Department of Electronic and Telecommunication Engineering
University of Moratuwa

EN4020: ADVANCED DIGITAL SYSTEMS

CUSTOM SERIAL BUS

With

Multi Master Handling

Bus Arbitration based on Priority

Split Transaction

J.K.S. Anjana 160027U
W.H.B.M. Himaruwan 160659H
S.J. Wanigasekara 160662K

Contents

Figures

Tables

1. Introduction

The report includes a detailed description of a functionality of priority based serial System Bus. The bus is capable of handling up to 4 Masters with 4 priority levels and 8 slave nodes. The design can be extended for higher no. of Masters and Slaves with minor modifications. Each Master can initialize its own priority level and if needed, priority levels can be change by the Master at any time. If two or more Masters have the same priority, the Master with highest numeric order will get the priority to access the bus. Split transaction is achieved by splitting the transaction in to 5 main pipeline stages.

A higher bus performance with low latency is achieved by designing two separate buses for Master to Slave communication and Salve to Master communication. This designing concept will help to increase the performance of pipeline architecture.

Arbiter design is achieved in two ways. If a Master is requesting something, the access will be given considering the priority level. But if a higher priority Master is requesting a task from a busy Slave, then the access will be given to next highest priority Master with a free Slave. A simple handshaking protocol is designed for the effective communication. The handshaking protocol that we followed is very much similar to AVALON Memory Map Handshaking Protocol.

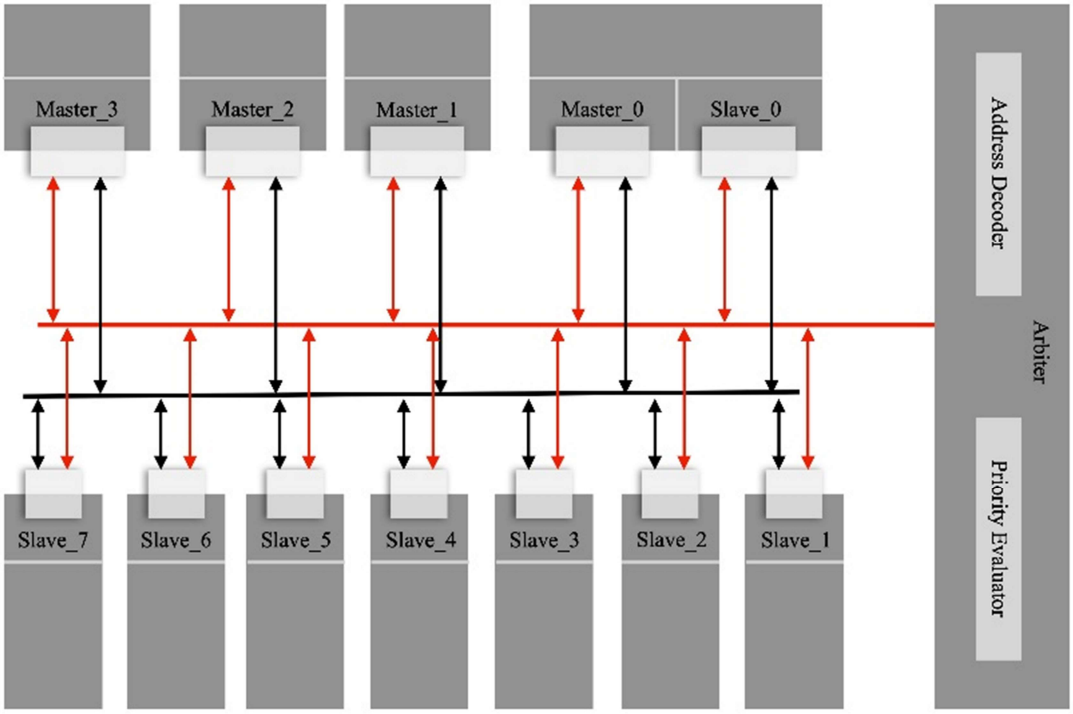


Figure 1.1 Highlevel Design Diagram

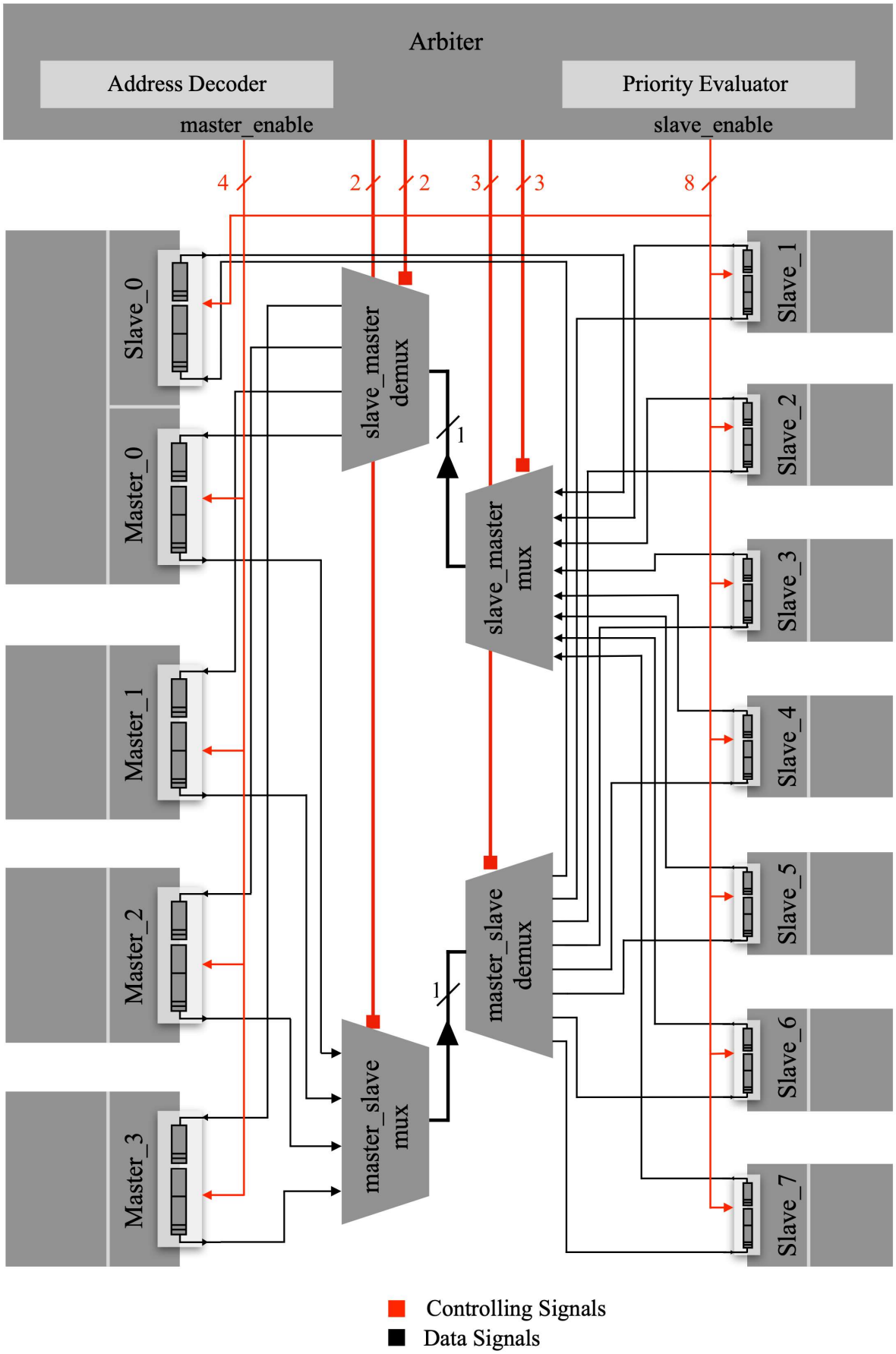


Figure 1.2 System Design with Data Paths

2. Master

Master plays a major role when setting up the communication to the Slave device through the System Bus. Master can either write data to a Slave or request data from a Slave. The design is capable of handling 4 priority levels for the Masters. Each Master can initialize its own priority level and if needed priority levels can be change by the Master at any time.

2.1 Master Interface

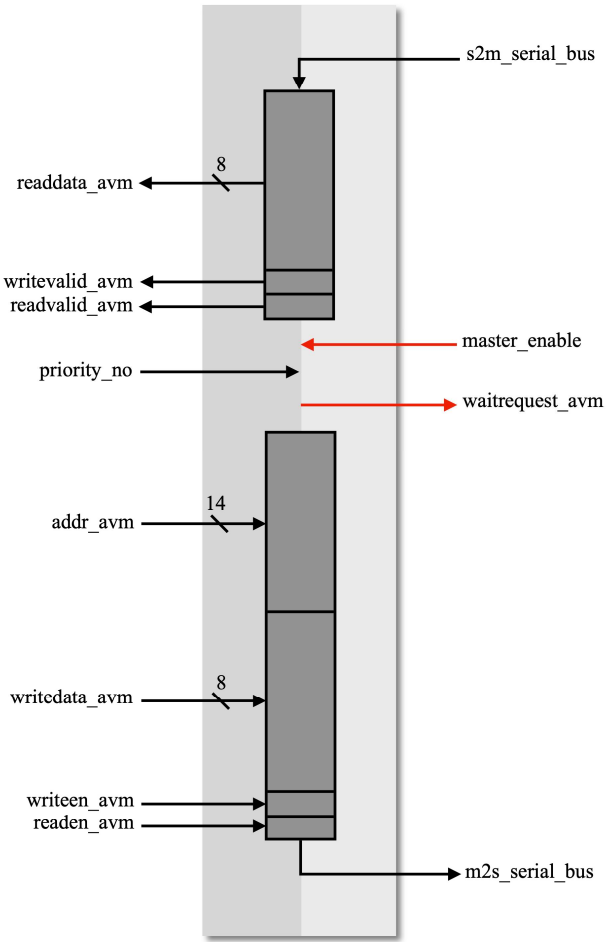


Figure 2.1 Bus Master Interface

Since data transmission through System Bus happens serially, it requires the parallel to serial conversion as well as serial to parallel conversion when transmitting the data to the Slave and receiving the data from the Slave, respectively.

Port	Width	Input/ Output	Description
readdata_avm	8	Output	Contains received data from Slave
readvalid_avm	1	Output	Indicates whether the read request is successful or not 1 - Successful 0 - Trouble / Master requested for Write
writevalid_avm	1	Output	Indicates whether the write request is successful or not 1 - Successful 0 - Trouble / Master requested for Read
addr_avm	14	Input	Contains the address where data to be written or read
writedata_avm	8	Input	Contains the data to be written
writeen_avm	1	Input	Indicates master is going to write the data to slave 1 - Write 0 - Read
readen_avm	1	Input	Indicates master is going to read the data from slave 1 - Read 0 - Write
priority_no	4	Input	Contains the priority level of the master 0001 - Lowest Priority 1000 - Highest Priority
m2s_serial_bus	-	Output	Serial output bus
s2m_serial_bus	-	Input	Serial input bus
master_enable	1	Input	Indicates whether the master is currently enable or not 1 - Enable / Busy 0 - Disable / Free
waitrequest_avm	1	Output	Indicates that the slave should keep its responds unchanged until waitrequest_avs goes low. (waitrequest_avm is driven by a master just after requesting read or write from a slave)

Table-1 Bus Master Interface I/O Signals

When writing the data , a segment of 24 bits which contains read enable signal (***readen_avm***), write enable signal (***written_avm***), data (***writedata_avm***/***readdata_avm***) and the relevant address (***addr_avm***) will be passed to Slave side through ***shift_reg_master*** considering the relevant priority conditions. This priority conditions will only take in to count when two or more Masters request the access at the same time. Then the Slave will send the signal ***writevalid_avs***, acknowledging the transaction is successful.

