ff1 = 1;
15  reg reset_release_ff2 = 1;
16  wire reset_with_synchronous_release;
17
18  assign reset_with_synchronous_release = reset_release_ff2 && reset;
19
20  always @(posedge clk)
21  begin
22      reset_release_ff1 <= reset;
23      reset_release_ff2 <= reset_release_ff1;
24  end
25
26  always #1 clk = ~clk;
27
28  arbiter a1(
29      .clk                          (clk),
30      .reset                        (reset_with_synchronous_release),
31      .req_from_master              (req_from_master),
32      .split_req_from_slave         (split_req_from_slave),
33      .bus_utilization              (bus_utilization),
34      .grant_to_master              (grant_to_master),
35      .notify_granted_master_to_slave(notify_granted_master_to_slave));
36
37  initial begin
38      //initially no requests from either masters or slaves
39      clk = 0;
40      reset = 1;
41      req_from_master = 2'b00;
42      split_req_from_slave = 6'b000000;
43      bus_utilization = 2'b00;
44
```

```
45
46      //master 2 requests from arbiter
47      #6 req_from_master = 2'b01;
48      #3 bus_utilization = 2'b01;
49      #2 req_from_master = 2'b00;
50
51      //both masters request
52      #20 req_from_master = 2'b11;
53      #6 bus_utilization = 2'b00;   //bus utilization become low
54      #2 bus_utilization = 2'b10;   //after 1 clock cycle utiliztion becomes high
55      #2 req_from_master = 2'b00;
56
57      //slave 2 sends a split request to master 2
58      #20 split_req_from_slave =6'b000100;
59      #5 bus_utilization = 2'b00;   //previous task finished
60      #2 bus_utilization = 2'b01;
61
62      //reseting the arbiter
63      #20 reset = 1'b0;
64      //inputs to arbiter from masters and slave also reset because
65          //the entire design resets at the same time
66      #2 bus_utilization = 2'b00;
67      req_from_master = 2'b00;
68      split_req_from_slave = 6'b000000;
69
70  end
71
72  endmodule
```

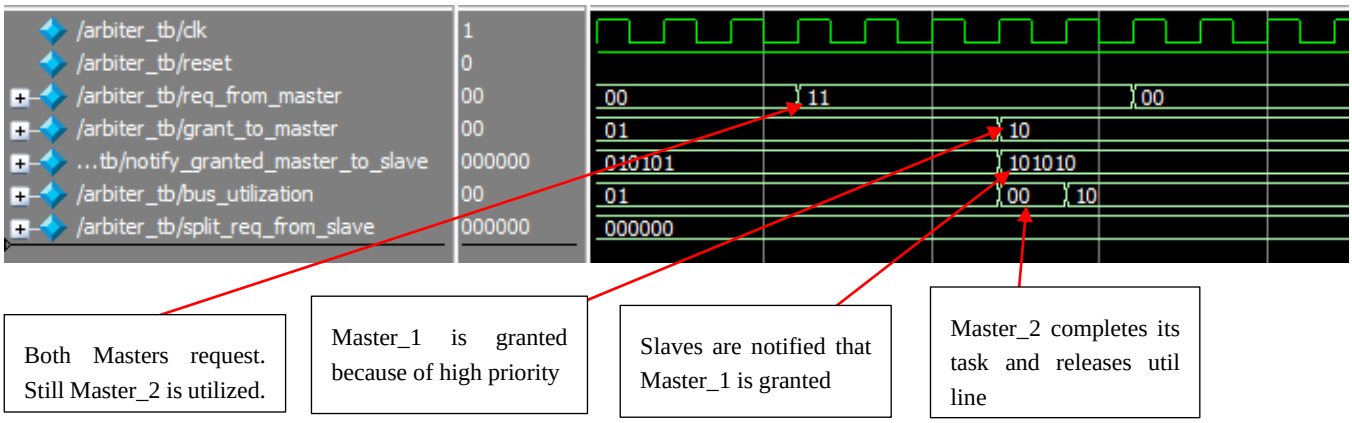
5.2 Simulation Diagrams

Following are the timing diagrams for the given cases extracted using ModelSim.

5.2.1 Single Master Request



5.2.2 2 Masters Request



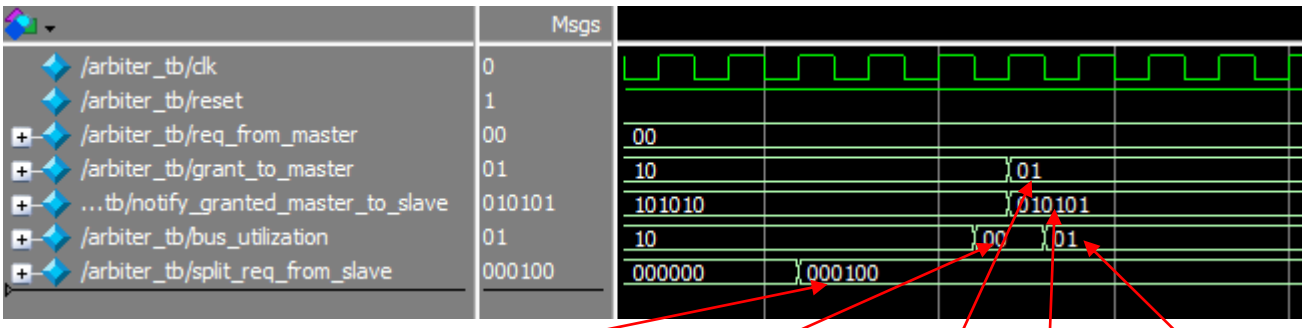
Both Masters request. Still Master_2 is utilized.

Master_1 is granted because of high priority

Slaves are notified that Master_1 is granted

Master_2 completes its task and releases util line

5.2.3 Split Transaction



Slave_2 sends a split request asking to activate Master_2

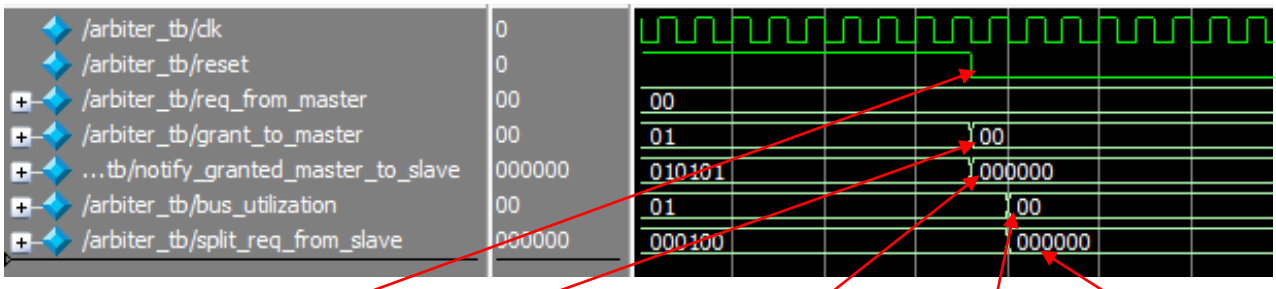
Master_1 utilized signal becomes low

Master_2 is granted

Slaves are notified that Master_2 is granted

Master_2 becomes utilized

5.2.4 Reset



Asynchronous active low reset is applied

No Master is granted

Slaves are notified that no master is granted

Utilization line becomes low because master interface also resets (reset is global)

Split requests become low because slave interface resets

6 Address Decoder (Slave Interface) Design

The Slave interface connects the slave device (RAM) to the system bus. The address decoder for the slave interface is built inside the slave device. The purpose of the address decoder is to enable the correct slave device for a particular master request. The input and output ports of the slave interface is defined as follows. (full code for the slave_interface is in APPENDIX I)

```
module slave_interface(

    input wire clk,     //system clock input
    input wire reset,   //active low reset
    input wire addr,    //address bus
    input wire w_data,  //write data bus
    output wire r_data, //read data bus
    output [1:0] response,       //response bus
    output [1:0] split_request,  //request split transaction
    input  [1:0] granted_master, //indicates the active master on the bus
    input wire [1:0]slave_address, //sets the device address of the slave
    output reg slave_en, // enable signal for tri-state buffers
    input slave_busy,    // indicates that the slave is busy
    //------for ram----//
    input wire [7:0]q,   // output data from the ram
    output [11:0]ram_address, //address input of the ram
    output [7:0]ram_data, // data input to the ram
    output wr_en
);
```

The module has number of registers to store variable data and parameters to denote constants

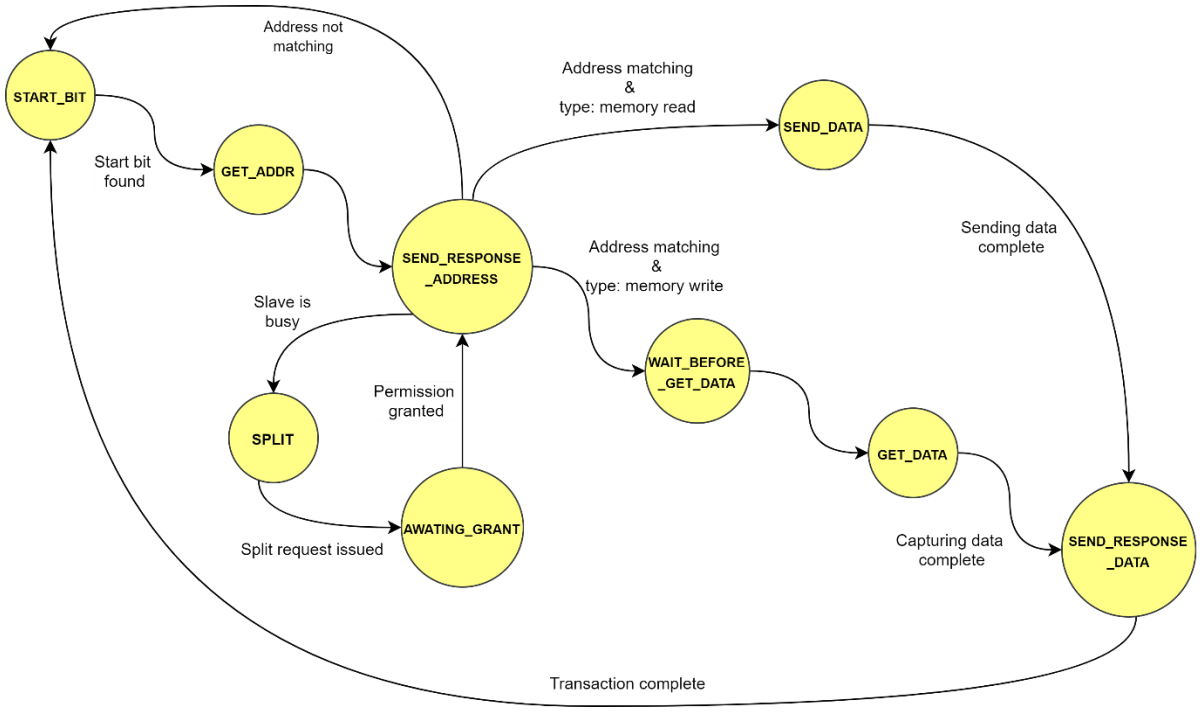
```
//-----------PARAMETERS-------------//
 //---STATES---//
 parameter AWATING_GRANT = 4'b0000;
 parameter START_BIT =4'b0001;
 parameter GET_ADDR = 4'b0010;
 parameter SEND_RESPONSE_ADDRESS = 4'b0011;
 parameter GET_DATA = 4'b0100 ;
 parameter SEND_DATA = 4'b0101 ;
 parameter SEND_RESPONSE_DATA = 4'b0110 ;
 parameter SPLIT =4'b0111;
 parameter WAIT_BEFORE_GET_DATA =4'b1000;
 //---RESPONSES---//
 parameter RESP_NAK =2'b00;
 parameter PESP_BUSY=2'b01;
 parameter RESP_OK  =2'b10;
 parameter RESP_DONE=2'b11;
```

```
 //-----------REGISTERS--------------//
 reg [3:0] state = START_BIT;

 reg [15:0]r_addr     =0;
 reg [7:0]r_data_in  =0;
 reg r_data_out =0;
 reg [1:0]r_response =0;
 reg [1:0]r_split    =0;
 reg [4:0]addr_count =0;
 reg [3:0]data_count =0;
 reg wr_en_bit =0;
 reg [15:0]split_adress=0;
 reg [1:0]split_master =0;
 reg [1:0]r_split_request =0;
 reg [3:0]busy_count=5;
```

The Slave interface clocks from the main system clock and it has an asynchronous active low reset which sets all registers to their default values. The module functions as a state machine with 9 states. The names and functionality of each state is described in the following diagram.

STATE	Value	Description
START_BIT	0001	Default state of the module. During this state, the address bus is continuously checked for the start condition. (start bit)
GET_ADDR	0010	15-bit address which follows the start bit is captured during this state.
SEND_RESPONSE _ADDRESS	0011	Decode the 2-bit slave device address, r/w bit and the12-bit slave memory address. Responds accordingly to the master if the device address is matching.
GET_DATA	0100	Capture the 8-bit data from the master.
SEND_DATA	0101	Send 8-bit data to the master.
SEND_RESPONSE _DATA	0110	Respond accordingly to the master after the transaction is complete.
SPLIT	0111	Slave is currently busy. Wait until the slave becomes active again.
WAIT_BEFORE_ GET_DATA	1000	Wait one clock cycle before capturing data from the W_DATA bus.
AWAITING_GRANT	0000	After issuing a split transfer request, wait until the arbiter grants permission to transfer.

The following diagram shows the transitions between states.



Address Decoder

The address decoder inside the slave interface modules serve three purposes.

- Detect the start bit
- Capture the 15-bit address
- Decode the slave device address, slave memory address and identify the type of transaction.

Since the backbone of the slave interface module is a finite state machine, the above three functions of the address decoder is implemented in the START_BIT, GET_ADDR_ and SEND_RESPONSE_ADDRESS states respectively.

```
77 // This is the default state of the module
78 // identifies the start bit of the address
79          START_BIT: begin
80              slave_en <= 0;
81              if (addr==1) begin
82                  state <= GET_ADDR;
83                  addr_count<=14;
84                  wr_en_bit<=0;
85                  r_addr<=0;
86                  r_data_in<=0;
87                  r_data_out<=0;
88              end
89              else
90                  state <= START_BIT;
91
92              addr_count<=15;
93              r_response<=RESP_NAK;
94          end
```

The START_BIT state is the default state of the slave interface module. During this state, the module continuously checks the address bus to detect its transition from logic 0 to logic 1 which is the start bit. After successfully detecting a start bit, the slave interface changes its state to GET_ADDR.