When requesting a data from Slave, a segment of 16 bits which contains read enable signal (***readen_avm***), write enable signal (***written_avm***), and relevant address (***addr_avm***) will be passed into Slave side. Then the Slave will send the relevant data back to the Master.

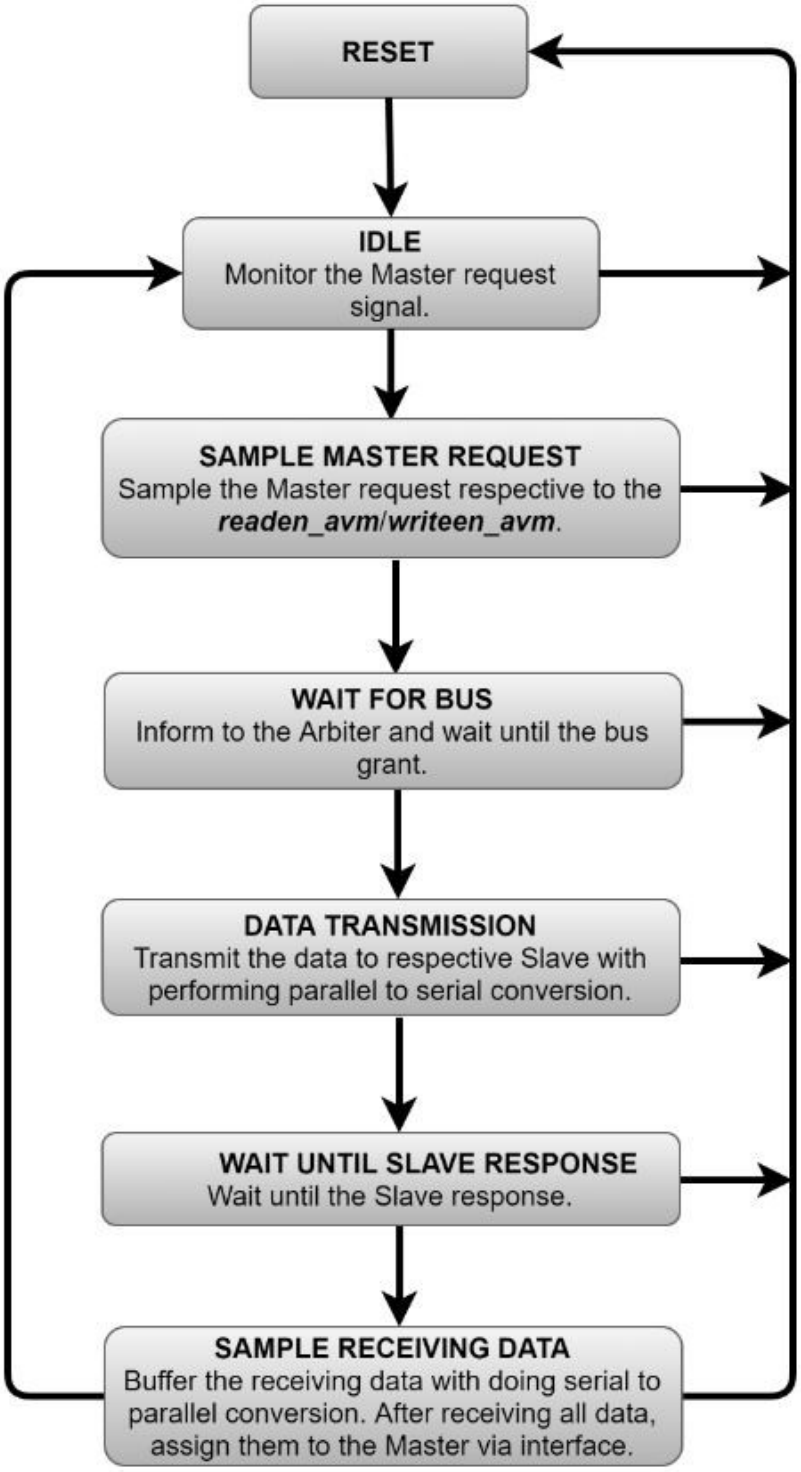


Figure 2.2 Master Interface State Diagram

3. Slave

The Slave is the controlling unit which will fulfill the Master's requests. The conversion of Global Address which provided by the Master, into a Local Address in a way that Slave can identify is done by the System Bus.

3.1 Slave Interface

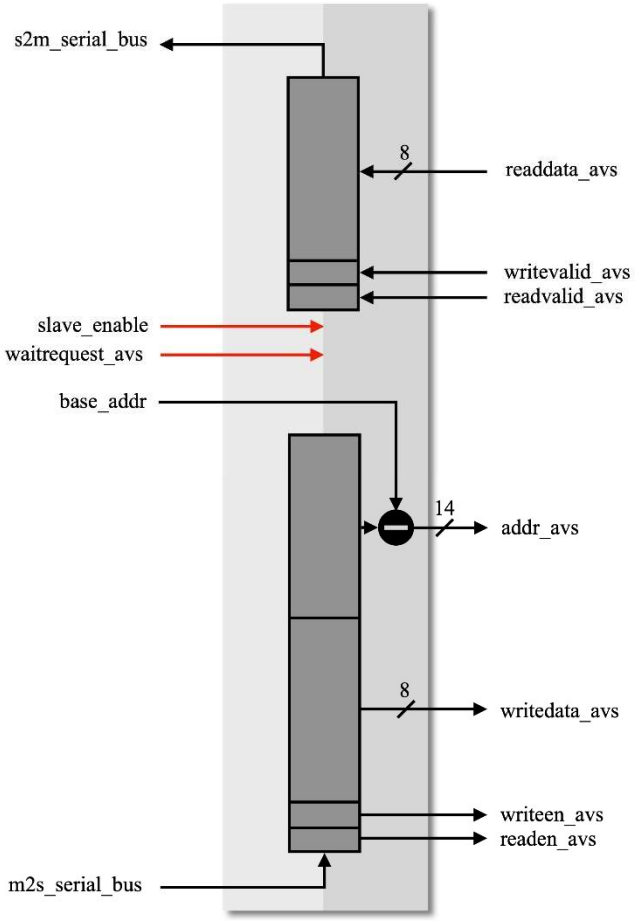


Figure 3.1 Bus Slave Interface

Global address to local address conversion is done at each Slave interface by subtracting the Base Address from Global Address. Shift registers are used to serial to parallel conversions and parallel to serial conversions when receiving and transmitting data. Slave interface will sample and store data, which is sent by the Slaves using ***readvalid_avs*** and ***writevalid_avs*** signals.

Port	Width	Input/ Output	Description
readdata_avs	8	Input	Contains read data from slave
writevalid_avs	1	Input	Indicates whether the write request is successful or not 1 - Successful 0 - Trouble / Master requested for Read
readvalid_avs	1	Input	Indicates whether the read request is successful or not 1 - Successful 0 - Trouble / Master requested for Read
addr_avs	14	Output	Contains the address where data to be written or read
writedata_avs	8	Output	Contains the data to be written
writeen_avs	1	Output	Indicates master is going to write the data to slave 1 - Write 0 - Read
readen_avs	1	Output	Indicates master is going to read the data from slave 1 - Read 0 - Write
s2m_serial_bus	-	Output	Serial output bus
m2s_serial_bus	-	Input	Serial input bus
slave_enable	1	Input	Indicates whether the slave is currently enable or not 1 - Enable / Busy 0 - Disable / Free
base_addr	14	Input	Contains the base address of the slave
waitrequest_avs	1	Input	Indicates that the slave should keep its responds unchanged until waitrequest_avs goes low.

Table-1 Bus Slave Interface I/O Signals

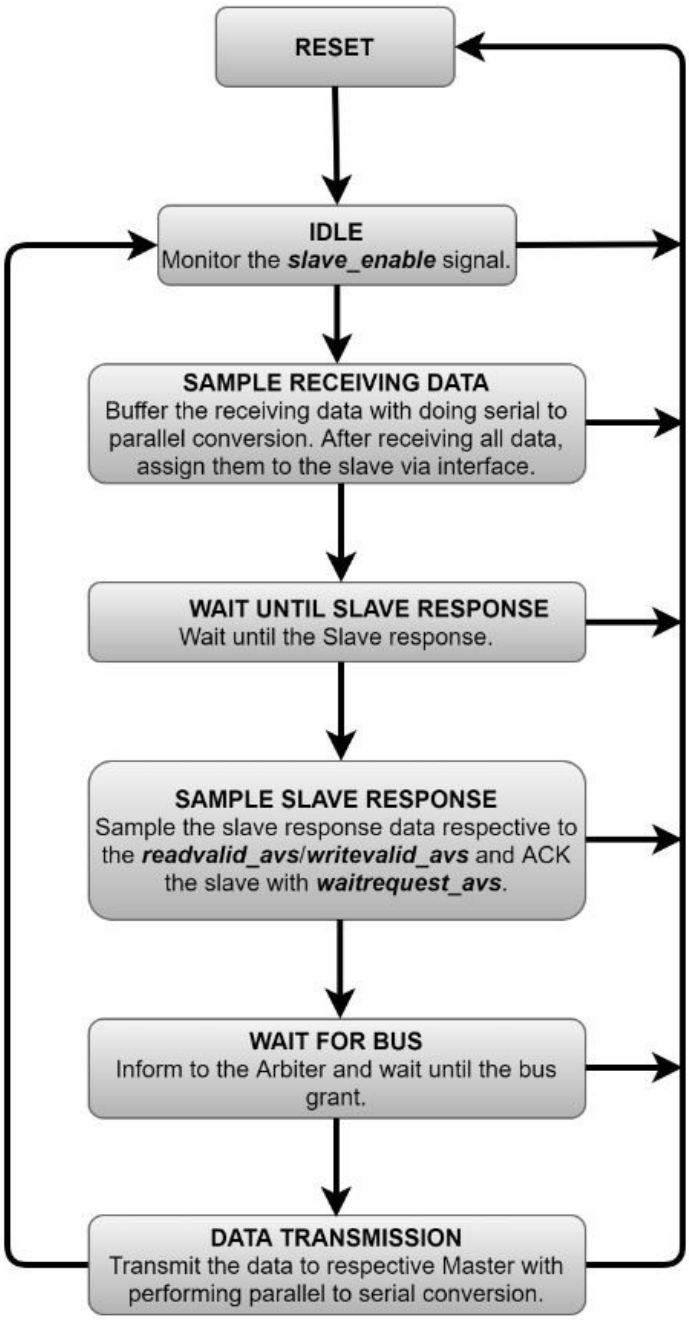


Figure 3.2 Slave Interface State Diagram

4. Bus Arbitration

Bus arbitration is done based on priority. The bus is capable of handling up to 4 Masters and 8 Slave nodes. The design is capable of handling 4 priority levels for the Masters. Each Master can initialize its own priority level and if needed priority levels can be change by the Master at any time. The design can be extended for higher no. of Masters and Slaves with minor modifications.

The priority levels are ***4'b1000, 4'b0100, 4'b0010, 4'b0001***. The System has designed in a way that Master with ***P1=4'b1000*** has the highest priority and Master with ***P4=4'b0001*** has the lowest priority. When a request comes from two or more Masters the selection will be done based on the priority level. If we have same priority level Masters, then the bus allocation will be done with the Master with highest numerical value.

Split transaction is achieved by splitting the transaction in to 5 main pipeline stages.

1. Master identification, selection (based on valid signals and priority) and identifying the corresponding Slaves.
    - Identify all the Masters who requested the bus and give the access of bus to the available most prioritized Master.
2. Writing data to corresponding Slave Interface. (Bus acquire)
    - Master can request bus either for writing data to a Slave or reading data from a Slave. The relevant controlling signals and data will be collected to send them serially through the Bus.
3. Identifying replied Slaves.
    - Catch the serial data and make them parallel at Slave's end so that interface of the Slave can feed data as a parallel stream to the Slave.
4. Writing the reply (Data/ACK) to corresponding Master. (Bus acquire)
    - Slave responds to the relevant Master according to its requirement. The relevant controlling signals and data will be collected to send them serially through the Bus.
5. Releaser
    - After completing the transaction, Master and Slave will be directed into a free state from their Busy state.

Arbiter also consists of Address decoder, which is the main controlling unit for Slave selection. The relevant Slave will be selected from comparing the targeted address with the base addresses and end addresses of each Slave.

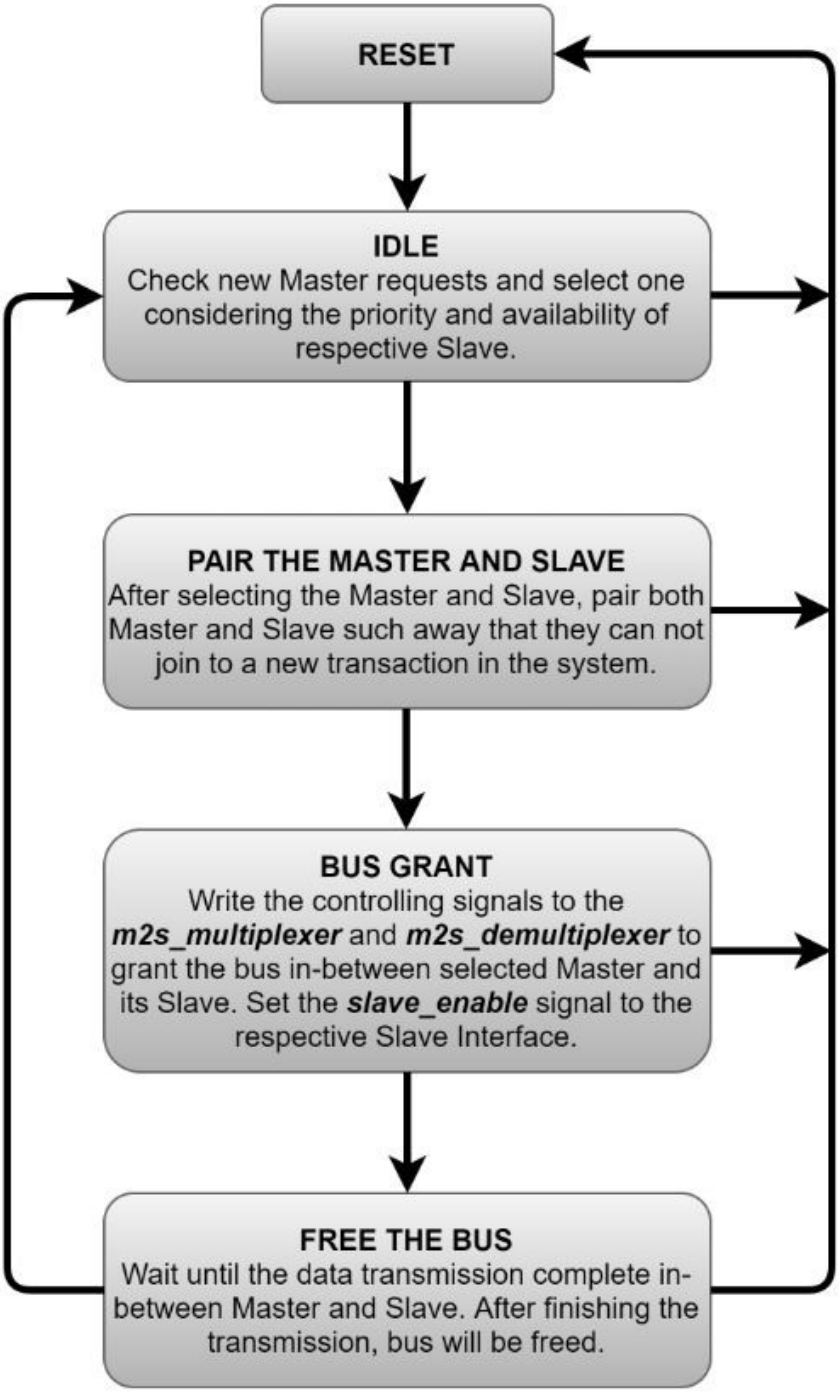


Figure 4.1 Master to Slave - Bus Arbitration State Diagram

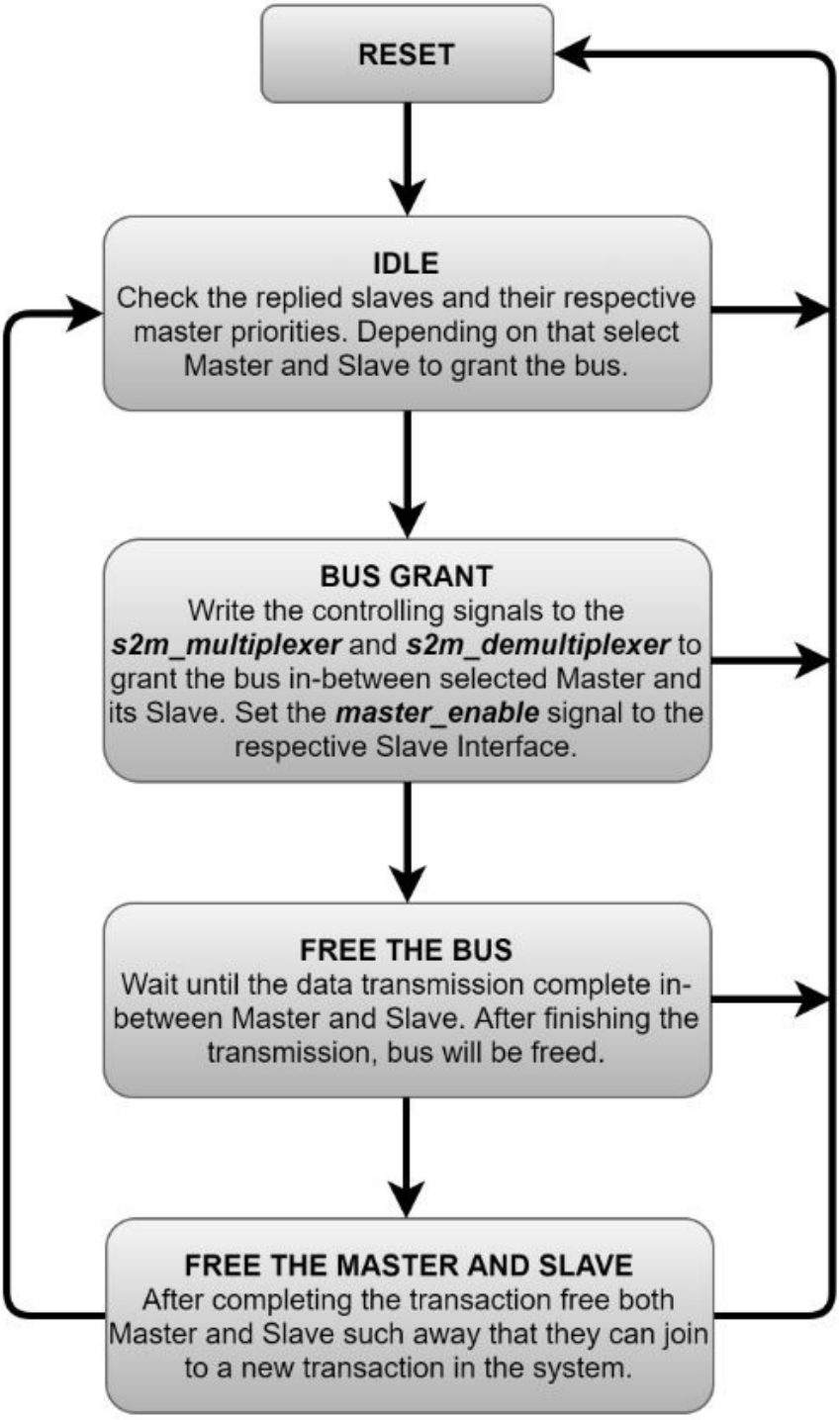


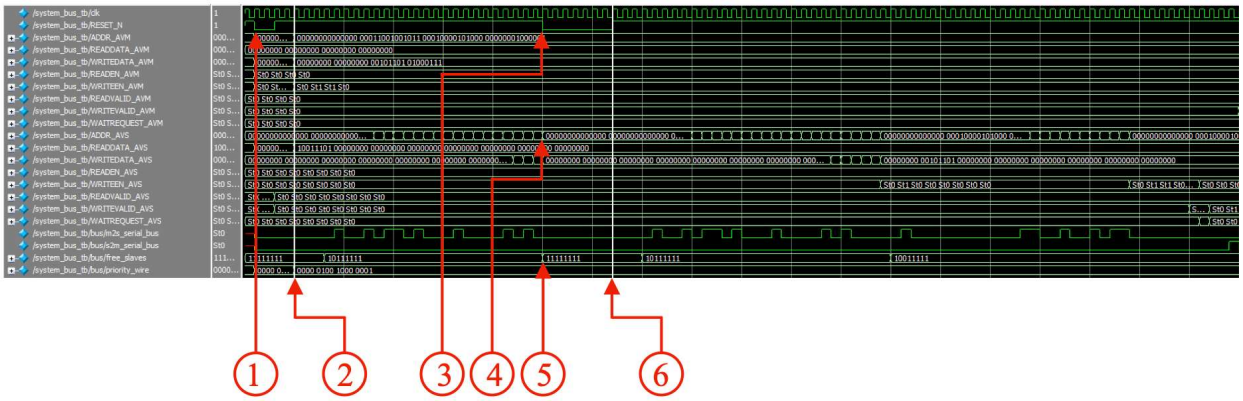
Figure 4.2 Slave to Master - Bus Arbitration State Diagram

5. Timing Diagrams

5.1 Reset Operations

Explanation:

Module are created with an asynchronous active low reset logic. When the ***RESET_N*** is pulled low, all the modules will go to its IDLE state and registers are assigned with their default values while all the incomplete Master requests will start from beginning.