```
95 //Capturs the 15-bit address which follows the
start bit
96      GET_ADDR: begin
97
98          if (addr_count>1) begin
99              r_addr[15:0] <= {r_addr[14:0],addr};
100             addr_count <= addr_count -1;
101             state <= GET_ADDR;
102         end else begin
103            r_addr[15:0] <= {r_addr[14:0],addr};
104            state <= SEND_RESPONSE_ADDRESS;
105         end
106         wr_en_bit<=0;
107         r_response<=RESP_NAK;
108      end
```

During the GET_ADDR state the slave interface samples the address bus at each positive clock edge to capture the 15-bit address which immediately comes after a start bit. After successfully capturing the address the module changes its state to SEND_RESPONSE_ADDRESS.

```
110 //decode the slave address and the type of
transaction (mem r/w)
111 //and send the appropriate response to the
master.
112       SEND_RESPONSE_ADDRESS: begin
113       // check for slave address
114       if (r_addr[14:13] == slave_address) begin
115           slave_en <= 1;
116
117           if(slave_busy == 0)begin
118               if (r_addr[12]==1) begin
              // do this for mem write
119                   state<= WAIT_BEFORE_GET_DATA;
120                   r_response<=RESP_OK;
121                   data_count<=8;
122                end
123                else begin// for mem read
124                    state<= SEND_OK;
125                    r_response<=RESP_DONE;
126                    data_count<=7;
127                end
128           end
129           else begin
130               r_response<=RESP_BUSY;
                  // send BUSY response
131               split_adress<= r_addr;
132               split_master<= granted_master;
133               state <= SPLIT;
134           end
135         end
136            else begin
137                state<=START_BIT;
138            end
139       end
```

During the SEND_RESPONSE_ADDRESS state the module decodes the 15-bit address which was successfully captured during the previous state. The module first checks the 15th and 14th bits of the captured address which represent the slave device address of the master request against the actual slave device address. If the slave device address is not matching it change its state to START_BIT which is the idle state of the slave interface. If the device address is matching, then the module checks if the slave is busy. If the slave is busy, then a BUSY response is sent back to the master and the module changes its state to SPLIT. Otherwise the module checks the type of the transaction. If the transaction is a memory write type, then it issues an OK response and move to the WAIT_BEFORE_GET_DATA state. If it is a memory read type transaction the slave interface issues an OK response and move to the GET_DATA state.

7 Address Decoder (Slave Interface) Verification

7.1 Slave Interface Testbench

```
1   `timescale 1ps/1ps
2   module slave_top();
3   //-------TB_INPUTS---------//
4   reg clock50;
5   reg reset;
6   reg address_bus;
7   reg w_data_bus;
8   reg [1:0] slave_address;
9   reg [1:0] granted_master;
10  reg slave_busy;
11
12  //-------TB_OUTPUTS--------//
13  wire r_data_bus;
14  wire [1:0]split_request;
15  wire [1:0]response_bus;
16
17  //-------INTERNAL_WIRE-----------//
18  wire [7:0]q;
19  wire [11:0]ram_address;
20  wire [7:0]ram_data;
21  wire wr_en;
22  wire        sibufin_r_data_bus;
23  wire [1:0]  sibufin_response_bus;
24  wire        sibufin_en;
25  //-----Trisate Buffers for RDATA,RESPONSE bus-----//
26  bufif1  buf_r_data_bus(r_data_bus, sibufin_r_data_bus, sibufin_en);
27  bufif1  buf_response_bus1(response_bus[1], sibufin_response_bus[1], sibufin_en);
28  bufif1  buf_response_bus0(response_bus[0], sibufin_response_bus[0], sibufin_en);
29
30  //----------SLAVE_INTERFACE INSTANTIATION----//
31  slave_interface SLAVE(
32      .clk                (clock50),
33      .reset              (reset),
34      .addr               (address_bus),
35      .w_data             (w_data_bus),
36      .r_data             (sibufin_r_data_bus),
37      .response           (sibufin_response_bus),
38      .split_request      (split_request),
39      .slave_address      (slave_address),
40      .granted_master     (granted_master),
41      .q                  (q),
42      .ram_address        (ram_address),
43      .ram_data           (ram_data),
44      .wr_en              (wr_en),
45      .slave_en           (sibufin_en),
46      .slave_busy         (slave_busy)
47  );
48
49  //---------RAM_INSTANCE-------//
50  syncRAM RAM_BASIC(
51      .dataOut            (q),
52      .dataIn             (ram_data),
53      .Clk                (clock50),
54      .Addr               (ram_address),
```

```
55      .Wr_en              (wr_en),
56      .reset              (reset)
57  );
58
59  reg reset_release_ff1 = 1;
60  reg reset_release_ff2 = 1;
61  wire reset_with_synchronous_release;
62
63  assign reset_with_synchronous_release = reset_release_ff2 && reset;
64
65  always @(posedge clock50)
66  begin
67      reset_release_ff1 <= reset;
68      reset_release_ff2 <= reset_release_ff1;
69  end
70
71
72  reg[15:0] address;
73  reg[7:0] data;
74  integer i;
75  always #1 clock50 = ~clock50;
76
77  //---TEST_INPUTS
78
79  initial begin
80
81  //---SET THE INITIAL CONDITIONS
82  clock50=1'b0;
83  reset =1'b1;
84  address_bus=1'b0;
85  w_data_bus=1'b0;
86  slave_address=2'b01;
87  slave_busy=1'b0;
88  granted_master=2'b01;
89  address = {1'b1,2'b01,1'b1,12'b001110001010};
90  data=8'b10001010;
91
92  //--- master 01 is writing data to slave 01------//
93  //--- sending address
94  #2;
95  reset=0;
96  #2;reset=1;
97
98  #1;
99  for (i=0;i<16;i=i+1) begin
100     address_bus=address[15-i]; #2;
101     end
102 #4;
103
104 //----sending data
105 for (i=0;i<8;i=i+1) begin
106     w_data_bus=data[7-i]; #2;
107     end
108 //------------transaction done-----------------//
109 #16;
110 //--- master 01 is reading from slave 01--------//
111 address = {1'b1,2'b01,1'b0,12'b001110001010};
112 //--- sending address
113 for (i=0;i<16;i=i+1) begin
114     address_bus=address[15-i]; #2;
115     end
```

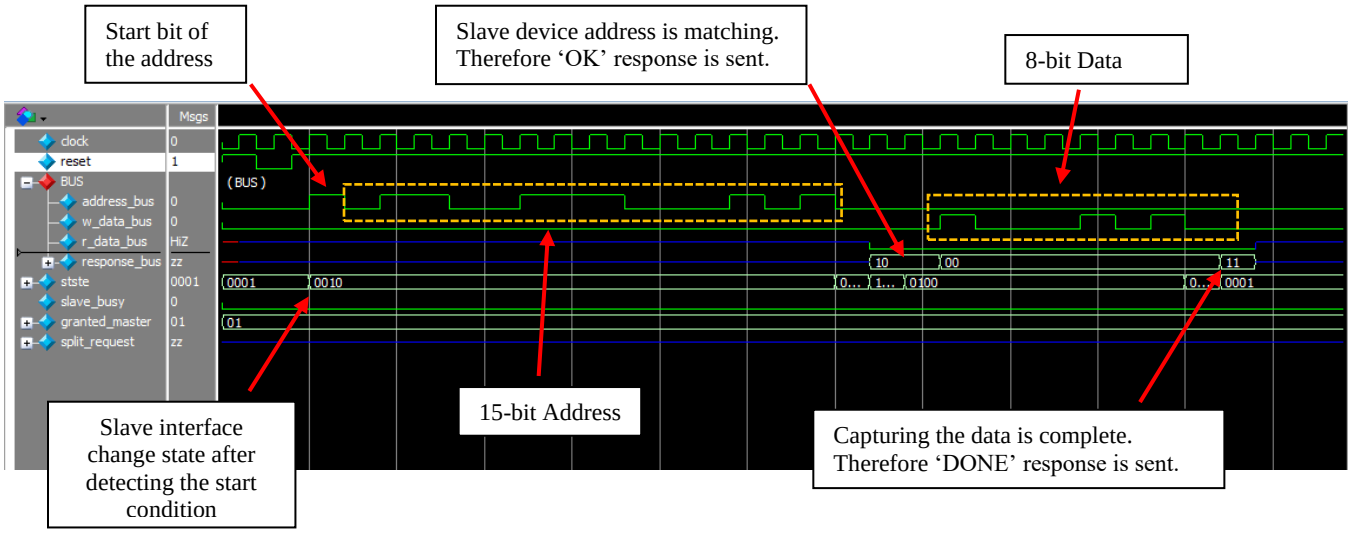
```
116 #28;
117 //------------transaction done------------------//
118
119 //--- master 01 reading from slave 11--------- -//
120 address = {1'b1,2'b11,1'b0,12'b001110001010};
121 //--- sending address
122 for (i=0;i<16;i=i+1) begin
123     address_bus=address[15-i]; #2;
124     end
125 #10;
126
127 //------------transaction done------------------//
128
129 //--- master 01 reading from slave 01-------- --//
130 //--- but the slave 01 is busy -----------------//
131 //--- sending address
132 address = {1'b1,2'b01,1'b0,12'b001110001010};
133 slave_busy=1;
134 for (i=0;i<16;i=i+1) begin
135     address_bus=address[15-i]; #2;
136     end
137 #5;
138 //--- arbiter release the master
139 granted_master=2'b00;
140 #5;
141 //--- slave become active after 10 clock periods
142 slave_busy=0;
143 #3;
144 //--- arbiter grant master 01
145 granted_master=2'b01;
146 //------------transaction done------------------//
147 end
148 endmodule
```

7.2 Simulation Diagrams

Following are screenshots extracted from the full timing diagram taken using ModelSim.

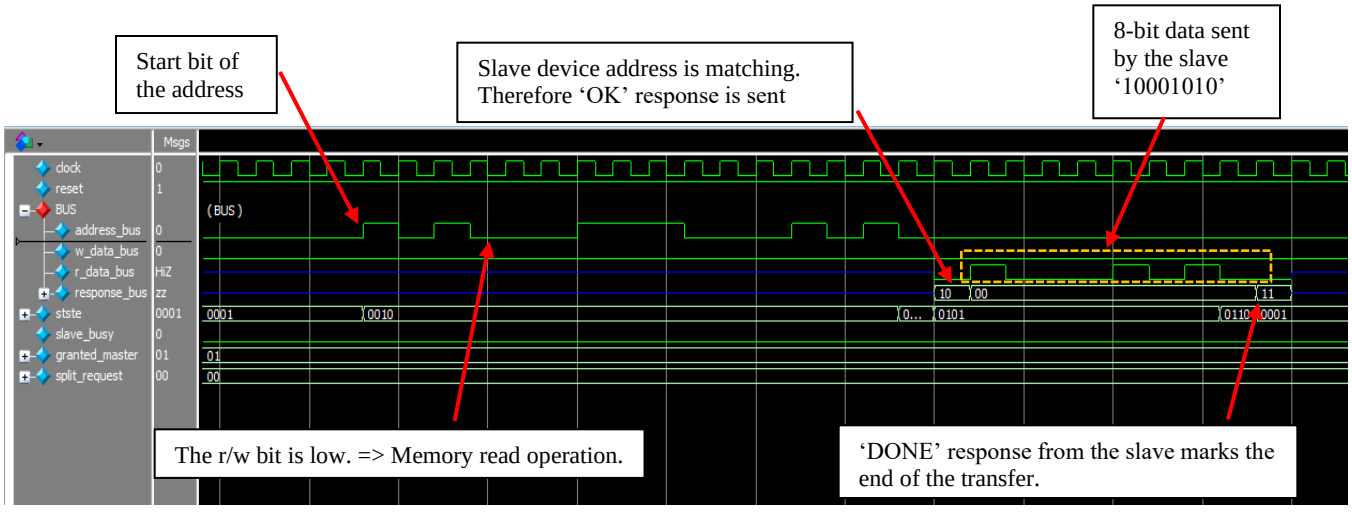
7.2.1 Master write to slave (address matching)

In the following diagram the master 01 is writing a byte of data (10001010) to the memory address 001110001010 of the slave device with device id '01'.



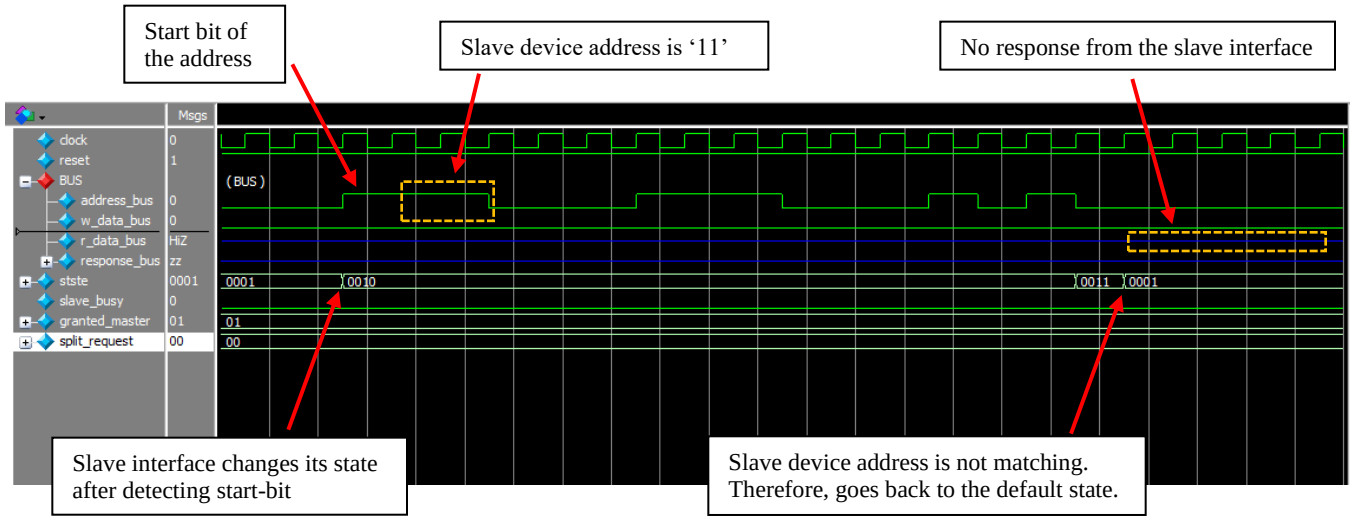
7.2.2 Master read from slave (address matching)

In the following diagram, the master 01 is reading a byte of the data from the slave memory address 001110001010. This is the same memory location where the master 01 wrote 10001010 in the first transaction. Therefore, the slave is expected to send 10001010 in the *r_data_bus.*



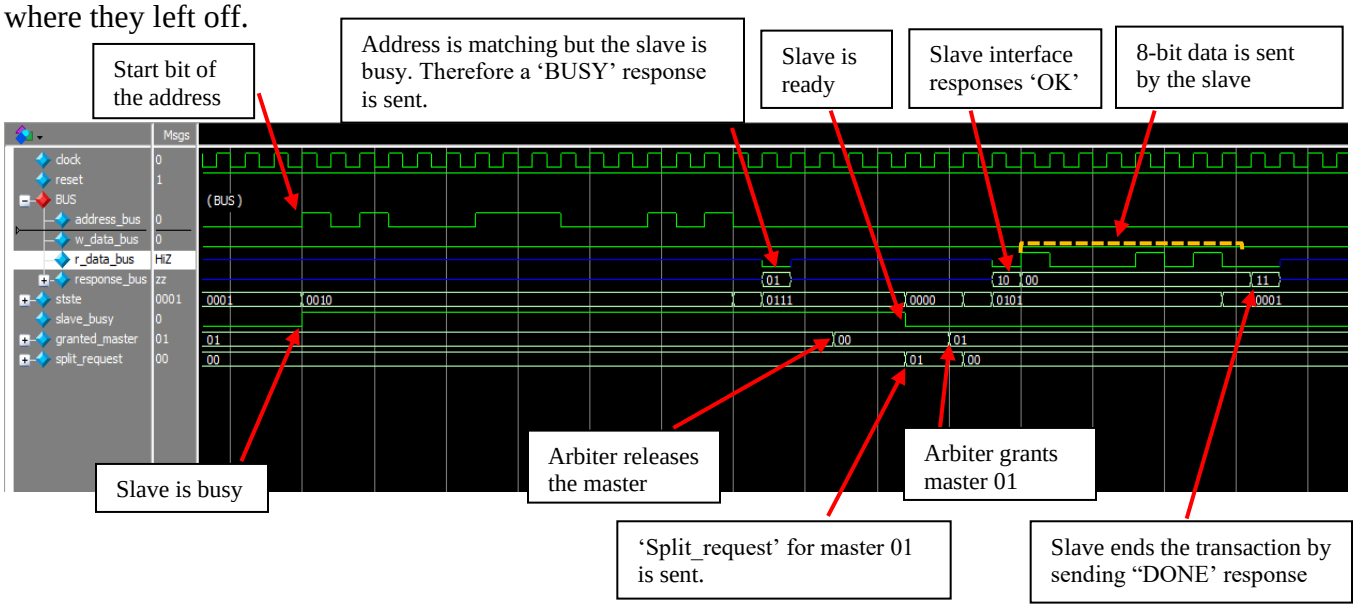
7.2.3 Master is reading from slave (address is not matching)

The following diagram demonstrate a scenario where master 01 is trying to read 8-bit data from slave with device address ‘11’ but such slave does not exist on the bus. The slave devices with different device addresses are expected to not to respond to such a master request.



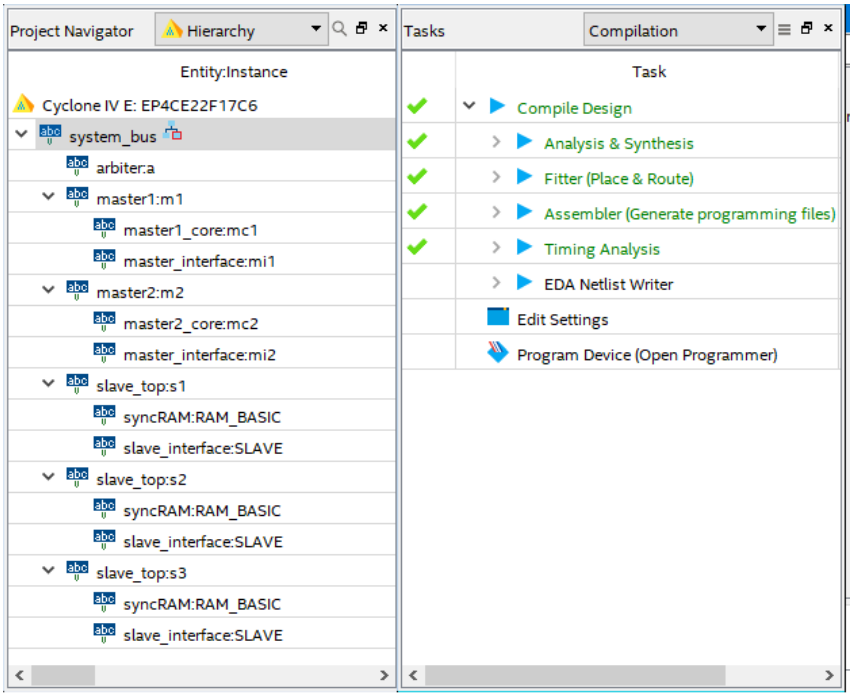
7.2.4 Master is reading from slave (slave is busy)

The following diagram shows a scenario where master 01 is trying to read from slave 01 but the slave is busy and cannot continue the transaction. In such cases, the slave is expected to issue a ‘BUSY’ response and the master will release the bus. Once the slave is ready, the slave interface will request a split transaction from the arbiter. Once the arbiter grants permission to the master, they will continue from where they left off.



8 Top Level Verification

In the final stage we integrated all the separately verified components into a final complete bus design with two masters with master1 having high priority and three slaves with 2k, 2k, 4k memory spaces. Following is the hierarchy of the design.



8.1 Top Level Testbench

```
1   module system_bus_tb;
2
3   //Only input variables from tb to the design are clock and reset
4   reg clk    = 0;
5   reg reset  = 0;
6
7   //asynchronous reset with a 2-flop synchronizer
8   reg reset_release_ff1 = 0;
9   reg reset_release_ff2 = 0;
10  wire reset_with_synchronous_release;
11  assign reset_with_synchronous_release = reset_release_ff2 && reset;
12
13  always @(posedge clk)begin
14      reset_release_ff1 <= reset;
15      reset_release_ff2 <= reset_release_ff1;
16  end
17
18  //Initially the design is in reset state
19  initial begin
20      clk = 0;
21      #25 reset = 1; //Releasing the initial reset
22      #10281 reset = 0; //Reset test in the middle
23      #28 reset = 1; //Releasing reset in the reset test
24  end
```

```
25
26  //Clock signal generation
27  always #10 clk = ~clk;
28
29  //System Bus Instantiation
30  system_bus sb(
31      .clk    (clk),
32      .reset (reset_with_synchronous_release)
33  );
34
35  endmodule
```

Testbench for the final bus design is very small because all the operations are handled in the master cores themselves. Only the clock and the reset signals were given to the system bus design.

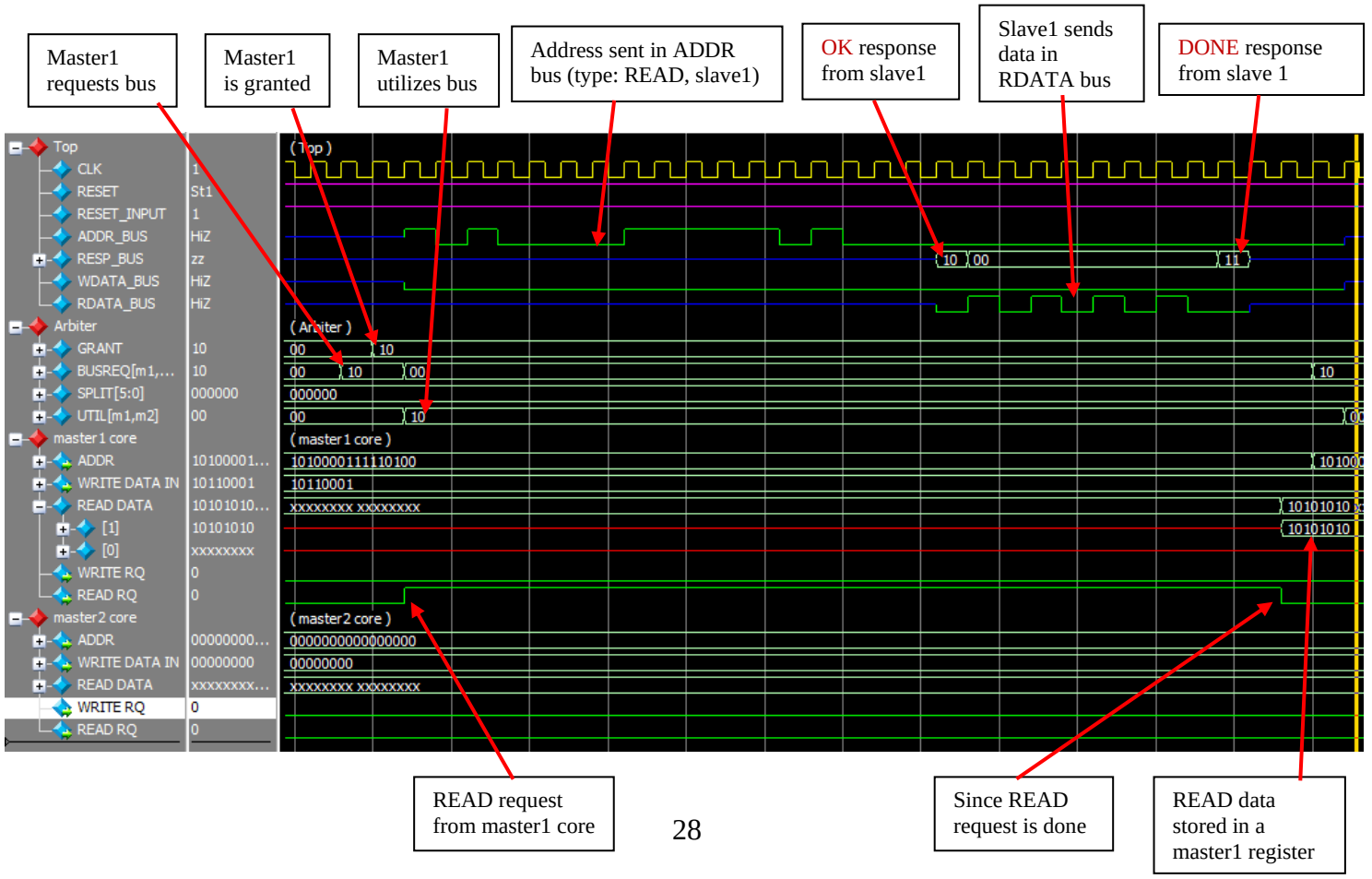
Each master interface is attached to a master core and their flow of operation of is a predefined set of read, write tasks, and idle times. (APPENDIX C shows how these 2 are interfaced together). Codes of the master1 and master2 cores, used for the top-level verification are in APPENDIX D and APPENDIX E .

8.2 Simulation Results

Following are screenshots extracted from the full timing diagram taken using ModelSim.

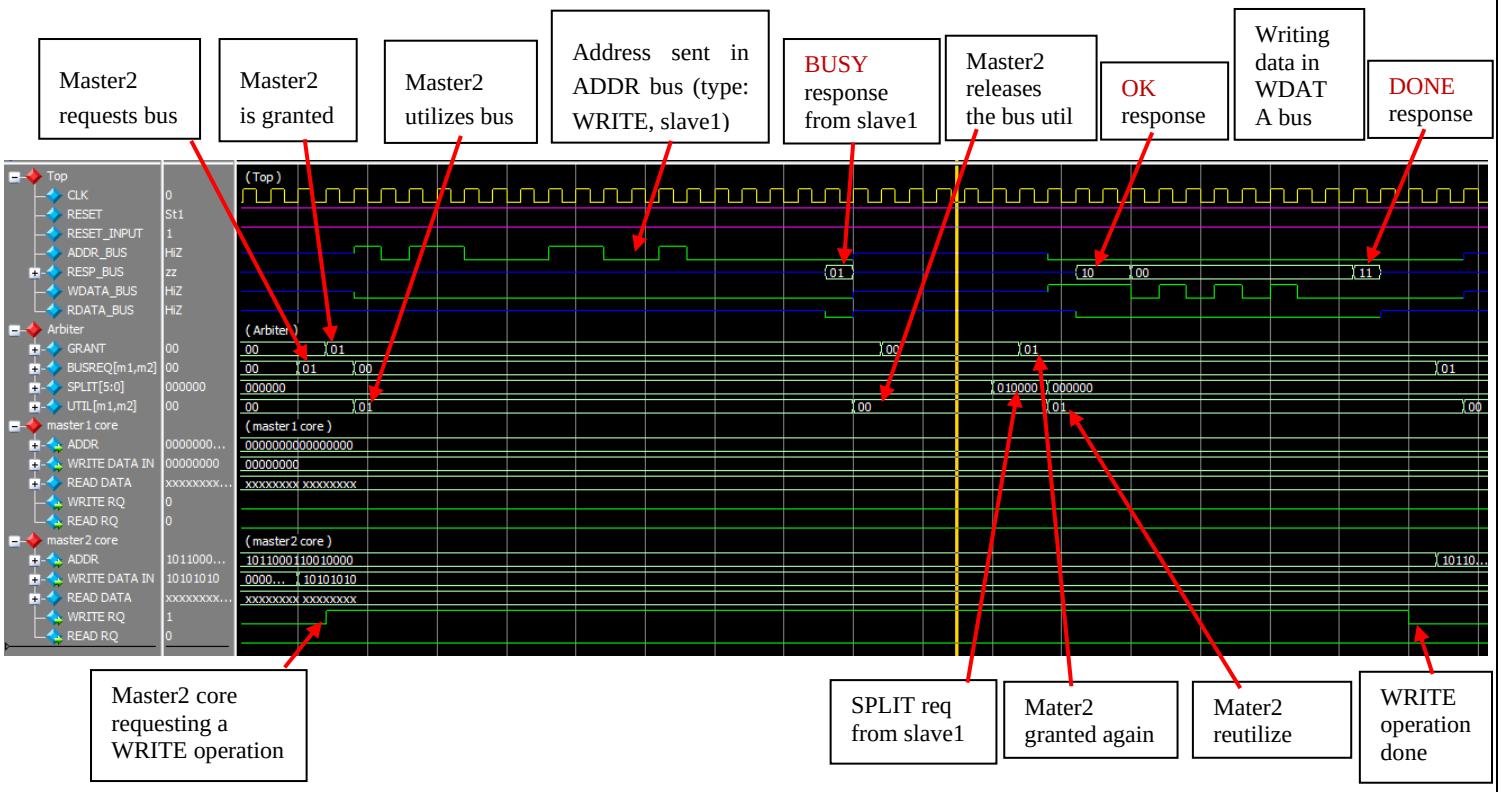
8.2.1 One master request

READ Operation



In the above scenario the address sent in the address bus is 1**010**000111110100. There the slave device address is 01 (slave 1), transfer type bit is 0 (READ transfer) and the slave memory address is 500. Therefore, master 1 is reading the 500th memory location of slave1 and storing it in a register in master1.

WRITE Operation (with a SPLIT viable scenario)



8.2.2 Two Master Request

If there are two bus requests from both masters, the moment the bus becomes unutilized, master1 will be granted because it has the highest priority. This can be easily observed when one master requesting the bus while the other master is doing a burst transaction. Following are such cases.

Master2 requesting while master1 in doing a burst

Master1 requesting while master2 in doing a burst

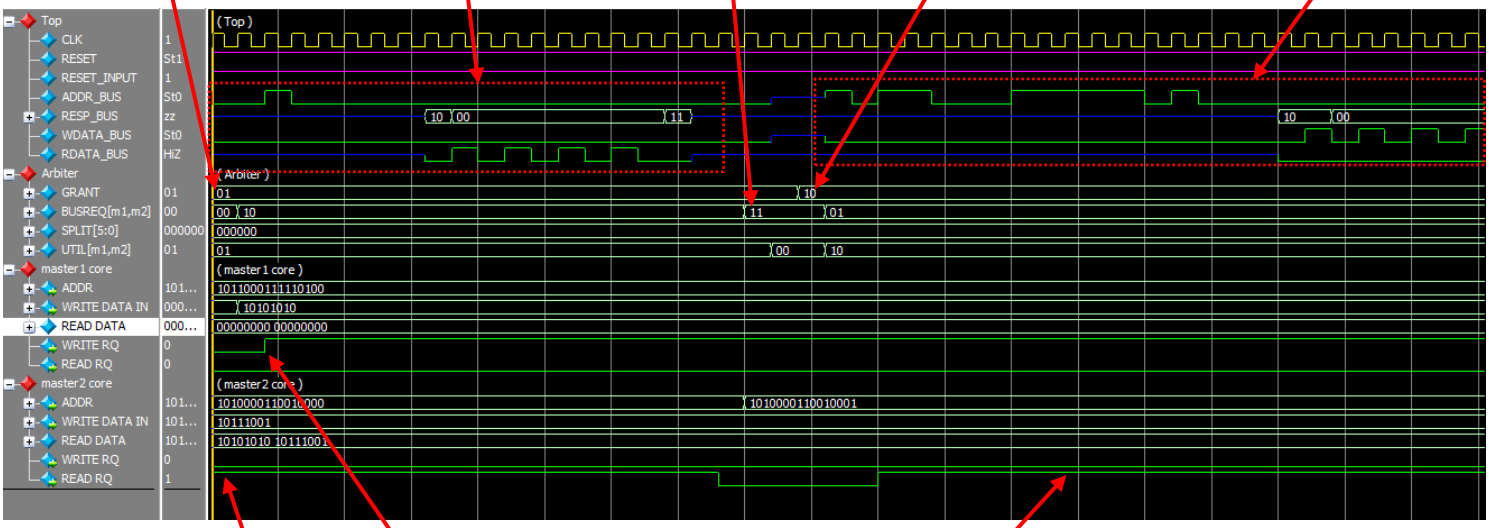
Initially master2 granted and in the process of **burst** reads

Master2's READ operation during which master1 makes a request

2 bus requests standing by