1. Initial Asynchronous Reset.
2. Start Performing Master's requests from the beginning
3. Asynchronous Reset.
4. ***addr_avs*** set to zero.
5. All slaves become free.
6. Restart Performing Master's requests from the last incomplete requests.

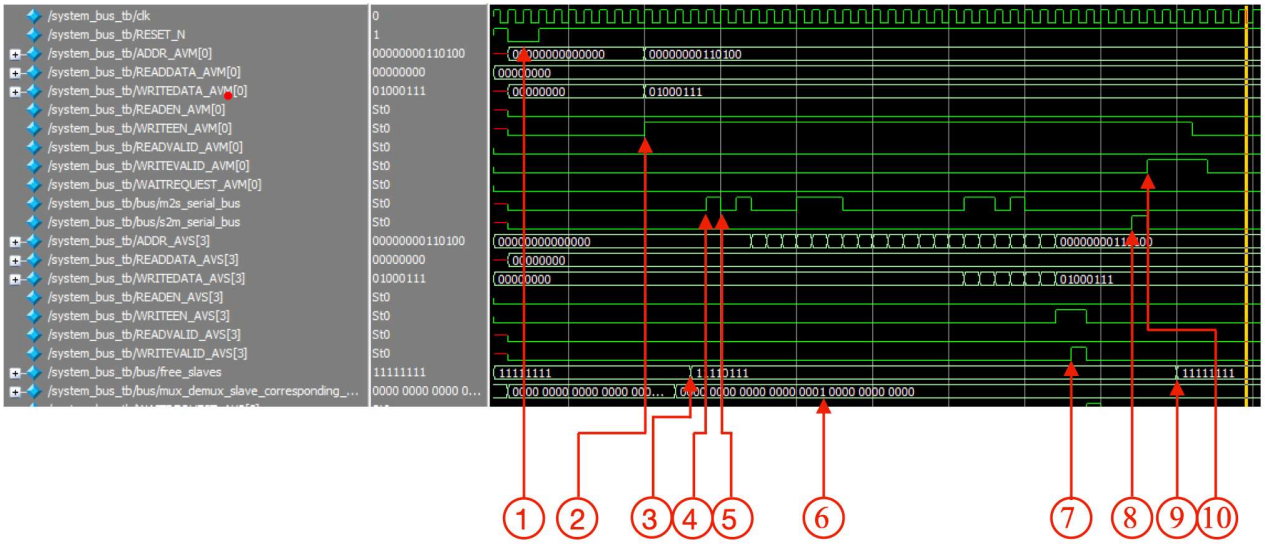
Figure 5.1 Timing Diagram for Reset option

5.2 One Master Request

Explanation:

This examples illustrate a situation where a Mater's request on a write transaction or a read transaction.

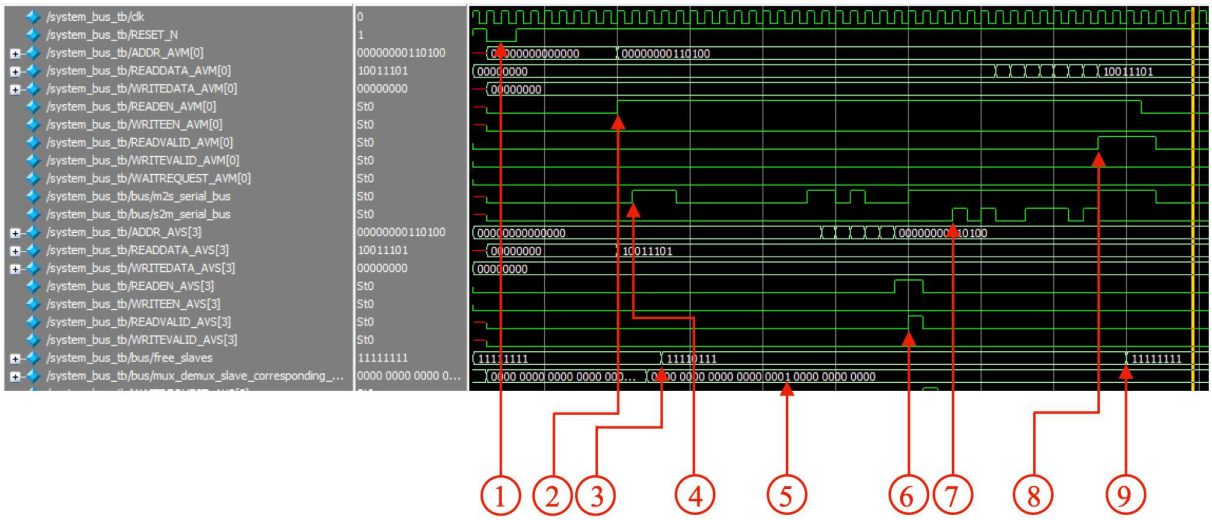
1. Write Transaction



1. Asynchronous Reset.
2. ***Master_0*** requests the bus for write transaction to the address space of ***Slave_3***.
3. Arbiter gives the access to the ***Master_0*** while making corresponding slave as a busy slave.
4. Master Interface maps the ***writeen_avm*** single onto ***m2s_serial_bus***.
5. Master Interface maps the ***addr_avm*** single onto ***m2s_serial_bus***.
6. Arbiter pairs ***Master_0*** with the ***Slave_3***.
7. Slave sends the ***writevalid_avs*** signal to the slave interface.
8. Slave interface maps ***writevalid_avs*** signal onto ***s2m_serial bus***.
9. Arbiter terminates the transaction by making the slave as a free slave.
10. Master interface samples ***s2m_serial bus*** and maps it onto ***writevalid_avm*** .

Figure 5.2 Timing Diagram for write Transaction

2. Read Transaction



1. Asynchronous Reset.
2. ***Master_0*** requests the bus for read transaction from the address space of ***Slave_3.***
3. Arbiter gives the access to the ***Master_0*** while making corresponding slave as a busy slave.
4. Master Interface maps the ***readen_avm*** single onto ***m2s_serial_bus***.
5. Arbiter pairs ***Master_0*** with the ***Slave_3***.
6. Slave sends the ***readvalid_avs*** signal to the slave interface.
7. Slave interface maps ***readvalid_avs*** signal onto ***s2m_serial bus.***
8. Master interface samples ***s2m_serial bus*** and maps it onto ***readvalid_avm.***
9. Arbiter terminates the transaction by making the slave as a free slave.

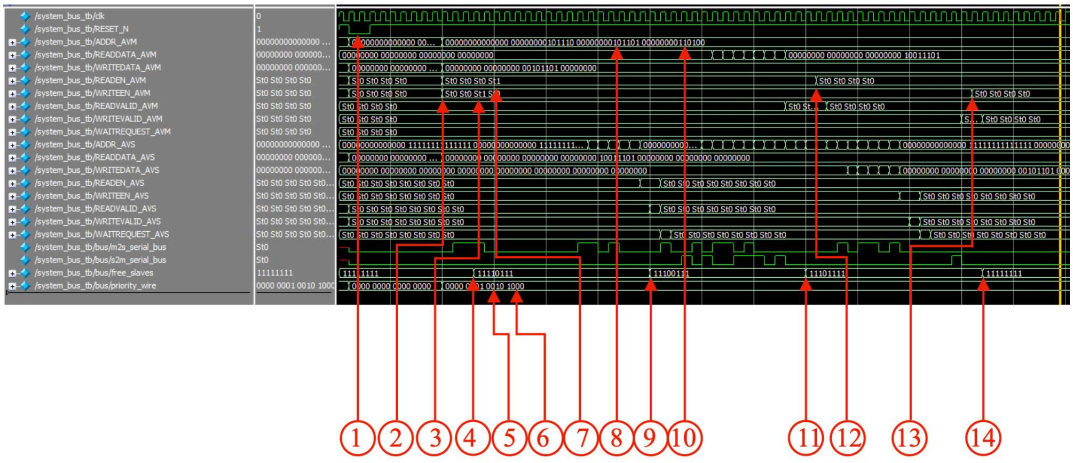
Figure 5.3 Timing Diagram for read Transaction

5.3 Priority levels

Explanation:

When we have 2 Masters with different priorities, the priority will be given to the Master with highest priority. But if that highest priority Master is requesting a busy Slave, then the priority will be given to the Master with next highest priority value requesting a free Slave. If we have same priority level Masters, then the bus allocation will be done with the Master with highest numerical value.

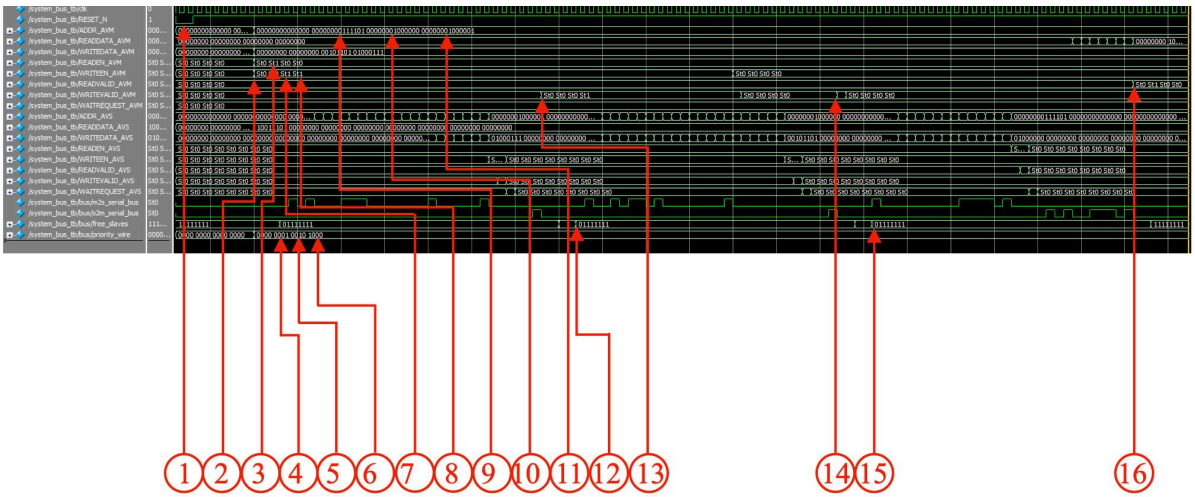
1. Two Masters with different priority.



1. Asynchronous Reset
2. Two masters request the bus at the same time.
3. ***Master_1*** requests the bus for write transaction.
4. Arbiter gives the access of m2s_serial_bus to the Master_0 first while making Slave_3 as a busy slave.
5. ***Master_1*** has priority level 2.
6. ***Master_0*** has priority level 4 (Highest Priority).
7. ***Master_0*** requests the bus for read transaction.
8. ***Master_1*** corresponds to the address space of ***Slave_4***.
9. Arbiter gives the access of ***m2s_serial_bus*** to the ***Master_1*** while making ***Slave_4*** as a busy slave.
10. ***Master_0*** corresponds to the address space of ***Slave_3***.
11. ***Master_0*** completes the read transaction while making ***Slave_3*** as a free slave.
12. Arbiter completes the read request of ***Master_0***.
13. Arbiter completes the write request of ***Master_1***.
14. ***Master_1*** completes the write transaction while making ***Slave_4*** as a free slave.