Master1 getting the grant interrupting and pausing mater2 burst since master1 has a high priority

Master1 WRITE operation



Master2 has already started the burst transaction and performing READ operations

Master1 making a write request in the middle

Next READ operation in the master2's burst operation is paused till master1 is finished

8.2.3 SPLIT Transaction Viable Scenario

Master1 requests bus

Master1 granted

Master1 utilizes

Master1 sends address

Master1 gets resp BUSY

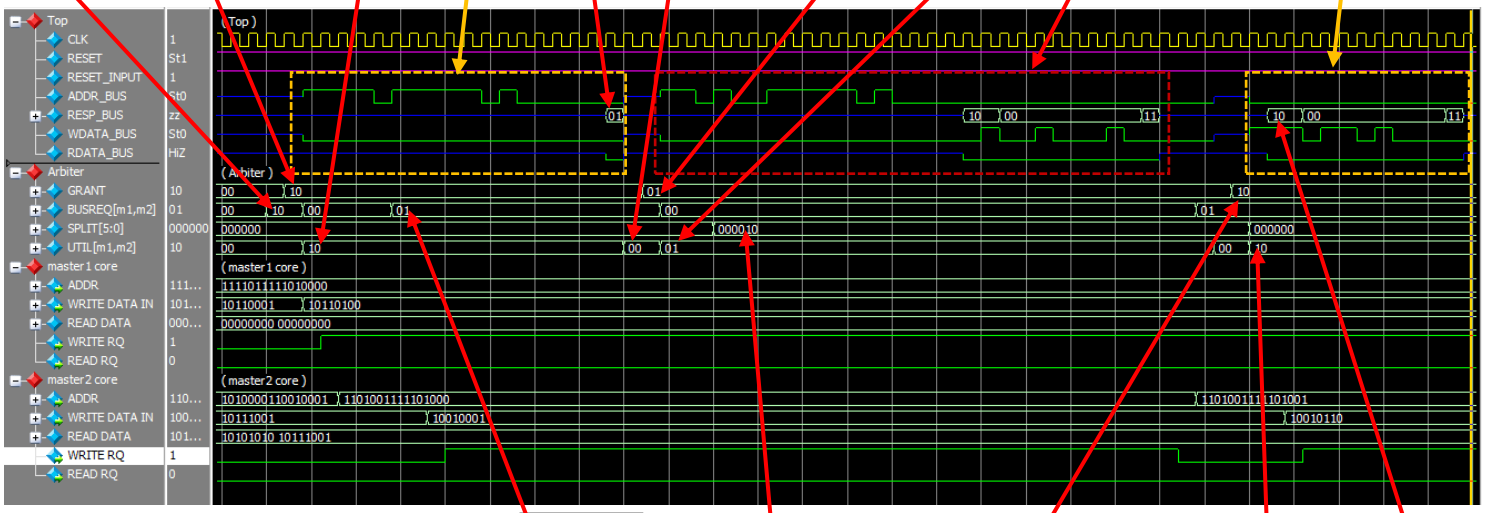
Master1 releases bus

Master 2 granted

Master 2 utilizes

Master2 completes the transaction

Master1 writes data



Master2 requests bus

Slave3 which was busy earlier, is now ready and sends a SPLIT request, requesting mater1

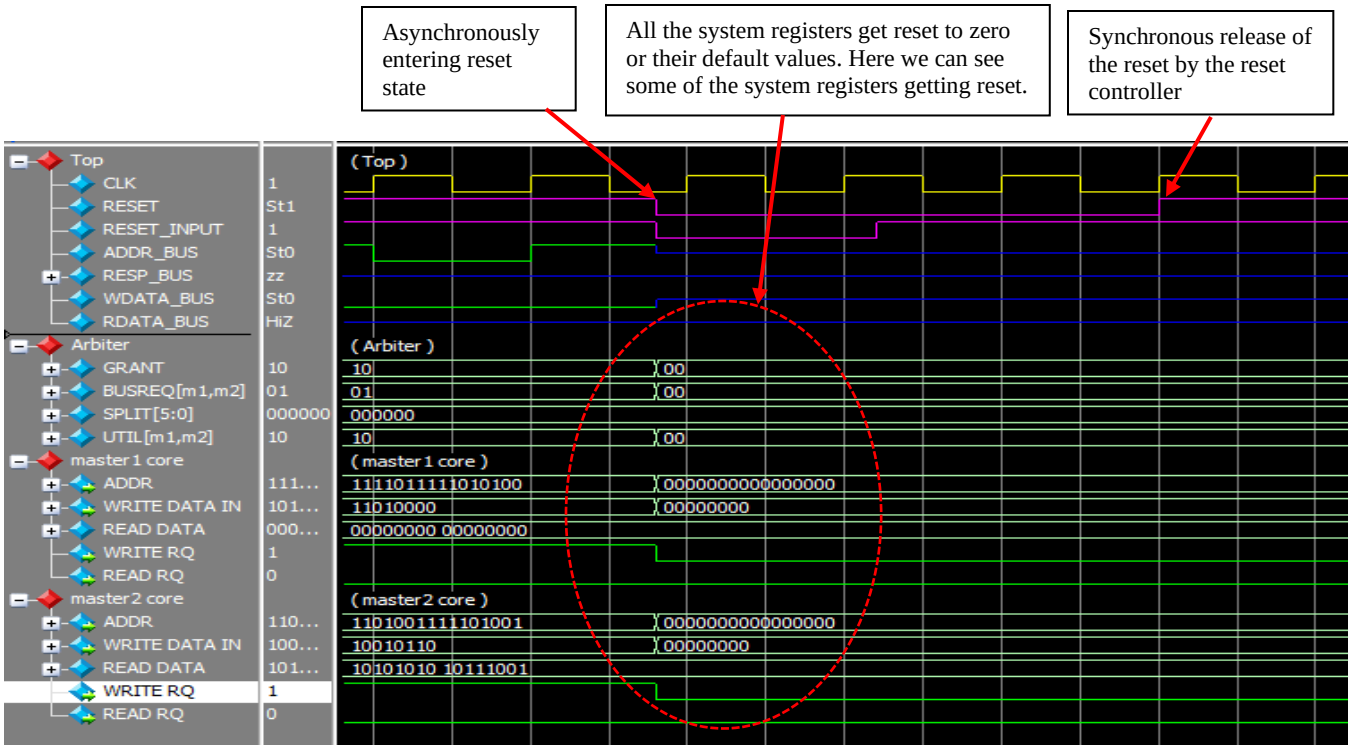
Master1 granted due to SPLIT request

Master1 utilizes bus to continue the split transaction

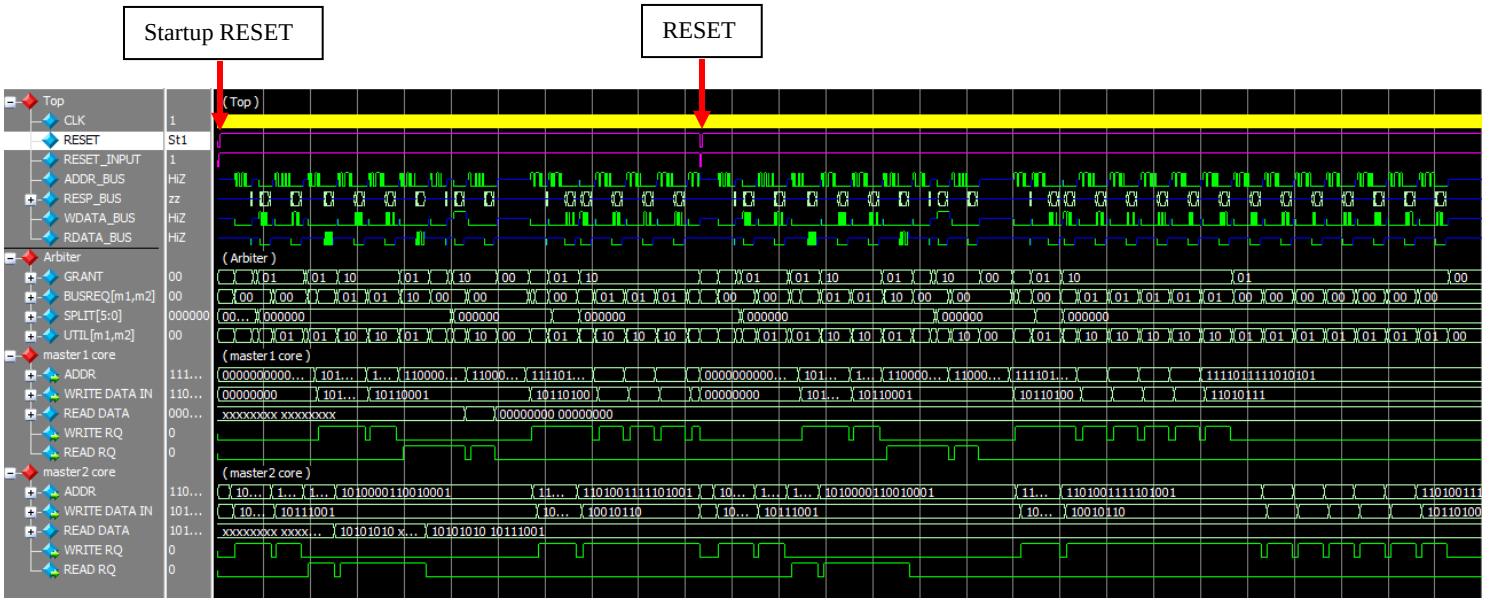
Master1 gets resp OK

8.2.4 Reset Test

Finally, while the execution sequences of the two master cores were running, a reset input was given to verify whether the whole system gets properly reset and start from the beginning.



Below diagram shows how all the execution sequences of two master cores repeat after reset.



9 APPENDIX

All the RTL design codes for of the completed system bus design made targeting the top-level verification are included in this section.

9.1 APPENDIX A

Top Level Testbench for top level verification. (system_bus_tb.v)

```
1   module system_bus_tb;
2
3   //Only input variables from tb to the design are clock and reset
4   reg clk    = 0;
5   reg reset  = 0;
6
7   //asynchronous reset with a 2-flop sychronizer
8   reg reset_release_ff1 = 0;
9   reg reset_release_ff2 = 0;
10  wire reset_with_synchronous_release;
11  assign reset_with_synchronous_release = reset_release_ff2 && reset;
12
13  always @(posedge clk)begin
14      reset_release_ff1 <= reset;
15      reset_release_ff2 <= reset_release_ff1;
16  end
17
18  //Initially the design is in reset state
19  initial begin
20      clk = 0;
21      #25 reset = 1; //Releasing the initial reset
22      #10281 reset = 0; //Reset test in the middle
23      #28 reset = 1; //Releasing reset in the reset test
24  end
25
26  //Clock signal generation
27  always #10 clk = ~clk;
28
29  //System Bus Instantiation
30  system_bus sb(
31      .clk   (clk),
32      .reset (reset_with_synchronous_release)
33  );
34
35  endmodule
```

9.2 APPENDIX B

Top level module of the system bus design (system_bus.v)

```
1   module system_bus(
2       input clk,
3       input reset
4   );
5
6   //Connection wires with arbiter
7   wire  [1:0]   GRANT;
8   wire  [1:0]   BUSREQ;
```

```
9   wire  [1:0]   UTIL;
10  wire  [5:0]   GMASTER;
11  wire  [5:0]   SPLIT;
12
13
14  //Connection wires of the BUS
15  wire      rdata_bus;
16  wire [1:0]response_bus;
17  wire      addr_bus;
18  wire      wdata_bus;
19
20  //Slave addresses
21  reg [1:0] slave1_addr = 2'b01;
22  reg [1:0] slave2_addr = 2'b10;
23  reg [1:0] slave3_addr = 2'b11;
24
25  //arbter instantiation
26  arbiter a(
27      .clk                            (clk),
28      .reset                          (reset),
29      .req_from_master                (BUSREQ),
30      .bus_utilization                (UTIL),
31      .grant_to_master                (GRANT),
32      .notify_granted_master_to_slave(GMASTER),
33      .split_req_from_slave           (SPLIT)
34  );
35
36  //master1 instantiation
37  master1 m1(
38      .clk              (clk),
39      .reset            (reset),
40      .from_arb_grant   (GRANT[1]),
41      .to_arb_req_bus   (BUSREQ[1]),
42      .to_arb_bus_util  (UTIL[1]),
43      .to_addr_bus      (addr_bus),
44      .to_wdata_bus     (wdata_bus),
45      .from_rdata_bus   (rdata_bus),
46      .from_response_bus(response_bus)
47  );
48
49  //master2 instantiation
50  master2 m2(
51      .clk              (clk),
52      .reset            (reset),
53      .from_arb_grant   (GRANT[0]),
54      .to_arb_req_bus   (BUSREQ[0]),
55      .to_arb_bus_util  (UTIL[0]),
56      .to_addr_bus      (addr_bus),
57      .to_wdata_bus     (wdata_bus),
58      .from_rdata_bus   (rdata_bus),
59      .from_response_bus(response_bus)
60  );
61
62  //slave1 instantiation
63  slave_top s1(
64      .clock50       (clk),
65      .reset         (reset),
66      .address_bus   (addr_bus),
67      .w_data_bus    (wdata_bus),
68      .r_data_bus    (rdata_bus),
69      .response_bus  (response_bus),
70      .split_request (SPLIT[5:4]),
71      .granted_master(GMASTER[5:4]),
```

```
72      .slave_address (slave1_addr)
73  );
74
75  //slave2 instantiation
76  slave_top s2(
77      .clock50       (clk),
78      .reset         (reset),
79      .address_bus   (addr_bus),
80      .w_data_bus    (wdata_bus),
81      .r_data_bus    (rdata_bus),
82      .response_bus  (response_bus),
83      .split_request (SPLIT[3:2]),
84      .granted_master(GMASTER[3:2]),
85      .slave_address (slave2_addr)
86  );
87
88  //slave3 instantiation
89  slave_top s3(
90      .clock50       (clk),
91      .reset         (reset),
92      .address_bus   (addr_bus),
93      .w_data_bus    (wdata_bus),
94      .r_data_bus    (rdata_bus),
95      .response_bus  (response_bus),
96      .split_request (SPLIT[1:0]),
97      .granted_master(GMASTER[1:0]),
98      .slave_address (slave3_addr)
99  );
100
101 endmodule
```

9.3 APPENDIX C

Master1 top module connecting master1 core module and master interface together(master1.v)

```
1   module master1(
2   input       clk,
3   input       reset,
4   input       from_arb_grant,
5   output      to_arb_req_bus,
6   output      to_arb_bus_util,
7   output tri  to_addr_bus,
8   output tri  to_wdata_bus,
9   input       from_rdata_bus,
10  input [1:0]from_response_bus
11  );
12
13  wire mibufin_addr;
14  wire mibufin_wdata;
15  wire mibufin_addr_en;
16  wire mibufin_wdata_en;
17
18  //Connection Wires
19  wire [15:0] mmi_addr;
20  wire        mmi_write_addr_req;
21  wire  [7:0] mmi_write_data;
22  wire        mmi_write_data_req;
23  wire        mmi_read_data_req;
24  wire        mmi_ok_response;
25  wire  [7:0] mmi_read_data;
26  wire        mmi_req_done;
```

```
wire        mmi_force_req;

bufif1  b_addr(to_addr_bus, mibufin_addr, mibufin_addr_en);
bufif1  b_wdata(to_wdata_bus, mibufin_wdata, mibufin_wdata_en);

//Master1 core instantiation
master1_core mc1(
    .clk                        (clk),
    .reset                      (reset),
    .addr_to_mi                 (mmi_addr),
    .write_addr_req_to_mi       (mmi_write_addr_req),
    .write_data_to_mi           (mmi_write_data),
    .write_data_req_to_mi       (mmi_write_data_req),
    .read_data_req_to_mi        (mmi_read_data_req),
    .ok_response_from_mi        (mmi_ok_response),
    .read_data_from_mi          (mmi_read_data),
    .req_done_from_mi           (mmi_req_done),
    .force_req_to_mi            (mmi_force_req)
);

//Master Interface instantiation for master1
master_interface mi1(
    .clk                        (clk),
    .reset                      (reset),
    .addr                       (mibufin_addr),
    .wdata                      (mibufin_wdata),
    .rdata                      (from_rdata_bus),
    .response                   (from_response_bus),
    .bus_req                        (to_arb_req_bus),
    .grant                      (from_arb_grant),
    .util                       (to_arb_bus_util),

    .addr_from_master           (mmi_addr),
    .write_addr_req_from_master (mmi_write_addr_req),
    .notify_ok_response_to_master(mmi_ok_response),
    .write_data_from_master     (mmi_write_data),
    .read_data_to_master        (mmi_read_data),
    .write_data_req_from_master (mmi_write_data_req),
    .read_data_req_from_master  (mmi_read_data_req),
    .req_done_to_master         (mmi_req_done),
    .force_req_from_master      (mmi_force_req),

    .addr_en                    (mibufin_addr_en),
    .wdata_en                   (mibufin_wdata_en)
);

endmodule
```

9.4 APPENDIX D

Master1 core module which executes a sequence of READ, WRITE operations (master1_core.v)

```
module master1_core (
    input clk,
    input reset,

    //connections with the ports of master interface to the the master side
    //NOTE : mi - master interface
    output reg [15:0]addr_to_mi,
    output reg write_addr_req_to_mi = 0,
    output reg [7:0]write_data_to_mi,
```

```
10      output reg write_data_req_to_mi = 0,
11      output reg read_data_req_to_mi  = 0,
12      output reg force_req_to_mi = 0,
13      input ok_response_from_mi,
14      input [7:0]read_data_from_mi,
15      input req_done_from_mi
16  );
17
18  //Address Parameters
19  parameter STARTBIT = 1'b1;
20  parameter SLAVE1 = 2'b01, SLAVE2 = 2'b10, SLAVE3 = 2'b11;
21  parameter WRITE = 1'b1, READ = 1'b0;
22
23  /*_____________________________________________________________________________
24  | ---------------------------------------------------------------------------|
25  |EXECUTION:
26  |Flow of execution of master 1 is a predefined set of read, write tasks and idle times.
27  |Flow starts from WRITE1 and ends with WRITE2 below, but we can customize any sequence.
28  |Below sequence has the following order,
29
30  |          INITIAL_IDLE -> WRITE1 -> READ1 -> IDLE1 -> WRIITE2 -> END
31  |
32  |We can change parameters of those 4 tasks from below list of parameters.
33  |_____________________________________________________________________________*/
34
35  //Execution Sequence Parameters
36  parameter INITIAL_IDLE_TIME = 8'd100;
37
38  parameter WRITE1_SLAVE      = SLAVE1;
39  parameter WRITE1_START_ADDR = 12'd500;
40  parameter WRITE1_DATA       = 8'd170;
41  parameter WRITE1_DATA_INCR  = 7;
42  parameter WRITE1_SIZE       = 4'd2;
43
44  parameter READ1_SLAVE       = SLAVE2;
45  parameter READ1_START_ADDR  = 12'd400;
46  parameter READ1_SIZE        = 2'd2;
47
48  parameter IDLE1_CYCLES      = 8'd30;
49
50  parameter WRITE2_SLAVE      = SLAVE3;
51  parameter WRITE2_START_ADDR = 12'd2000;
52  parameter WRITE2_DATA       = 8'd180;
53  parameter WRITE2_DATA_INCR  = 7;
54  parameter WRITE2_SIZE       = 4'd6;
55
56
57  //Registers Needed for the Execution Sequence
58  reg [5:0]  state              = 5'd0;
59
60  reg [7:0]  initial_idle_time  = INITIAL_IDLE_TIME;
61
62  reg [11:0] write1_addr        = WRITE1_START_ADDR;
63  reg [7:0]  write1_data        = WRITE1_DATA;
64  reg [3:0]  write1_size        = WRITE1_SIZE;
65
66  reg [11:0] read1_addr         = READ1_START_ADDR;
67  reg [3:0]  read1_size         = READ1_SIZE;
68  reg [7:0]  read1_data[1:0];
69
70  reg [7:0]  idle1_cycles       = IDLE1_CYCLES;
71
72  reg [11:0] write2_addr        = WRITE2_START_ADDR;
```

```
73  reg [7:0]  write2_data        = WRITE2_DATA;
74  reg [3:0]  write2_size        = WRITE2_SIZE;
75
76
77  /*################################## STATE MACHINE ##########################
78  ---------------------------------------------------------------------------*/
79  //STATES:
80  parameter INITIAL_IDLE                  = 0;
81
82  parameter WRITE1_UPDATE_ADDR            = 1;
83  parameter WRITE1_ADDR_RQ                = 2;
84  parameter WRITE1_OK_RES                 = 3;
85  parameter WRITE1_UPDATE_DATA            = 4;
86  parameter WRITE1_DATA_RQ                = 5;
87  parameter WRITE1_RQ_DONE                = 6;
88
89  parameter READ1_UPDATE_ADDR             = 7;
90  parameter READ1_ADDR_RQ                 = 8;
91  parameter READ1_OK_RES                  = 9;
92  parameter READ1_DATA_RQ                 = 10;
93  parameter READ1_RQ_DONE                 = 11;
94
95  parameter IDLE1                         = 12;
96
97  parameter WRITE2_UPDATE_ADDR            = 13;
98  parameter WRITE2_ADDR_RQ                = 14;
99  parameter WRITE2_OK_RES                 = 15;
100 parameter WRITE2_UPDATE_DATA            = 16;
101 parameter WRITE2_DATA_RQ                = 17;
102 parameter WRITE2_RQ_DONE                = 18;
103
104 parameter ALL_DONE                      = 19;
105
106 //STATE MACHINE:
107 always@(posedge clk or negedge reset) begin
108     if(~reset)begin
109         addr_to_mi           <= 0;
110         write_addr_req_to_mi <= 0;
111         write_data_to_mi     <= 0;
112         write_data_req_to_mi <= 0;
113         read_data_req_to_mi  <= 0;
114         force_req_to_mi      <= 0;
115
116         state                <= 5'd0;
117         initial_idle_time    <= INITIAL_IDLE_TIME;
118         write1_addr          <= WRITE1_START_ADDR;
119         write1_data          <= WRITE1_DATA;
120         write1_size          <= WRITE1_SIZE;
121         read1_addr           <= READ1_START_ADDR;
122         read1_size           <= READ1_SIZE;
123         idle1_cycles         <= IDLE1_CYCLES;
124         write2_addr          <= WRITE2_START_ADDR;
125         write2_data          <= WRITE2_DATA;
126         write2_size          <= WRITE2_SIZE;
127     end
128
129     else begin
130         case(state)
131
132             //###################### INITIAL IDLE TIME ##################
133             INITIAL_IDLE: begin
134                 if(initial_idle_time > 0)
135                     initial_idle_time <= initial_idle_time -1;
```

```
136                 else
137                     state <= WRITE1_UPDATE_ADDR;
138             end
139
140             //############################ WRITE1 ########################
141             WRITE1_UPDATE_ADDR: begin
142                 if(write1_size == 0)begin
143                     state <= READ1_UPDATE_ADDR;
144                 end
145                 else begin
146                     force_req_to_mi <= 0;
147                     write1_size <= write1_size-1;
148                     addr_to_mi <= {STARTBIT, WRITE1_SLAVE, WRITE, write1_addr};
149                     write1_addr <= write1_addr + 1;
150                     state <= WRITE1_ADDR_RQ;
151                 end
152             end
153
154             WRITE1_ADDR_RQ: begin
155                 write_addr_req_to_mi <= 1;
156                 state <= WRITE1_OK_RES;
157             end
158
159             WRITE1_OK_RES: begin
160                 if(ok_response_from_mi == 1)begin
161                     write_addr_req_to_mi <= 0;
162                     state <= WRITE1_UPDATE_DATA;
163                 end
164             end
165
166             WRITE1_UPDATE_DATA: begin
167                 write_data_to_mi <= write1_data;
168                 write1_data <= write1_data + WRITE1_DATA_INCR;
169                 state <= WRITE1_DATA_RQ;
170             end
171
172             WRITE1_DATA_RQ: begin
173                 write_data_req_to_mi <= 1;
174                 state <= WRITE1_RQ_DONE;
175             end
176             WRITE1_RQ_DONE: begin
177                 if(req_done_from_mi == 1) begin
178                     write_data_req_to_mi <= 0;
179                     state <= WRITE1_UPDATE_ADDR;
180
181                     if(write1_size != 0)begin
182                         force_req_to_mi <= 1;
183                     end
184                 end
185             end
186
187
188             //############################ READ1 ########################
189             READ1_UPDATE_ADDR: begin
190                 if(read1_size == 0)begin
191                     state <= IDLE1;
192                 end
193                 else begin
194                     force_req_to_mi <= 0;
195                     read1_size  <= read1_size-1;
196                     addr_to_mi  <= {STARTBIT, READ1_SLAVE, READ, read1_addr};
197                     read1_addr  <= read1_addr + 1;
198                     state <= READ1_ADDR_RQ;
```