Figure 5.4 Timing Diagram for Two Masters with different priority.

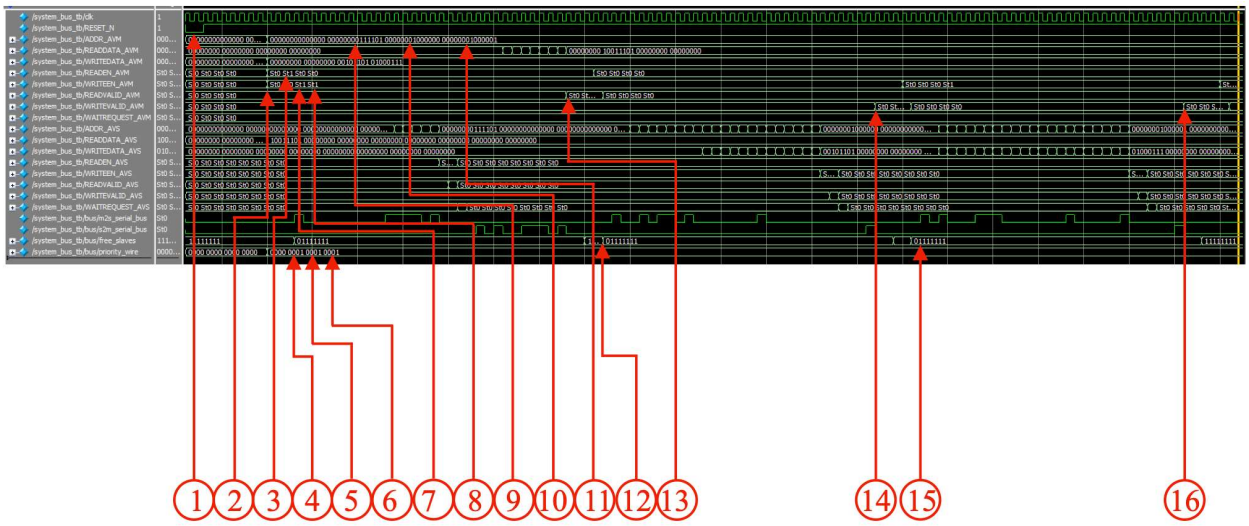
2. Three Masters with different priority but requesting from the same Slave.



1. Asynchronous Reset
2. Three masters request the bus at the same time.
3. ***Master_2*** request the bus for read transaction.
4. ***Master_2*** has the priority level 1.
5. ***Master_1*** has the priority level 2.
6. ***Master_0*** has the priority level 4.
7. ***Master_1*** request the bus for write transaction.
8. ***Master_0*** request the bus for write transaction.
9. ***Master_2*** corresponds to the address space of ***Slave_7***.
10. ***Master_1*** corresponds to the address space of ***Slave_7***.
11. ***Master_0*** corresponds to the address space of ***Slave_7***.
12. Arbiter gives the access of ***m2s_serial_bus*** to the ***Master_1*** while making ***Slave_7*** as a busy slave.
13. Arbiter completes the write request of ***Master_0***.
14. Arbiter gives the access of ***m2s_serial_bus*** to the ***Master_0*** while making ***Slave_7*** as a busy slave.
15. Arbiter completes the write request of ***Master_1***.
16. Arbiter completes the read request of ***Master_2***.

Figure 5.5 Timing Diagram for three Master scenario 1

3. Three Masters with Same priority



1. Asynchronous Reset
2. Three masters request the bus at the same time.
3. ***Master_2*** request the bus for read transaction.
4. ***Master_2*** has the priority level 1.
5. ***Master_1*** has the priority level 1.
6. ***Master_0*** has the priority level 1.
7. ***Master_1*** request the bus for write transaction.
8. ***Master_0*** request the bus for write transaction.
9. ***Master_2*** corresponds to the address space of ***Slave_7***.
10. ***Master_1*** corresponds to the address space of ***Slave_7***.
11. **Master_0** corresponds to the address space of ***Slave_7***.
12. Arbiter gives the access of ***m2s_serial_bus*** to the ***Master_1*** while making ***Slave_7*** as a busy slave.
13. Arbiter completes the write request of ***Master_2***.
14. Arbiter gives the access of ***m2s_serial_bus*** to the ***Master_0*** while making ***Slave_7*** as a busy slave.
15. Arbiter completes the write request of ***Master_1***.
16. Arbiter completes the read request of ***Master_0***.

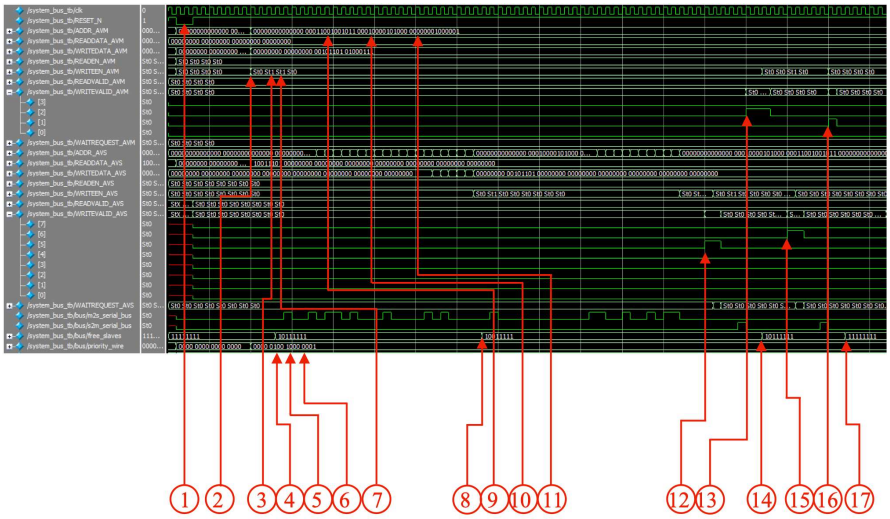
Figure 5.6 Timing Diagram for three Master scenario 2

5.4 Split Transaction

Explanation:

When the Slave responding for a Master's request, there may be some Slaves that will respond faster than other Slaves. If this split transaction is not there, when a higher priority Master requests a data from a slower Slave, the next priority Master must wait until the previous transaction complete. But with this split transaction, Bus will be dynamically allocated for the most suitable master without waiting to complete an entire transaction in between particular Master and Slave.

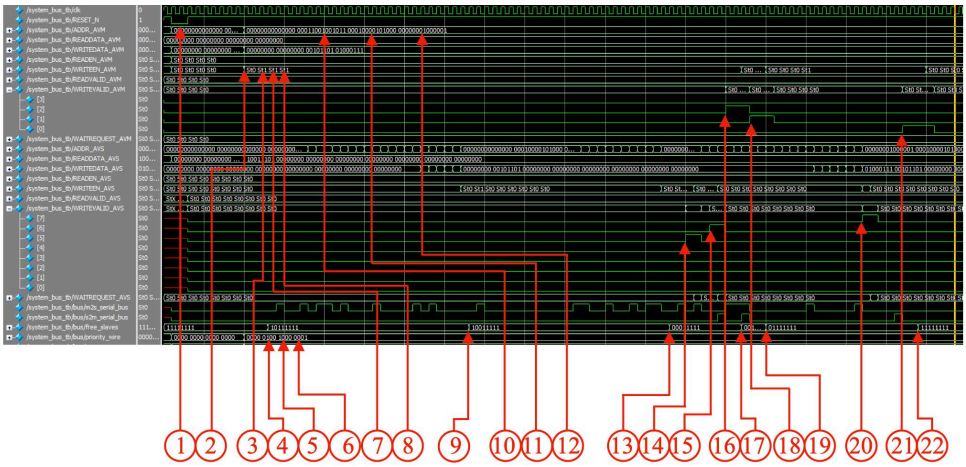
1. Higher priority Master requesting from a slower Slave and lower priority Master requesting from a faster Slave.(Two Master scenario)



1. Asynchronous Reset.
2. Two masters request the bus at the same time.
3. ***Master_2*** request the bus for write transaction.
4. ***Master_2*** has the priority level 3.
5. ***Master_1*** has the priority level 4.
6. ***Master_0*** has the priority level 1.
7. ***Master_1*** request the bus for write transaction.
8. Arbiter gives the access of ***m2s_serial_bus*** to the ***Master_2***.
9. ***Master_2*** corresponds to the address space of ***Slave_5***.
10. ***Master_1*** corresponds to the address space of ***Slave_6***.
11. ***Master_0*** corresponds to the address space of ***Slave_7***.
12. ***writevalid_avs*** of ***Slave_5*** gets high first.
13. ***writevalid_avs*** of ***Slave_6*** gets high finally.
14. Arbiter completes the write request of ***Master_2***.
15. ***writevalid_avm*** of ***Master_2*** gets high first.
16. ***writevalid_avm*** of ***Master_1*** gets high finally.
17. Arbiter completes the write request of ***Master_1***.

Figure 5.7 Timing Diagram foe split transaction Scenario 1

2. Higher priority Master requesting from a slower Slave and lower priority Master requesting from a faster Slave. (Three Master scenario)



1. Asynchronous Reset.
2. Three masters request the bus at the same time.
3. ***Master_2*** request the bus for write transaction.
4. ***Master_2*** has the priority level 3.
5. ***Master_1*** has the priority level 4.
6. ***Master_0*** has the priority level 1.
7. ***Master_1*** request the bus for write transaction.
8. ***Master_0*** request the bus for write transaction.
9. Arbiter gives the access of m2s_serial_bus to the ***Master_2***.
10. ***Master_2*** corresponds to the address space of ***Slave_5***.
11. ***Master_1*** corresponds to the address space of ***Slave_6***.
12. ***Master_0*** corresponds to the address space of ***Slave_7***.
13. Arbiter gives the access of ***m2s_serial_bus*** to the ***Master_0***.
14. ***writevalid_avs*** of ***Slave_5*** gets high first.
15. ***writevalid_avs*** of ***Slave_6*** gets high secondly.
16. ***writevalid_avm*** of ***Master_2 gets*** high first.
17. Arbiter completes the write request of ***Master_0***.
18. ***writevalid_avm*** of ***Master_1*** gets high secondly.
19. Arbiter completes the write request of ***Master_1***.
20. ***writevalid_avs*** of ***Slave_7*** gets high finally.
21. ***writevalid_avm*** of ***Master_0*** gets high finally.
22. Arbiter completes the write request of ***Master_2***.