```
199                 end
200             end
201
202             READ1_ADDR_RQ: begin
203                 write_addr_req_to_mi <= 1;
204                 state <= READ1_OK_RES;
205             end
206
207             READ1_OK_RES: begin
208                 if(ok_response_from_mi == 1)begin
209                     write_addr_req_to_mi <= 0;
210                     state <= READ1_DATA_RQ;
211                 end
212             end
213
214             READ1_DATA_RQ: begin
215                 read_data_req_to_mi <= 1;
216                 state <= READ1_RQ_DONE;
217             end
218
219             READ1_RQ_DONE: begin
220                 if(req_done_from_mi == 1) begin
221                     read_data_req_to_mi <= 0;
222                     read1_data[read1_size] <= read_data_from_mi;
223                     state <= READ1_UPDATE_ADDR;
224
225                     if(read1_size != 0)begin
226                         force_req_to_mi <= 1;
227                     end
228                 end
229             end
230
231
232             //############################ IDLE1 ########################
233             IDLE1: begin
234                 if(idle1_cycles == 0) begin
235                     state <= WRITE2_UPDATE_ADDR;
236                 end
237                 idle1_cycles <= idle1_cycles -1;
238             end
239
240
241             //############################ WRITE2 ########################
242             WRITE2_UPDATE_ADDR: begin
243                 if(write2_size == 0)begin
244                     state <= ALL_DONE;
245                 end
246                 else begin
247                     force_req_to_mi <= 0;
248                     write2_size <= write2_size-1;
249                     addr_to_mi <= {STARTBIT, WRITE2_SLAVE, WRITE, write2_addr};
250                     write2_addr <= write2_addr + 1;
251                     state <= WRITE2_ADDR_RQ;
252                 end
253             end
254
255             WRITE2_ADDR_RQ: begin
256                 write_addr_req_to_mi <= 1;
257                 state <= WRITE2_OK_RES;
258             end
259
260             WRITE2_OK_RES: begin
261                 if(ok_response_from_mi == 1)begin
```

```
                    write_addr_req_to_mi <= 0;
                    state <= WRITE2_UPDATE_DATA;
                end
            end

            WRITE2_UPDATE_DATA: begin
                write_data_to_mi <= write2_data;
                write2_data <= write2_data + WRITE2_DATA_INCR;
                state <= WRITE2_DATA_RQ;
            end

            WRITE2_DATA_RQ: begin
                write_data_req_to_mi <= 1;
                state <= WRITE2_RQ_DONE;
            end
            WRITE2_RQ_DONE: begin
                if(req_done_from_mi == 1) begin
                    write_data_req_to_mi <= 0;
                    state <= WRITE2_UPDATE_ADDR;

                    if(write2_size != 0)begin
                        force_req_to_mi <= 1;
                    end
                end
            end


            //############################## END ##########################
            ALL_DONE: begin
                state <= ALL_DONE;
            end

        endcase
    end

end


endmodule
```

9.5 APPENDIX E

Master2 top module connecting master2 core module and master interface together(master2.v)

```
module master2(
input       clk,
input       reset,
input       from_arb_grant,
output      to_arb_req_bus,
output      to_arb_bus_util,
output tri  to_addr_bus,
output tri  to_wdata_bus,
input       from_rdata_bus,
input [1:0]from_response_bus
);

wire mibufin_addr;
wire mibufin_wdata;
wire mibufin_addr_en;
wire mibufin_wdata_en;

```

```
18  //Connection Wires
19  wire [15:0] mmi_addr;
20  wire        mmi_write_addr_req;
21  wire  [7:0] mmi_write_data;
22  wire        mmi_write_data_req;
23  wire        mmi_read_data_req;
24  wire        mmi_ok_response;
25  wire  [7:0] mmi_read_data;
26  wire        mmi_req_done;
27  wire        mmi_force_req;
28
29  bufif1  b_addr(to_addr_bus, mibufin_addr, mibufin_addr_en);
30  bufif1  b_wdata(to_wdata_bus, mibufin_wdata, mibufin_wdata_en);
31
32  //Master2 core instantiation
33  master2_core mc2(
34      .clk                          (clk),
35      .reset                        (reset),
36      .addr_to_mi                   (mmi_addr),
37      .write_addr_req_to_mi         (mmi_write_addr_req),
38      .write_data_to_mi             (mmi_write_data),
39      .write_data_req_to_mi         (mmi_write_data_req),
40      .read_data_req_to_mi          (mmi_read_data_req),
41      .ok_response_from_mi          (mmi_ok_response),
42      .read_data_from_mi            (mmi_read_data),
43      .req_done_from_mi             (mmi_req_done),
44      .force_req_to_mi              (mmi_force_req)
45  );
46
47  //Master Interface instantiation for master2
48  master_interface mi2(
49      .clk                          (clk),
50      .reset                        (reset),
51      .addr                         (mibufin_addr),
52      .wdata                        (mibufin_wdata),
53      .rdata                        (from_rdata_bus),
54      .response                     (from_response_bus),
55      .bus_req                          (to_arb_req_bus),
56      .grant                        (from_arb_grant),
57      .util                         (to_arb_bus_util),
58
59      .addr_from_master             (mmi_addr),
60      .write_addr_req_from_master   (mmi_write_addr_req),
61      .notify_ok_response_to_master(mmi_ok_response),
62      .write_data_from_master       (mmi_write_data),
63      .read_data_to_master          (mmi_read_data),
64      .write_data_req_from_master   (mmi_write_data_req),
65      .read_data_req_from_master    (mmi_read_data_req),
66      .req_done_to_master           (mmi_req_done),
67      .force_req_from_master        (mmi_force_req),
68
69      .addr_en                      (mibufin_addr_en),
70      .wdata_en                     (mibufin_wdata_en)
71  );
72
73  endmodule
```

9.6 APPENDIX F

Master2 core module which executes a sequence of READ, WRITE operations (master2_core.v)

```
1   module master2_core (
2       input clk,
3       input reset,
4
5       //connections with the ports of master interface to the the master side
6       //NOTE : mi - master interface
7       output reg [15:0]addr_to_mi,
8       output reg write_addr_req_to_mi = 0,
9       output reg [7:0]write_data_to_mi,
10      output reg write_data_req_to_mi = 0,
11      output reg read_data_req_to_mi  = 0,
12      output reg force_req_to_mi = 0,
13      input ok_response_from_mi,
14      input [7:0]read_data_from_mi,
15      input req_done_from_mi
16  );
17
18  //Address Parameters
19  parameter STARTBIT = 1'b1;
20  parameter SLAVE1 = 2'b01, SLAVE2 = 2'b10, SLAVE3 = 2'b11;
21  parameter WRITE = 1'b1, READ = 1'b0;
22
23  /*_____________________________________________________________________________
24  | ------------------------------------------------------------------------------|
25  |EXECUTION:
26  |Flow of execution of master 2 is a predefined set of read, write tasks and idle times.
27  |Flow starts from WRITE1 and ends with WRITE2 below, but we can customize any sequence.
28  |Below sequence has the following order,
29
30  |          INITIAL_IDLE -> WRITE1 -> READ1 -> IDLE1 -> WRITE2 -> END
31  |
32  |We can change parameters of those 4 tasks from below list of parameters.
33  |_____________________________________________________________________________*/
34
35  //Execution Sequence Parameters
36  parameter INITIAL_IDLE_TIME = 8'd10;
37
38  parameter WRITE1_SLAVE      = SLAVE1;
39  parameter WRITE1_START_ADDR = 12'd400;
40  parameter WRITE1_DATA       = 8'd170;
41  parameter WRITE1_DATA_INCR  = 15;
42  parameter WRITE1_SIZE       = 4'd2;
43
44  parameter READ1_SLAVE       = SLAVE1;
45  parameter READ1_START_ADDR  = 12'd400;
46  parameter READ1_SIZE        = 2'd2;
47
48  parameter IDLE1_CYCLES      = 8'd110;
49
50  parameter WRITE2_SLAVE      = SLAVE2;
51  parameter WRITE2_START_ADDR = 12'd1000;
52  parameter WRITE2_DATA       = 8'd145;
53  parameter WRITE2_DATA_INCR  = 5;
54  parameter WRITE2_SIZE       = 4'd8;
55
56
57  //Registers Needed for the Execution Sequence
58  reg [5:0]  state              = 5'd0;
```

```
59
60  reg [7:0]  initial_idle_time  = INITIAL_IDLE_TIME;
61
62  reg [11:0] write1_addr        = WRITE1_START_ADDR;
63  reg [7:0]  write1_data        = WRITE1_DATA;
64  reg [3:0]  write1_size        = WRITE1_SIZE;
65
66  reg [11:0] read1_addr         = READ1_START_ADDR;
67  reg [3:0]  read1_size         = READ1_SIZE;
68  reg [7:0]  read1_data[1:0];
69
70  reg [7:0]  idle1_cycles       = IDLE1_CYCLES;
71
72  reg [11:0] write2_addr        = WRITE2_START_ADDR;
73  reg [7:0]  write2_data        = WRITE2_DATA;
74  reg [3:0]  write2_size        = WRITE2_SIZE;
75
76
77  /*################################### STATE MACHINE ##########################
78  ----------------------------------------------------------------------------*/
79  //STATES:
80  parameter INITIAL_IDLE                  = 0;
81
82  parameter WRITE1_UPDATE_ADDR            = 1;
83  parameter WRITE1_ADDR_RQ                = 2;
84  parameter WRITE1_OK_RES                 = 3;
85  parameter WRITE1_UPDATE_DATA            = 4;
86  parameter WRITE1_DATA_RQ                = 5;
87  parameter WRITE1_RQ_DONE                = 6;
88
89  parameter READ1_UPDATE_ADDR             = 7;
90  parameter READ1_ADDR_RQ                 = 8;
91  parameter READ1_OK_RES                  = 9;
92  parameter READ1_DATA_RQ                 = 10;
93  parameter READ1_RQ_DONE                 = 11;
94
95  parameter IDLE1                         = 12;
96
97  parameter WRITE2_UPDATE_ADDR            = 13;
98  parameter WRITE2_ADDR_RQ                = 14;
99  parameter WRITE2_OK_RES                 = 15;
100 parameter WRITE2_UPDATE_DATA            = 16;
101 parameter WRITE2_DATA_RQ                = 17;
102 parameter WRITE2_RQ_DONE                = 18;
103
104 parameter ALL_DONE                      = 19;
105
106 //STATE MACHINE:
107 always@(posedge clk or negedge reset) begin
108     if(~reset)begin
109         addr_to_mi           <= 0;
110         write_addr_req_to_mi <= 0;
111         write_data_to_mi     <= 0;
112         write_data_req_to_mi <= 0;
113         read_data_req_to_mi  <= 0;
114         force_req_to_mi      <= 0;
115
116         state                <= 5'd0;
117         initial_idle_time    <= INITIAL_IDLE_TIME;
118         write1_addr          <= WRITE1_START_ADDR;
119         write1_data          <= WRITE1_DATA;
120         write1_size          <= WRITE1_SIZE;
121         read1_addr           <= READ1_START_ADDR;
```

```
122         read1_size          <= READ1_SIZE;
123         idle1_cycles        <= IDLE1_CYCLES;
124         write2_addr         <= WRITE2_START_ADDR;
125         write2_data         <= WRITE2_DATA;
126         write2_size         <= WRITE2_SIZE;
127     end
128
129     else begin
130         case(state)
131
132             //######################## INITIAL IDLE TIME ######################
133             INITIAL_IDLE: begin
134                 if(initial_idle_time > 0)
135                     initial_idle_time <= initial_idle_time -1;
136                 else
137                     state <= WRITE1_UPDATE_ADDR;
138             end
139
140             //############################ WRITE1 ##########################
141             WRITE1_UPDATE_ADDR: begin
142                 if(write1_size == 0)begin
143                     state <= READ1_UPDATE_ADDR;
144                 end
145                 else begin
146                     force_req_to_mi <= 0;
147                     write1_size <= write1_size-1;
148                     addr_to_mi <= {STARTBIT, WRITE1_SLAVE, WRITE, write1_addr};
149                     write1_addr <= write1_addr + 1;
150                     state <= WRITE1_ADDR_RQ;
151                 end
152             end
153
154             WRITE1_ADDR_RQ: begin
155                 write_addr_req_to_mi <= 1;
156                 state <= WRITE1_OK_RES;
157             end
158
159             WRITE1_OK_RES: begin
160                 if(ok_response_from_mi == 1)begin
161                     write_addr_req_to_mi <= 0;
162                     state <= WRITE1_UPDATE_DATA;
163                 end
164             end
165
166             WRITE1_UPDATE_DATA: begin
167                 write_data_to_mi <= write1_data;
168                 write1_data <= write1_data + WRITE1_DATA_INCR;
169                 state <= WRITE1_DATA_RQ;
170             end
171
172             WRITE1_DATA_RQ: begin
173                 write_data_req_to_mi <= 1;
174                 state <= WRITE1_RQ_DONE;
175             end
176             WRITE1_RQ_DONE: begin
177                 if(req_done_from_mi == 1) begin
178                     write_data_req_to_mi <= 0;
179                     state <= WRITE1_UPDATE_ADDR;
180
181                     if(write1_size != 0)begin
182                         force_req_to_mi <= 1;
183                     end
184                 end
```

```
185             end
186
187
188             //############################ READ1 ##########################
189             READ1_UPDATE_ADDR: begin
190                 if(read1_size == 0)begin
191                     state <= IDLE1;
192                 end
193                 else begin
194                     force_req_to_mi <= 0;
195                     read1_size  <= read1_size-1;
196                     addr_to_mi  <= {STARTBIT, READ1_SLAVE, READ, read1_addr};
197                     read1_addr  <= read1_addr + 1;
198                     state <= READ1_ADDR_RQ;
199                 end
200             end
201
202             READ1_ADDR_RQ: begin
203                 write_addr_req_to_mi <= 1;
204                 state <= READ1_OK_RES;
205             end
206
207             READ1_OK_RES: begin
208                 if(ok_response_from_mi == 1)begin
209                     write_addr_req_to_mi <= 0;
210                     state <= READ1_DATA_RQ;
211                 end
212             end
213
214             READ1_DATA_RQ: begin
215                 read_data_req_to_mi <= 1;
216                 state <= READ1_RQ_DONE;
217             end
218
219             READ1_RQ_DONE: begin
220                 if(req_done_from_mi == 1) begin
221                     read_data_req_to_mi <= 0;
222                     read1_data[read1_size] <= read_data_from_mi;
223                     state <= READ1_UPDATE_ADDR;
224
225                     if(read1_size != 0)begin
226                         force_req_to_mi <= 1;
227                     end
228                 end
229             end
230
231
232             //############################ IDLE1 ##########################
233             IDLE1: begin
234                 if(idle1_cycles == 0) begin
235                     state <= WRITE2_UPDATE_ADDR;
236                 end
237                 idle1_cycles <= idle1_cycles -1;
238             end
239
240
241             //############################ WRITE2 ##########################
242             WRITE2_UPDATE_ADDR: begin
243                 if(write2_size == 0)begin
244                     state <= ALL_DONE;
245                 end
246                 else begin
247                     force_req_to_mi <= 0;
```

```
248                     write2_size <= write2_size-1;
249                     addr_to_mi <= {STARTBIT, WRITE2_SLAVE, WRITE, write2_addr};
250                     write2_addr <= write2_addr + 1;
251                     state <= WRITE2_ADDR_RQ;
252                 end
253             end
254 
255             WRITE2_ADDR_RQ: begin
256                 write_addr_req_to_mi <= 1;
257                 state <= WRITE2_OK_RES;
258             end
259 
260             WRITE2_OK_RES: begin
261                 if(ok_response_from_mi == 1)begin
262                     write_addr_req_to_mi <= 0;
263                     state <= WRITE2_UPDATE_DATA;
264                 end
265             end
266 
267             WRITE2_UPDATE_DATA: begin
268                 write_data_to_mi <= write2_data;
269                 write2_data <= write2_data + WRITE2_DATA_INCR;
270                 state <= WRITE2_DATA_RQ;
271             end
272 
273             WRITE2_DATA_RQ: begin
274                 write_data_req_to_mi <= 1;
275                 state <= WRITE2_RQ_DONE;
276             end
277             WRITE2_RQ_DONE: begin
278                 if(req_done_from_mi == 1) begin
279                     write_data_req_to_mi <= 0;
280                     state <= WRITE2_UPDATE_ADDR;
281 
282                     if(write2_size != 0)begin
283                         force_req_to_mi <= 1;
284                     end
285                 end
286             end
287 
288 
289             //############################## END ##########################
290             ALL_DONE: begin
291                 state <= ALL_DONE;
292             end
293 
294         endcase
295     end
296 
297 end
298 
299 endmodule
```

9.7 APPENDIX G

Master interface module (master_interface.v)

```
1   module master_interface(clk,
2                   reset,
3                   addr,
4                   wdata,
5                   rdata,
6                   response,
7                   bus_req,
8                   grant,
9                   util,
10
11                  addr_from_master,
12                  write_addr_req_from_master,
13                  notify_ok_response_to_master,
14                  write_data_from_master,
15                  read_data_to_master,
16                  write_data_req_from_master,
17                  read_data_req_from_master,
18                  req_done_to_master,
19                  force_req_from_master,
20
21                  addr_en,
22                  wdata_en
23  );
24  //INPUTS &OUTPUTS:
25  // from and to the design
26  input clk;              // clock signal
27  input reset;            // reset signal
28  output addr;            //address bus
29  output wdata;           // data write bus
30  input rdata;            // data read bus
31  input [1:0] response;   // response bus from slave
32  output reg bus_req = 0; // request to arbiter
33  input grant;            // grant permission from arbiter
34  output reg util = 0;    // utilized flag of master
35  output reg addr_en = 0; // enable tristate buffer to addr bus
36  output reg wdata_en= 0; // enable tristate buffer to wdata bus
37
38  //from master module
39  input [15:0] addr_from_master; //address to write to addr bus
40  input write_addr_req_from_master;// address write request
41  output reg notify_ok_response_to_master = 0;
42  input [7:0] write_data_from_master;     // data to WDATA bus
43  output reg [7:0] read_data_to_master =0;// data from RDATA bus
44  input write_data_req_from_master;       // data write command
45  input read_data_req_from_master;        // data read command
46  output reg req_done_to_master = 0;
47  input force_req_from_master;
48
49  // states of state machine
50  parameter  IDLE         = 4'b0000;
51  parameter REQUEST_BUS   = 4'b0001;
52  parameter SENDADDRESS   = 4'b0010;
53  parameter READDATA      = 4'b0011;
54  parameter WRITEDATA     = 4'b0100;
55  parameter SPLIT         = 4'b0101;
56  parameter WAIT_FOR_SPLIT= 4'b0110;
57  parameter CHECK_NEXT    = 4'b0111;
58  parameter CHECK_NEXT2   = 4'b1000;
```

```
59
60  //responses from slave
61  parameter OK = 2'b10, BUSY = 2'b01, DONE = 2'b11,  NCK = 2'b00;
62
63  //internal registers
64  reg [3:0] state              = IDLE;
65  reg [1:0] burst_check_cycles = 2;
66  reg [15:0] serial_slave_address;
67  reg [7:0] serial_slave_data;
68
69  //send values to address and data buses
70  assign addr  = serial_slave_address[15];
71  assign wdata = serial_slave_data[7];
72
73  //STATE MACHINE:
74  always @(posedge clk or negedge reset)
75  begin
76      if(~reset) // reseting signals and registers
77      begin
78          state                <= IDLE;
79          bus_req              <= 0;
80          util                 <= 0;
81          addr_en              <= 0;
82          wdata_en             <= 0;
83          notify_ok_response_to_master <= 0;
84          read_data_to_master  <= 0;
85          req_done_to_master   <= 0;
86          serial_slave_address <= 0;
87          serial_slave_data    <= 0;
88          burst_check_cycles   <= 2;
89      end
90      else begin
91          case(state)
92              IDLE:begin // idle state
93                  util <= 0;
94                  addr_en <= 0;
95                  wdata_en <= 0;
96                  notify_ok_response_to_master <= 0;
97                  read_data_to_master  <= 0;
98                  serial_slave_address <= 0;
99                  serial_slave_data    <= 0;
100                 burst_check_cycles   <= 2;
101                 /*check for a request from master core to write an address*/
102                 if(write_addr_req_from_master) begin
103                     state <= REQUEST_BUS;//go to next state if req received
104                     req_done_to_master <= 1'b0;
105                 end
106             end
107
108             REQUEST_BUS:begin //requesting bus from arbiter
109                 bus_req <= 1'b1;
110                 serial_slave_address <= addr_from_master;
111                 notify_ok_response_to_master <= 1'b1;
112                 if(grant)begin //if granted, go to next state
113                     state <= SENDADDRESS;
114                     addr_en  <= 1;
115                     wdata_en <= 1;
116                     util     <= 1; //make utilized flag high
117                     bus_req  <= 0;
118                 end
119                 else begin    //else wait in the same state
120                     state <= REQUEST_BUS;
121                 end
```

```
122             end
123
124             SENDADDRESS:begin // sending address
125                 notify_ok_response_to_master <= 1'b0;
126
127                 //check for OK response from slave
128                 if(write_data_req_from_master && response == OK)
129                 begin
130                     state <= WRITEDATA; // go to write data state
131                     serial_slave_data <= write_data_from_master;
132                 end
133                 else if(read_data_req_from_master && response == OK) begin
134                     state <= READDATA;  // go to read data state
135                 end
136                 else if(response == BUSY) begin //if slave BUSY
137                     state    <= WAIT_FOR_SPLIT; //go to split state
138                     serial_slave_data <= write_data_from_master;
139                     util     <= 1'b0; //release bus utilization
140                     addr_en  <= 0;
141                     wdata_en <= 0;
142                 end
143                 else begin
144                     state <= SENDADDRESS;
145                      //send the address serially
146                     serial_slave_address[15:0] <= {serial_slave_address[14:0],1'b0};
147                 end
148             end
149
150             READDATA:begin  //read data state
151                 if(response == DONE) begin //if slave completed sending data
152                     state <= CHECK_NEXT;
153                     req_done_to_master <= 1;//notify master core about completion
154                 end
155                 else begin
156                     state <= READDATA;
157                     //read data from data bus serially
158                     read_data_to_master[7:0] <= {read_data_to_master[6:0],rdata};
159                 end
160             end
161
162             WRITEDATA:begin  //write data state
163                 if(response == DONE) begin //if slave responds completion
164                     state <= CHECK_NEXT;
165                     req_done_to_master <= 1;//notify master core about completion
166                 end
167                 else begin
168                     state <= WRITEDATA;
169                     //write data to data bus serially
170                     serial_slave_data[7:0] <= {serial_slave_data[6:0],1'b0};
171                 end
172             end
173
174             WAIT_FOR_SPLIT: begin //intermediate state to pass 1 clk cycle
175                 state <= SPLIT;
176             end
177
178
179             SPLIT:begin //split transaction state
180                 if(grant)begin //once grant is given back
181                     util <= 1'b1; //make the util line high
182                     addr_en  <= 1;
183                     wdata_en <= 1;
184                 end
```

```
185                 //if OK response received and having a WRITE req, go to write state
186                 if(write_data_req_from_master && response == OK && grant) begin
187                     state <= WRITEDATA;
188                 end
189                 //if OK response received and having a READ req, go to read state
190                 else if(read_data_req_from_master && response == OK && grant) begin
191                     state <= READDATA;
192                 end
193                 else begin
194                     state <= SPLIT;
195                 end
196             end
197 
198             CHECK_NEXT: begin
199                 state <= CHECK_NEXT2;
200                 if(force_req_from_master == 1)begin
201                     bus_req <= 1;
202                 end
203             end
204 
205             CHECK_NEXT2: begin
206                 state <= IDLE;
207                 if(force_req_from_master == 1)begin
208                     bus_req <= 1;
209                 end
210             end
211 
212         endcase
213     end
214 end
215 endmodule
```

9.8 APPENDIX H

Slave top module connecting slave interface with a 4K RAM module (**slave_top.v**)

```
1   module slave_top(
2 
3       input wire clock50,
4       input wire reset,
5       input wire address_bus,
6       input wire w_data_bus,
7       output tri r_data_bus,
8       output tri [1:0] response_bus,
9       output wire [1:0] split_request,
10      input wire [1:0] granted_master,
11      input wire [1:0] slave_address
12  );
13  //-----------WIRES-----------//
14  wire [7:0]q;
15  wire [11:0]ram_address;
16  wire [7:0]ram_data;
17  wire wr_en;
18  wire        sibufin_r_data_bus;
19  wire [1:0]  sibufin_response_bus;
20  wire        sibufin_en;
21 
22  //-----Trisate Buffers for RDATA,RESPONSE bus-----//
23  bufif1  buf_r_data_bus(r_data_bus, sibufin_r_data_bus, sibufin_en);
24  bufif1  buf_response_bus1(response_bus[1], sibufin_response_bus[1], sibufin_en);
25  bufif1  buf_response_bus0(response_bus[0], sibufin_response_bus[0], sibufin_en);
```

```
26
27  //----------SLAVE_INTERFACE INSTANTIATION----//
28  slave_interface SLAVE(
29      .clk                (clock50),
30      .reset              (reset),
31      .addr               (address_bus),
32      .w_data             (w_data_bus),
33      .r_data             (sibufin_r_data_bus),
34      .response           (sibufin_response_bus),
35      .split_request      (split_request),
36      .slave_address      (slave_address),
37      .granted_master     (granted_master),
38      .q                  (q),
39      .ram_address        (ram_address),
40      .ram_data           (ram_data),
41      .wr_en              (wr_en),
42      .slave_en           (sibufin_en)
43  );
44
45  //---------RAM_INSTANCE-------//
46  syncRAM RAM_BASIC(
47      .dataOut            (q),
48      .dataIn             (ram_data),
49      .Clk                (clock50),
50      .Addr               (ram_address),
51      .Wr_en              (wr_en),
52      .reset              (reset)
53  );
54  endmodule
```

9.9 APPENDIX I

Slave interface with address decoder (**slave_interface.v**)

```
1   module slave_interface(
2
3       input wire clk,     //system clock input
4       input wire reset,   //active low reset
5       input wire addr,    //address bus
6       input wire w_data,  //write data bus
7       output wire r_data, //read data bus
8       output [1:0] response,       //response bus
9       output [1:0] split_request,  //request split transaction
10      input  [1:0] granted_master, //indicates the active master on the bus
11      input wire [1:0]slave_address, //sets the device address of the slave
12      output reg slave_en, // enable signal for tri-state buffers
13
14      //------for ram----//
15      input wire [7:0]q,   // output data from the ram
16      output [11:0]ram_address, //address input of the ram
17      output [7:0]ram_data, // data input to the ram
18      output wr_en
19  );
20  //-----------PARAMETERS-------------//
21  //---STATES---//
22  parameter AWATING_GRANT         = 4'b0000;
23  parameter START_BIT             =4'b0001;
24  parameter GET_ADDR = 4'b0010;
25  parameter SEND_RESPONSE_ADDRESS = 4'b0011 ;
26  parameter GET_DATA = 4'b0100 ;
27  parameter SEND_DATA = 4'b0101 ;
```

```
28  parameter SEND_RESPONSE_DATA = 4'b0110 ;
29  parameter SPLIT =4'b0111;
30  parameter WAIT_BEFORE_GET_DATA =4'b1000;
31  //---RESPONSES---//
32  parameter RESP_NAK =2'b00;
33  parameter RESP_BUSY=2'b01;
34  parameter RESP_OK  =2'b10;
35  parameter RESP_DONE=2'b11;
36
37  //-----------REGISTERS--------------//
38  reg [3:0] state         = START_BIT;
39
40  reg [15:0]r_addr        =0;
41  reg [7:0]r_data_in      =0;
42  reg r_data_out          =0;
43  reg [1:0]r_response     =0;
44  reg [1:0]r_split        =0;
45  reg [4:0]addr_count     =0;
46  reg [3:0]data_count     =0;
47  reg wr_en_bit           =0;
48  reg [15:0]split_adress  =0;
49  reg [1:0]split_master   =0;
50  reg [1:0]r_split_request=0;
51  reg [3:0]busy_count     =5;
52
53
54  //-----------STATE_MACHINE----------//
55
56  always @ (posedge clk or negedge reset) begin
57      // This is a active low asynchronous reset.
58      // This will reset all the registers to their defaulst state.
59      if (~reset) begin
60          state <= START_BIT;
61          wr_en_bit<=0;
62          r_addr<=0;
63          r_data_in<=0;
64          r_data_out<=0;
65          r_response<=RESP_NAK;
66          r_split<=0;
67          addr_count<=0;
68          data_count<=0;
69          busy_count<=5;
70          slave_en <=0;
71          split_adress <= 0;
72          split_master <= 0;
73          r_split_request <= 0;
74      end else begin
75
76      case (state)
77      // This is the default state of the module
78      // This states ideentifies the start bit of the address
79          START_BIT: begin
80              slave_en <= 0;
81              if (addr==1) begin
82                  state <= GET_ADDR;
83                  addr_count<=14;
84                  wr_en_bit<=0;
85                  r_addr<=0;
86                  r_data_in<=0;
87                  r_data_out<=0;
88              end
89              else
90                  state <= START_BIT;
```