Figure 5.8 Timing Diagram foe split transaction Scenario 2

6. Appendix

```
/* Advanced System Bus Design :
        Language - Verilog

        properties - Supports upto 4 Masters and 8 Slaves
                        5 stage piplined Arbiter to perform split transaction
                        Masters can hold 4 different priority stages and they allow to change their priority at any time.
                        Two aes are used in data transmision
                        master and slave interfaces are similar to the Avalon MM satndard (Avalon Memory Mapped - Altera)
                        Address width and Data width can be easily changed with parameters. (upgrading no of Masters and Slaves can be done with following same pattern used in case statements)

        Authors - Bhashitha Mishara, Sandaru Jayawardana, Shan Anjana

        License - MIT License
*/

module system_bus
    #(parameter DATA_WIDTH = 8,
    parameter ADDR_WIDTH = 14,
    parameter NO_OF_MASTER = 4,
    parameter NO_OF_SLAVE = 8,
    parameter [NO_OF_SLAVE*ADDR_WIDTH-1:0] SLAVE_BASE_ADDR = {14'd10, 14'd20, 14'd30, 14'd40, 14'd50, 14'd60, 14'd70, 14'd80},
    parameter [NO_OF_SLAVE*ADDR_WIDTH-1:0] SLAVE_END_ADDR = {14'd19, 14'd29, 14'd39, 14'd49, 14'd59, 14'd69, 14'd79, 14'd89}
    )

    (
    input clk,
    input reset_n,

    /// MASTER
    input [ADDR_WIDTH*NO_OF_MASTER-1:0]  addr_avm,
    output [DATA_WIDTH*NO_OF_MASTER-1:0]  readdata_avm,
    input [DATA_WIDTH*NO_OF_MASTER-1:0]  writedata_avm,
    input [NO_OF_MASTER-1:0]  readen_avm,
    input [NO_OF_MASTER-1:0]  writeen_avm,
    output [NO_OF_MASTER-1:0]  readvalid_avm,
    output [NO_OF_MASTER-1:0]  writevalid_avm,
    input [NO_OF_MASTER-1:0]  waitrequest_avm,

    // priority
    input [4*NO_OF_MASTER-1:0]priority_no,

    /// SALVE
    output [ADDR_WIDTH*NO_OF_SLAVE-1:0]  addr_avs,
    input [DATA_WIDTH*NO_OF_SLAVE-1:0]  readdata_avs,
    output [DATA_WIDTH*NO_OF_SLAVE-1:0]  writedata_avs,
    output [NO_OF_SLAVE-1:0]  readen_avs,
    output [NO_OF_SLAVE-1:0]  writeen_avs,
    input [NO_OF_SLAVE-1:0]  readvalid_avs,
    input [NO_OF_SLAVE-1:0]  writevalid_avs,
    output reg [NO_OF_SLAVE-1:0]  waitrequest_avs
    );


    reg [NO_OF_MASTER-1:0] readen_avm_last, writeen_avm_last, readen_avm_reg, writeen_avm_reg;
```

```
55.     reg [NO_OF_MASTER-
    1:0]  readvalid_avm_reg, readvalid_avm_reg_out, writevalid_avm_reg_out;
56.     reg [DATA_WIDTH-1:0]  readdata_avm_reg [NO_OF_MASTER-1:0];
57.     reg [7:0] readvalid_avs_last, writevalid_avs_last;
58.     reg [DATA_WIDTH-1:0]  readdata_avs_reg [NO_OF_SLAVE-1:0];
59.     reg [NO_OF_SLAVE:0] readvalid_avs_reg=9'b0;
60.     reg [NO_OF_SLAVE:0] writevalid_avs_reg=9'b0;
61.     reg [ADDR_WIDTH-1:0] addr_avs_reg [NO_OF_SLAVE-1:0];
62.     reg [DATA_WIDTH-1:0]  writedata_avs_reg [NO_OF_SLAVE-1:0];
63.     reg [NO_OF_SLAVE-
    1:0]  readen_avs_reg, writeen_avs_reg, readen_avs_reg_out, writeen_avs_reg_out;
64.
65.     wire [ADDR_WIDTH-1:0] addr_avm_wire [NO_OF_MASTER-1:0];
66.     wire [DATA_WIDTH-1:0] writedata_avm_wire [NO_OF_MASTER-1:0];
67.     wire [3:0] priority_wire [NO_OF_MASTER-1:0];
68.     wire [4*NO_OF_MASTER-1:0] priority_no_with_active;
69.
70.     reg [7:0] free_slaves = 8'hff;
71.     reg [3:0] last_state_stage_1_2;
72.     reg [1:0] master_slave_mux_master;
73.     reg [7:0] slave_master_demux_slave;
74.     reg [3:0] counter_s_p_master [NO_OF_MASTER-1:0];
75.     reg [3:0] value_s_p_master [NO_OF_MASTER-1:0];
76.     reg [7:0] slave_en_flag;
77.     reg [4:0] s_p_conversion_counter;
78.     reg [4:0] s_p_conversion_value;
79.     reg [3:0] s_p_conversion_counter_4;
80.     reg [3:0] s_p_conversion_value_4;
81.     reg stage_2_state;
82.     reg stage_4_state;
83.     reg [DATA_WIDTH + 2 - 1:0] shift_reg_slave;
84.     reg [DATA_WIDTH + ADDR_WIDTH + 2-1:0] shift_reg_master;
85.     reg [3:0] master_enable;
86.     reg [7:0] last_state_stage_3_4;
87.     reg [7:0] last_state_stage_3_4_single;
88.     reg [7:0] slave_active_stage_3;
89.     wire [31:0] slave_master_priority;
90.     wire [31:0] slave_master_priority_with_active;
91.     wire [3:0] master_enable_wire;
92.     wire m2s_serial_bus,s2m_serial_bus;
93.     wire [7:0] master_corresponding_slave [3:0];
94.     wire [3:0] master_active_stage_1;
95.
96.     reg [3:0] mux_demux_slave_corresponding_master [7:0] ;
97.     reg [1:0] mux_demux_slave_corresponding_master_2bit [7:0] ;
98.     reg [2:0] master_corresponding_slave_3bit [NO_OF_MASTER-1:0];
99.     reg [4:0] counter_s_p_slave [NO_OF_SLAVE-1:0];
100.        reg [4:0] value_s_p_slave [NO_OF_SLAVE-1:0];
101.        reg [1:0] state_master_interface [NO_OF_MASTER-1:0];
102.        reg [1:0] state_slave_interface [NO_OF_SLAVE-1:0];
103.        reg [7:0] slave_master_mux_slave;
104.        reg [3:0] slave_master_mux_slave_3b;
105.        reg [3:0] slave_master_mux_slave_3b_tem;
106.        reg [3:0] master_releaser_stage;
107.        reg [7:0] slave_releaser_stage;
108.
109.        assign master_active_stage_1=(readen_avm_reg | writeen_avm_reg) & (~last
    _state_stage_1_2) & (~{((master_corresponding_slave[3] & free_slaves) == 8'b0),((ma
    ster_corresponding_slave[2] & free_slaves) == 8'b0),((master_corresponding_slave[1]
     & free_slaves) == 8'b0),((master_corresponding_slave[0] & free_slaves) == 8'b0)});

110.        assign slave_master_priority={priority_wire[mux_demux_slave_correspondin
    g_master_2bit[7]],priority_wire[mux_demux_slave_corresponding_master_2bit[6]],prior
    ity_wire[mux_demux_slave_corresponding_master_2bit[5]],priority_wire[mux_demux_slav
    e_corresponding_master_2bit[4]],priority_wire[mux_demux_slave_corresponding_master_
    2bit[3]],priority_wire[mux_demux_slave_corresponding_master_2bit[2]],priority_wire[
```

```
    mux_demux_slave_corresponding_master_2bit[1]],priority_wire[mux_demux_slave_corresp
    onding_master_2bit[0]]};
111.            assign readen_avs = readen_avs_reg_out;
112.            assign writeen_avs = writeen_avs_reg_out;
113.            assign readvalid_avm=readvalid_avm_reg_out;
114.            assign writevalid_avm=writevalid_avm_reg_out;
115.            assign master_enable_wire=master_enable;
116.            assign m2s_serial_bus=shift_reg_master[DATA_WIDTH + ADDR_WIDTH + 1];
117.            assign s2m_serial_bus=shift_reg_slave[DATA_WIDTH + 1];
118.
119.            //************************** DATA PRE-
    PROCESSING, PARALLEL SERIAL CONVERSIONS AT END POINTS ************************//
120.
121.            genvar i, j, k;
122.
123.            generate
124.                for ( i=0;i<NO_OF_MASTER;i=i+1) // VECTORIZE_INTERFACE_SIGNALS
125.                    begin : VECTORIZE_INTERFACE_SIGNALS
126.                        assign priority_wire[i]=priority_no[i*4+3:i*4];
127.                        assign addr_avm_wire[i]=addr_avm[(i+1)*ADDR_WIDTH-
    1:i*ADDR_WIDTH];
128.                        assign writedata_avm_wire[i]=writedata_avm[(i+1)*DATA_WIDTH-
    1:i*DATA_WIDTH];
129.                    end
130.                for ( k=0;k<NO_OF_MASTER;k=k+1) // ADDRESS_DECODING_0
131.                    begin : ADDRESS_DECODING_0 // to identify the slave devices corr
    esponding to the global address and assign them to respective master
132.                        for ( i=0;i<NO_OF_SLAVE;i=i+1)
133.                            begin : ADDRESS_DECODING_1
134.                                assign master_corresponding_slave[k][i]=(addr_avm[AD
    DR_WIDTH*k-1+ADDR_WIDTH:ADDR_WIDTH*k] >= SLAVE_BASE_ADDR[(i+1)*ADDR_WIDTH-
    1:i*ADDR_WIDTH])? ((addr_avm[ADDR_WIDTH*k-
    1+ADDR_WIDTH:ADDR_WIDTH*k] <= SLAVE_END_ADDR[(i+1)*ADDR_WIDTH-
    1:i*ADDR_WIDTH])? 1'b1:1'b0):1'b0 ;
135.                            end
136.                    end
137.                for ( i=0;i<NO_OF_MASTER;i=i+1) // BIT_CONVERSION_8_3
138.                    begin : BIT_CONVERSION_8_3
139.                        always @(*)
140.                            begin
141.                                case(master_corresponding_slave[i])
142.                                    8'b00000001:
143.                                        begin
144.                                            master_corresponding_slave_3bit[i]=3'b00
    0;
145.                                        end
146.                                    8'b00000010:
147.                                        begin
148.                                            master_corresponding_slave_3bit[i]=3'b00
    1;
149.                                        end
150.                                    8'b00000100:
151.                                        begin
152.                                            master_corresponding_slave_3bit[i]=3'b01
    0;
153.                                        end
154.                                    8'b00001000:
155.                                        begin
156.                                            master_corresponding_slave_3bit[i]=3'b01
    1;
157.                                        end
158.                                    8'b00010000:
159.                                        begin
160.                                            master_corresponding_slave_3bit[i]=3'b10
    0;
161.                                        end
```

```
                                        8'b00100000:
                                            begin
                                                master_corresponding_slave_3bit[i]=3'b101;
                                            end
                                        8'b01000000:
                                            begin
                                                master_corresponding_slave_3bit[i]=3'b110;
                                            end
                                        8'b10000000:
                                            begin
                                                master_corresponding_slave_3bit[i]=3'b111;
                                            end
                                        default:
                                            begin
                                                master_corresponding_slave_3bit[i]=3'b000;
                                            end
                                    endcase
                                end
                            end

                        for ( i=0;i<NO_OF_MASTER;i=i+1) // IDENTIFY_MASTER_REQUEST_DECTING_WITH_EDGE
                            begin : IDENTIFY_MASTER_REQUEST_DECTING_WITH_EDGE
                                always @(posedge clk)
                                    begin
                                        case({((~readen_avm_last[i]) & readen_avm[i]), master_releaser_stage[i]})
                                            2'b10,2'b11:
                                                begin
                                                    readen_avm_reg[i]<=1'b1;
                                                end
                                            2'b01:
                                                begin
                                                    readen_avm_reg[i]<=1'b0;
                                                end
                                            2'b00:
                                                begin
                                                    readen_avm_reg[i]<=readen_avm_reg[i];
                                                end
                                            default:
                                                begin
                                                    readen_avm_reg[i]<=1'b0;
                                                end
                                        endcase

                                        case({((~writeen_avm_last[i]) & writeen_avm[i]), master_releaser_stage[i]})
                                            2'b10,2'b11:
                                                begin
                                                    writeen_avm_reg[i]<=1'b1;
                                                end
                                            2'b01:
                                                begin
                                                    writeen_avm_reg[i]<=1'b0;
                                                end
                                            2'b00:
                                                begin
                                                    writeen_avm_reg[i]<=writeen_avm_reg[i];

                                                end
                                            default:
                                                begin
```

```
                                            writeen_avm_reg[i]<=1'b0;
                                        end
                                endcase
                            end
                    end

                for ( i=0;i<NO_OF_SLAVE;i=i+1) // BIT_CONVERSION_4_2
                    begin : BIT_CONVERSION_4_2
                        always @(*)
                            begin
                                case(mux_demux_slave_corresponding_master[i])
                                    4'b0001:
                                        begin
                                            mux_demux_slave_corresponding_master_2bi
    t[i]=2'b00;
                                        end
                                    4'b0010:
                                        begin
                                            mux_demux_slave_corresponding_master_2bi
    t[i]=2'b01;
                                        end
                                    4'b0100:
                                        begin
                                            mux_demux_slave_corresponding_master_2bi
    t[i]=2'b10;
                                        end