```
91
92              addr_count<=15;
93              r_response<=RESP_NAK;
94          end
95      //Capturs the 15-bit address which follows the start bit
96          GET_ADDR: begin
97
98              if (addr_count>1) begin
99                  r_addr[15:0] <= {r_addr[14:0],addr};
100                 addr_count <= addr_count -1;
101                 state <= GET_ADDR;
102             end else begin
103                 r_addr[15:0] <= {r_addr[14:0],addr};
104                 state <= SEND_RESPONSE_ADDRESS;
105             end
106             wr_en_bit<=0;
107             r_response<=RESP_NAK;
108         end
109
110     //decode the slave address and the type of transaction (mem r/w)
111     //and send the appropriate response to the master.
112         SEND_RESPONSE_ADDRESS: begin
113
114             if (r_addr[14:13] == slave_address) begin //check for slave address
115                 slave_en <= 1;
116
117                 if(busy_count == 0)begin
118                     if (r_addr[12]==1) begin // do this for mem write
119                         state<= WAIT_BEFORE_GET_DATA;
120                         r_response<=RESP_OK;
121                         data_count<=8;
122                     end
123                     else begin                // do this for mem read
124                         state<= SEND_DATA;
125                         r_response<=RESP_OK;
126                         data_count<=7;
127                     end
128                 end
129                 else begin
130                     r_response<=RESP_BUSY;  // send BUSY response
131                     split_adress<= r_addr;
132                     split_master<= granted_master;
133                     state <= SPLIT;
134                 end
135             end
136             else begin
137                 state<=START_BIT;
138             end
139         end
140
141     // capturs the 8bit data which is to be written to the ram
142         GET_DATA: begin
143            if (data_count>1) begin
144             r_data_in[7:0]<= {r_data_in[6:0],w_data};
145             data_count<=data_count-1;
146             state<=GET_DATA;
147            end else begin
148             r_data_in[7:0]<= {r_data_in[6:0],w_data};
149             state <= SEND_RESPONSE_DATA;
150             wr_en_bit<=1;
151            end
152            r_response<=RESP_NAK;
153         end
```

```
154
155     // send the data from the ram to the master
156         SEND_DATA: begin
157             if (data_count>0) begin
158             r_data_out<= q[data_count];
159             data_count<=data_count-1;
160             state<=SEND_DATA;
161            end else begin
162             r_data_out<= q[data_count];
163             state <= SEND_RESPONSE_DATA;
164             wr_en_bit<=0;
165             end
166             r_response<=RESP_NAK;
167         end
168     // send the DONE sesponse to the master which indicates
169     // a successfull data transfer.
170          SEND_RESPONSE_DATA: begin
171             wr_en_bit<=0;
172             r_response<=RESP_DONE;
173             state <=START_BIT;
174         end
175     // comes here after sending a BUSY response
176     // waits until the slave is ready.
177         SPLIT: begin
178             slave_en <= 0;
179             if (busy_count != 0) begin
180                 busy_count<=busy_count-1;
181             end
182             else begin
183                 r_split_request<=split_master;
184                 state <= AWATING_GRANT;
185             end
186         end
187
188     // after requesting a split transaction wait here until the
189     // arbiter grants permission
190         AWATING_GRANT: begin
191             if (granted_master==split_master) begin
192                 r_addr<=split_adress;
193                 state<= SEND_RESPONSE_ADDRESS;
194                 r_split_request<=0;
195             end else begin
196                 state <= AWATING_GRANT;
197             end
198         end
199     // wait one clock cycle before capture data
200         WAIT_BEFORE_GET_DATA: begin
201             state<=GET_DATA;
202         end
203     endcase // state
204
205     end
206
207 end
208 assign response = r_response;
209 assign ram_address = r_addr[11:0];
210 assign ram_data = r_data_in;
211 assign wr_en =wr_en_bit;
212 assign r_data=r_data_out;
213 assign split_request=r_split_request;
214 endmodule
```

9.10 APPENDIX J

RAM module connected to slave interface (**syncRAM.v**)

```
module syncRAM( dataIn,
                dataOut,
                Addr,
                Wr_en,
                Clk,
                reset
);


// parameters for the width
parameter ADR   = 12;
parameter DAT   = 8;
parameter DPTH  = 4096;

//ports
input      [DAT-1:0]  dataIn;
output reg [DAT-1:0]  dataOut;
input      [ADR-1:0]  Addr;
input                 Wr_en, Clk, reset;

//internal variables
integer i;
reg [DAT-1:0] SRAM [DPTH-1:0];

always @ (posedge Clk or negedge reset)
begin
  if(~reset)begin
      for (i=0; i<DPTH; i=i+1) SRAM[i] <= 0;
  end
  else begin
      if (Wr_en == 1'b1 ) begin
       SRAM [Addr] <= dataIn;
      end

      else  begin
       dataOut <= SRAM [Addr];
      end
  end
end

endmodule
```

9.11 APPENDIX K

Arbiter code (**arbiter.v**)

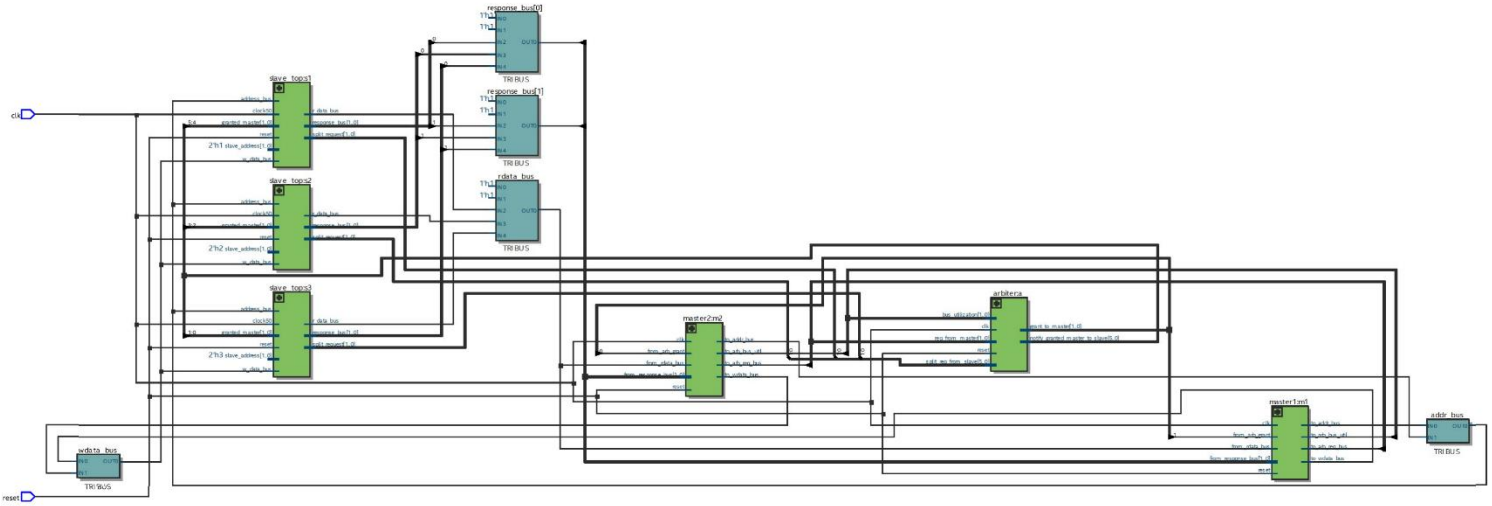
```
module arbiter (
    input clk,
    input reset,
    input [1:0]req_from_master,     //BUSREQ signals in format:[m1,m2]

    //SPLIT signals in format:[s1->m1,s1->m2,s2->m1,s2->m2,s3->m1,s3->m2]
    input [5:0]split_req_from_slave,

    input [1:0]bus_utilization,     //UTIL signals in format:[m1,m2]
    output reg [1:0]grant_to_master,//GRANT signals in format:[m1,m2]
```

```
11
12      //GMASTER signals in format:[s1<-m1,s1<-m2,s2<-m1,s2<-m2,s3<-m1,s3<-m2]
13      output reg [5:0]notify_granted_master_to_slave
14  );
15
16  wire [1:0]current_requests;//[m1 requested?, m2 requested ?]
17
18  /*either get responses from masters or get request from slaves waiting in
19  the split states to grant permission for corresponding master*/
20  assign current_requests[1] = req_from_master[1] || split_req_from_slave[5]
21                          || split_req_from_slave[3] || split_req_from_slave[1];
22  assign current_requests[0] = req_from_master[0] || split_req_from_slave[4]
23                          || split_req_from_slave[2] || split_req_from_slave[0];
24
25  // initially no master is granted. no master is notified to slave
26  initial begin
27      grant_to_master               <= 2'b00;
28      notify_granted_master_to_slave <= 6'b000000;
29  end
30
31  always @(posedge clk or negedge reset) begin
32      //asynchronous reset
33      if(~reset)begin
34          //reset to no grants and no granted notifications
35          grant_to_master               <= 2'b00;
36          notify_granted_master_to_slave <= 6'b000000;
37      end
38
39      //bus requests are checked and granted only when bus is not utilized
40      else if(bus_utilization == 2'b00) begin
41          case(current_requests) //process the requests from masters and slaves
42              2'b00:begin        //no requests
43                  grant_to_master                <= 2'b00;
44                  notify_granted_master_to_slave  <= 6'b000000;
45                  end
46              2'b10:begin        //request to grant master 1
47                  //grant master 1
48                  grant_to_master                <= 2'b10;
49                  //notify 3 slaves that master 1 is granted
50                  notify_granted_master_to_slave  <= 6'b101010;
51                  end
52              2'b01:begin        //request to grant master 2
53                  //grant master 2
54                  grant_to_master                <= 2'b01;
55                  //notify 3 slaves that master 2 is granted
56                  notify_granted_master_to_slave  <= 6'b010101;
57                  end
58              //high priority goes to master 1
59              2'b11:begin       //request to grant both master 1 and 2
60                  //master 1 is granted (high priority)
61                  grant_to_master                <= 2'b10;
62                  //notify 3 slaves that master 1 is granted
63                  notify_granted_master_to_slave  <= 6'b101010;
64                  end
65          endcase
66      end
67  end
68
69  endmodule
```

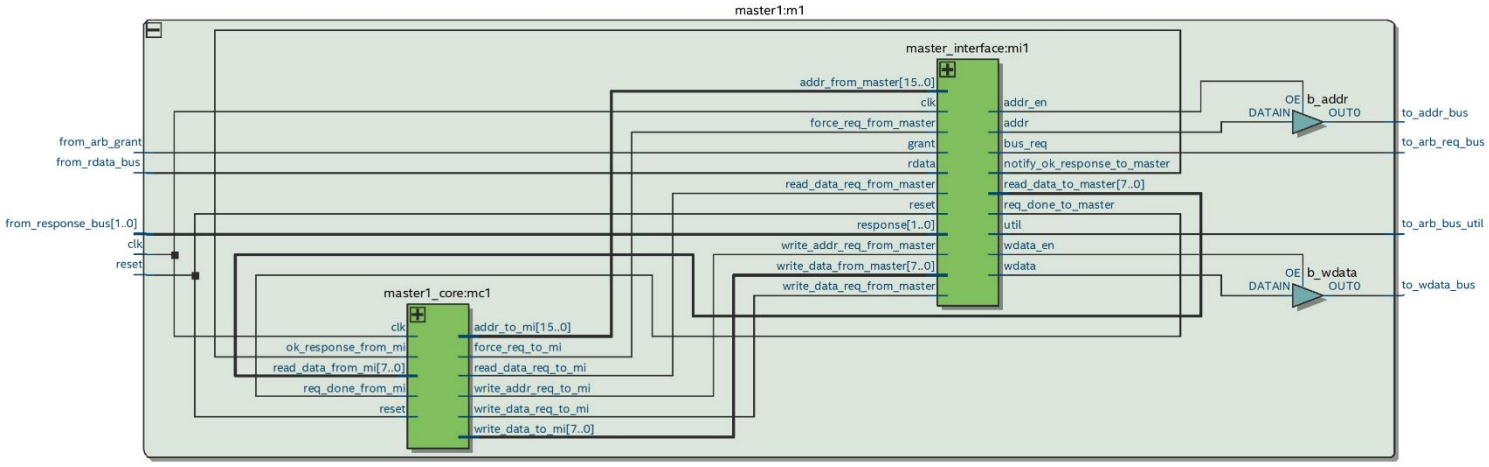
9.12 APPENDIX L

RTL View of the Design from Quartus

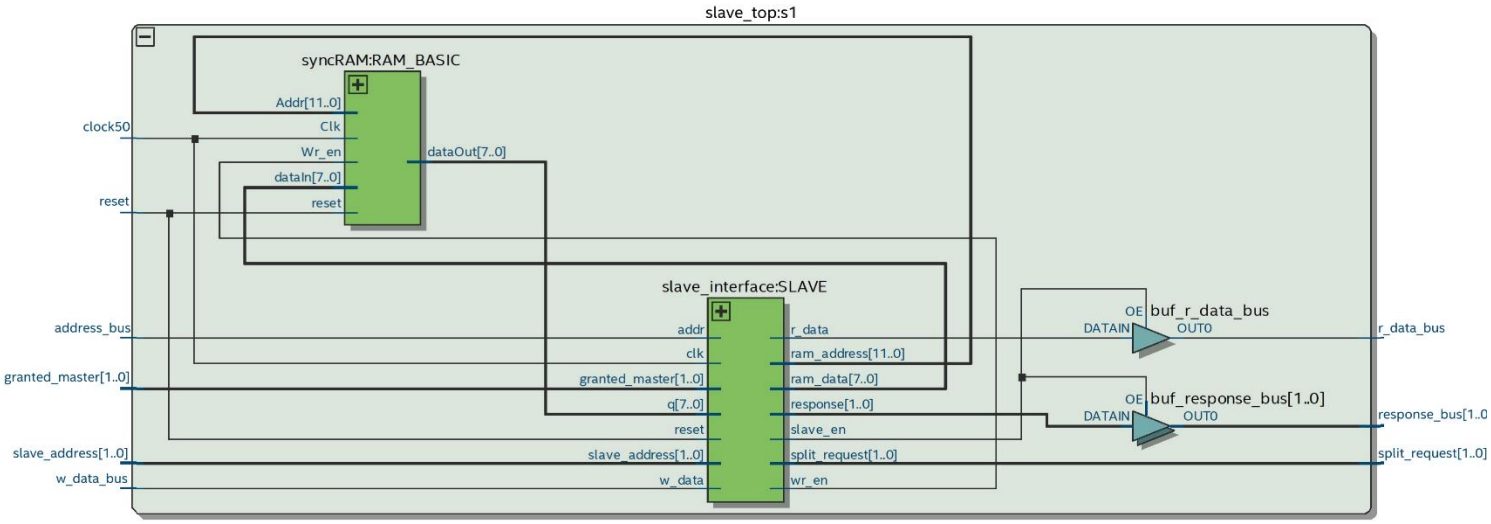
Top Level



Inside a Master



Inside a Slave



Inside Arbiter