                                    4'b1000:
                                        begin
                                            mux_demux_slave_corresponding_master_2bi
    t[i]=2'b11;
                                        end
                                    default:
                                        begin
                                            mux_demux_slave_corresponding_master_2bi
    t[i]=2'b00;
                                        end
                                endcase
                            end
                    end
                for ( i=0;i<NO_OF_MASTER;i=i+1) // DETECT_MASTER_REQUESTS_WITH_PRIOR
    ITY
                    begin : DETECT_MASTER_REQUESTS_WITH_PRIORITY
                        assign priority_no_with_active[i*4]=priority_no[i*4] & maste
    r_active_stage_1[i];
                        assign priority_no_with_active[i*4+1]=priority_no[i*4+1] & m
    aster_active_stage_1[i];
                        assign priority_no_with_active[i*4+2]=priority_no[i*4+2] & m
    aster_active_stage_1[i];
                        assign priority_no_with_active[i*4+3]=priority_no[i*4+3] & m
    aster_active_stage_1[i];
                    end

                for ( i=0;i<NO_OF_SLAVE;i=i+1) // DETECT_SLAVE_READY_WITH_PRIORITY
                    begin : DETECT_SLAVE_READY_WITH_PRIORITY
                        assign slave_master_priority_with_active[i*4]=slave_master_p
    riority[i*4] & slave_active_stage_3[i];
                        assign slave_master_priority_with_active[i*4+1]=slave_master
    _priority[i*4+1] & slave_active_stage_3[i];
                        assign slave_master_priority_with_active[i*4+2]=slave_master
    _priority[i*4+2] & slave_active_stage_3[i];
                        assign slave_master_priority_with_active[i*4+3]=slave_master
    _priority[i*4+3] & slave_active_stage_3[i];
                    end

                for (i=0;i<NO_OF_SLAVE;i=i+1) // SAMPLE_DATA_FROM_SLAVE_INTERFACE
                    begin : SAMPLE_DATA_FROM_SLAVE_INTERFACE
```

```
                        assign writedata_avs[(i+1)*DATA_WIDTH-
    1:i*DATA_WIDTH] = writedata_avs_reg[i];
                        assign addr_avs[(i+1)*ADDR_WIDTH-
    1:i*ADDR_WIDTH]=addr_avs_reg[i]-SLAVE_BASE_ADDR[i];
                        always @(posedge clk or negedge reset_n)
                            begin
                                if (~reset_n)
                                    begin
                                        readvalid_avs_reg[i]<=1'b0;
                                        writevalid_avs_reg[i]<=1'b0;
                                        readdata_avs_reg[i]<=8'b0;
                                        waitrequest_avs[i]<=1'b0;
                                    end
                                else
                                    begin
                                        case ({(readvalid_avs[i] & (~readvalid_avs_l
    ast[i])),(writevalid_avs[i] & (~writevalid_avs_last[i]))})
                                            2'b11:
                                                begin
                                                    readvalid_avs_reg[i]<=readvalid_
    avs[i];
                                                    writevalid_avs_reg[i]<=writevali
    d_avs[i];
                                                    readdata_avs_reg[i]<=readdata_av
    s[i*8+7:i*8];
                                                    waitrequest_avs[i]<=1'b1;
                                                end
                                            2'b10:
                                                begin
                                                    readvalid_avs_reg[i]<=readvalid_
    avs[i];
                                                    writevalid_avs_reg[i]<=writevali
    d_avs[i];
                                                    readdata_avs_reg[i]<=readdata_av
    s[i*8+7:i*8];
                                                    waitrequest_avs[i]<=1'b1;
                                                end
                                            2'b01:
                                                begin
                                                    readvalid_avs_reg[i]<=readvalid_
    avs[i];
                                                    writevalid_avs_reg[i]<=writevali
    d_avs[i];
                                                    readdata_avs_reg[i]<=readdata_av
    s_reg[i];
                                                    waitrequest_avs[i]<=1'b1;
                                                end
                                            2'b00:
                                                begin
                                                    readvalid_avs_reg[i]<=readvalid_
    avs_reg[i];
                                                    writevalid_avs_reg[i]<=writevali
    d_avs_reg[i];
                                                    readdata_avs_reg[i]<=readdata_av
    s_reg[i];
                                                    waitrequest_avs[i]<=1'b0;
                                                end
                                            default:
                                                begin
                                                    readvalid_avs_reg[i]<=1'b0;
                                                    writevalid_avs_reg[i]<=1'b0;
                                                    readdata_avs_reg[i]<=8'b0;
                                                    waitrequest_avs[i]<=1'b0;
                                                end
                                        endcase
                                    end
```

```
323.                             end
324.                     end
325.
326.                 for (j=0;j<NO_OF_SLAVE;j=j+1) // SERIAL_TO_PARALLEL_CONVERSION_AT_SL
    AVE_INTERFACES
327.                     begin : SERIAL_TO_PARALLEL_CONVERSION_AT_SLAVE_INTERFACES
328.                         always @(posedge clk or negedge reset_n)
329.                             begin
330.                                 if(~reset_n)
331.                                     begin
332.                                         counter_s_p_slave[j]<=5'b0;
333.                                         value_s_p_slave[j]<=5'b0;
334.                                         state_slave_interface[j]<=2'b0;
335.                                         writeen_avs_reg[j] <= 1'b0;
336.                                         readen_avs_reg[j] <= 1'b0;
337.                                         addr_avs_reg[j]<={ADDR_WIDTH{1'b0}};
338.                                         writedata_avs_reg[j]<=8'b0;
339.                                         writeen_avs_reg_out[j]<=1'b0;
340.                                         readen_avs_reg_out[j] <=1'b0;
341.                                     end
342.                                 else
343.                                     begin
344.                                         case(state_slave_interface[j])
345.                                             2'b0:
346.                                                 begin
347.                                                     writeen_avs_reg[j] <= writeen_av
    s_reg[j];
348.                                                     counter_s_p_slave[j]<=5'b0;
349.                                                     writedata_avs_reg[j]<=writedata_
    avs_reg[j];
350.                                                     addr_avs_reg[j]<=addr_avs_reg[j]
    ;
351.                                                     value_s_p_slave[j]<=value_s_p_sl
    ave[j];
352.                                                     if (slave_en_flag[j])
353.                                                         begin
354.                                                             state_slave_interface[j]
     <=2'b1;
355.                                                             readen_avs_reg[j] <= m2s
    _serial_bus;
356.                                                         end
357.                                                     else
358.                                                         begin
359.                                                             state_slave_interface[j]
     <=2'b0;
360.                                                             readen_avs_reg[j] <= 1'b
    0;
361.                                                         end
362.                                                     if (writeen_avs_reg_out[j] & wri
    tevalid_avs[j])
363.                                                         begin
364.                                                             writeen_avs_reg_out[j]<=
    1'b0;
365.                                                         end
366.                                                     else
367.                                                         begin
368.                                                             writeen_avs_reg_out[j]<=
    writeen_avs_reg_out[j];
369.                                                         end
370.                                                     if (readen_avs_reg_out[j] & read
    valid_avs[j])
371.                                                         begin
372.                                                             readen_avs_reg_out[j] <=
    1'b0;
373.                                                         end
374.                                                     else
```

```
375.                                                 begin
376.                                                     readen_avs_reg_out[j] <=
    readen_avs_reg_out[j];
377.                                                 end
378.                                             end
379.                                         2'b01:
380.                                             begin
381.                                                 writeen_avs_reg[j] <= m2s_serial
    _bus;
382.                                                 readen_avs_reg[j] <= readen_avs_
    reg[j];
383.                                                 counter_s_p_slave[j]<=5'b0;
384.                                                 state_slave_interface[j] <=2'b10
    ;
385.                                                 writedata_avs_reg[j]<=writedata_
    avs_reg[j];
386.                                                 addr_avs_reg[j]<=addr_avs_reg[j]
    ;
387.                                                 if(readen_avs_reg[j])
388.                                                     begin
389.                                                         value_s_p_slave[j]<=ADDR
    _WIDTH-1'b1;
390.                                                 end
391.                                                 else
392.                                                     begin
393.                                                         value_s_p_slave[j]<=ADDR
    _WIDTH+DATA_WIDTH-1'b1;
394.                                                 end
395.                                                 if (writeen_avs_reg_out[j] & wri
    tevalid_avs[j])
396.                                                     begin
397.                                                         writeen_avs_reg_out[j]<=
    1'b0;
398.                                                 end
399.                                                 else
400.                                                     begin
401.                                                         writeen_avs_reg_out[j]<=
    writeen_avs_reg_out[j];
402.                                                 end
403.                                                 if (readen_avs_reg_out[j] & read
    valid_avs[j])
404.                                                     begin
405.                                                         readen_avs_reg_out[j] <=
    1'b0;
406.                                                 end
407.                                                 else
408.                                                     begin
409.                                                         readen_avs_reg_out[j] <=
    readen_avs_reg_out[j];
410.                                                 end
411.                                             end
412.                                         2'b10:
413.                                             begin
414.                                                 writeen_avs_reg[j] <= writeen_av
    s_reg[j];
415.                                                 readen_avs_reg[j] <= readen_avs_
    reg[j];
416.                                                 value_s_p_slave[j]<=value_s_p_sl
    ave[j];
417.                                                 writedata_avs_reg[j]<={writedata
    _avs_reg[j],addr_avs_reg[j][ADDR_WIDTH-1]};
418.                                                 addr_avs_reg[j]<={addr_avs_reg[j
    ],m2s_serial_bus};
419.                                             if(value_s_p_slave[j] == counter_s_p
    _slave[j] )
420.                                                 begin
```

```
421.                                                    counter_s_p_slave[j]<=5'd0;
422.                                                    writeen_avs_reg_out[j]<=writ
    een_avs_reg[j];
423.                                                    readen_avs_reg_out[j] <= rea
    den_avs_reg[j];
424.                                                    state_slave_interface[j] <=2
    'b0;
425.                                                end
426.                                            else
427.                                                begin
428.                                                    counter_s_p_slave[j]<=counte
    r_s_p_slave[j]+5'd1;
429.                                                    writeen_avs_reg_out[j]<=writ
    een_avs_reg_out[j];
430.                                                    readen_avs_reg_out[j] <=read
    en_avs_reg_out[j];
431.                                                    state_slave_interface[j] <=2
    'b10;
432.                                                end
433.                                            end
434.
435.                                        default:
436.                                            begin
437.                                                counter_s_p_slave[j]<=5'b0;
438.                                                value_s_p_slave[j]<=5'b0;
439.                                                state_slave_interface[j]<=2'b0;
440.                                                writeen_avs_reg[j] <= 1'b0;
441.                                                readen_avs_reg[j] <= 1'b0;
442.                                                addr_avs_reg[j]<={ADDR_WIDTH{1'b
    0}};
443.                                                writedata_avs_reg[j]<=8'b0;
444.                                                writeen_avs_reg_out[j]<=1'b0;
445.                                                readen_avs_reg_out[j] <=1'b0;
446.                                            end
447.                                    endcase
448.                                end
449.                            end
450.                end
451.
452.                for (j=0;j<NO_OF_MASTER;j=j+1)
453.                    begin : SERIAL_TO_PARALLEL_CONVERSION_AT_MASTER_INTERFACES
454.                        assign readdata_avm[(j+1)*DATA_WIDTH-
    1:j*DATA_WIDTH]=readdata_avm_reg[j];
455.                        always @(posedge clk or negedge reset_n)
456.                            begin
457.                                if(~reset_n)
458.                                    begin
459.
460.                                        counter_s_p_master[j]<=4'b0;
461.                                        value_s_p_master[j]<=4'b0;
462.                                        state_master_interface[j]<=2'b0;
463.                                        readdata_avm_reg[j] <= 8'b0;
464.                                        readvalid_avm_reg[j] <= 1'b0;
465.                                        readvalid_avm_reg_out[j] <= 1'b0;
466.                                        writevalid_avm_reg_out[j] <= 1'b0;
467.
468.                                    end
469.                                else
470.                                    begin
471.                                        case(state_master_interface[j])
472.                                            2'b0:
473.                                                begin
474.                                                    counter_s_p_master[j]<=4'b0;
```

```
475.                                                 value_s_p_master[j]<=value_s_p_m
    aster[j];
476.                                                 readdata_avm_reg[j] <= readdata_
    avm_reg[j];
477.                                                 if (master_enable_wire[j])
478.                                                     begin
479.                                                         state_master_interface[j
    ] <=2'b1;
480.                                                     end
481.                                                 else
482.                                                     begin
483.                                                         state_master_interface[j
    ] <=2'b0;
484.                                                     end
485.                                                 casex({master_enable_wire[j],(re
    aden_avm[j] | writeen_avm[j] | waitrequest_avm[j])})
486.                                                     2'b1x:
487.                                                         begin
488.                                                             readvalid_avm_reg[j]
     <= s2m_serial_bus;
489.                                                             readvalid_avm_reg_ou
    t[j] <= 1'b0;
490.                                                             writevalid_avm_reg_o
    ut[j] <= 1'b0;
491.                                                         end
492.                                                     2'b01:
493.                                                         begin
494.                                                             readvalid_avm_reg[j]
     <= readvalid_avm_reg[j];
495.                                                             readvalid_avm_reg_ou
    t[j] <= readvalid_avm_reg_out[j];
496.                                                             writevalid_avm_reg_o
    ut[j] <= writevalid_avm_reg_out[j];
497.                                                         end
498.                                                     2'b00:
499.                                                         begin
500.                                                             readvalid_avm_reg[j]
     <= 1'b0;
501.                                                             readvalid_avm_reg_ou
    t[j] <= 1'b0;
502.                                                             writevalid_avm_reg_o
    ut[j] <= 1'b0;
503.                                                         end
504.                                                     default:
505.                                                         begin
506.                                                             readvalid_avm_reg[j]
     <= 1'b0;
507.                                                             readvalid_avm_reg_ou
    t[j] <= 1'b0;
508.                                                             writevalid_avm_reg_o
    ut[j] <= 1'b0;
509.                                                         end
510.                                                 endcase
511.                                             end
512.                                         2'b01:
513.                                             begin
514.                                                 counter_s_p_master[j]<=4'b0;
515.                                                 readvalid_avm_reg[j] <= readvali
    d_avm_reg[j];
516.
517.                                                 if(readvalid_avm_reg[j])
518.                                                     begin
519.                                                         value_s_p_master[j]<=DAT
    A_WIDTH-1'b1;
520.                                                         state_master_interface[j
    ] <=2'b10;
```

```
521.                                                     writevalid_avm_reg_out[j
    ] <= s2m_serial_bus;
522.                                                 end
523.                                             else
524.                                                 begin
525.                                                     value_s_p_master[j]<=4'd
    0;
526.                                                     state_master_interface[j
    ] <=2'b0;
527.                                                     writevalid_avm_reg_out[j
    ] <= s2m_serial_bus;
528.                                                 end
529.                                             readvalid_avm_reg_out[j] <= 1'b0
    ;
530.                                             readdata_avm_reg[j] <= readdata_
    avm_reg[j];
531.                                         end
532.                                     2'b10:
533.                                         begin
534.                                             value_s_p_master[j]<=value_s_p_m
    aster[j];
535.                                             readvalid_avm_reg[j] <= readvali
    d_avm_reg[j];
536.                                             readdata_avm_reg[j] <= {readdata
    _avm_reg[j],s2m_serial_bus};//shift_reg_slave[9]};
537.                                             writevalid_avm_reg_out[j] <= 1'b
    0;
538.                                             if(value_s_p_master[j] == counte
    r_s_p_master[j] )
539.                                                 begin
540.                                                     counter_s_p_master[j]<=4
    'd0;
541.                                                     readvalid_avm_reg_out[j]
     <= readvalid_avm_reg[j];
542.                                                     state_master_interface[j
    ] <=2'b0;
543.                                                 end
544.                                             else
545.
546.                                                 begin
547.                                                     counter_s_p_master[j]<=c
    ounter_s_p_master[j]+4'd1;
548.                                                     readvalid_avm_reg_out[j]
     <= 1'b0;
549.                                                     state_master_interface[j
    ] <=2'b10;
550.                                                 end
551.                                         end
552.                                     default:
553.                                         begin
554.                                             readdata_avm_reg[j] <= 8'b0;
555.                                             counter_s_p_master[j]<=4'b0;
556.                                             value_s_p_master[j]<=4'b0;
557.                                             state_master_interface[j]<=2'b0;
558.                                             readvalid_avm_reg[j] <= 1'b0;
559.                                             readvalid_avm_reg_out[j] <= 1'b0
    ;
560.                                             writevalid_avm_reg_out[j] <= 1'b
    0;
561.                                         end
562.                                 endcase
563.                             end
564.                         end
565.                 end
```

```
566.            endgenerate
567.
568.            always @(posedge clk or negedge reset_n)
569.                begin
570.                    if(~reset_n)
571.                        begin
572.                            readvalid_avs_last<=8'b0;
573.                            writevalid_avs_last<=8'b0;
574.                            readen_avm_last<=8'b0;
575.                            writeen_avm_last<=8'b0;
576.                        end
577.                    else
578.                        begin
579.                            readvalid_avs_last<=readvalid_avs;
580.                            writevalid_avs_last<=writevalid_avs;
581.                            readen_avm_last<=readen_avm;
582.                            writeen_avm_last<=writeen_avm;
583.                        end
584.                end
585.
586.            //************************** Pipeline stages of ARBITER ***************
    ************//
587.            /*
588.             1. Master identify and choose (based of valid signals and priority) and
     Identify the slave
589.             2. write data to corresponding slave (interface registers) (Bus acquire
    )
590.             3. Identify replied slaves.
591.             4. Write the reply to corresponding master (Data/ack) (Bus acquire)
592.             5. Releaser
593.            */
594.            //************************** PIPELINE STAGE 1: (SELECT MASTER PORT AND
    ASSIGN CONTROL SIGNALS TO MUX AND DEMUX) *************************//
595.
596.            always @(posedge clk or negedge reset_n)
597.                begin
598.                    if (~reset_n)
599.                        begin
600.                            master_slave_mux_master <=2'b0;
601.                            slave_master_demux_slave<=8'b0;
602.                            mux_demux_slave_corresponding_master[0] <= 4'b0;
603.                            mux_demux_slave_corresponding_master[1] <= 4'b0;
604.                            mux_demux_slave_corresponding_master[2] <= 4'b0;
605.                            mux_demux_slave_corresponding_master[3] <= 4'b0;
606.                            mux_demux_slave_corresponding_master[4] <= 4'b0;
607.                            mux_demux_slave_corresponding_master[5] <= 4'b0;
608.                            mux_demux_slave_corresponding_master[6] <= 4'b0;
609.                            mux_demux_slave_corresponding_master[7] <= 4'b0;
610.                        end
611.                    else
612.                        begin
613.                            casex({priority_no_with_active})
614.                                16'b1000xxxxxxxxxxx:
615.                                    begin
616.                                        master_slave_mux_master <=2'b11;
617.                                        slave_master_demux_slave<=master_correspondi
    ng_slave[3];
618.                                        mux_demux_slave_corresponding_master[master_
    corresponding_slave_3bit[3]] <= 4'b1000;//2'b11;
619.                                    end
620.                                16'b0xxx1000xxxxxxxx:
621.                                    begin
622.                                        master_slave_mux_master <=2'b10;
623.                                        slave_master_demux_slave<=master_correspondi
    ng_slave[2];
```

```
624.                                        mux_demux_slave_corresponding_master[master_
    corresponding_slave_3bit[2]] <= 4'b0100;
625.                                    end
626.                                16'b0xxx0xxx1000xxxx:
627.                                    begin
628.                                        master_slave_mux_master <=2'b01;
629.                                        slave_master_demux_slave<=master_correspondi
    ng_slave[1];
630.                                        mux_demux_slave_corresponding_master[master_
    corresponding_slave_3bit[1]] <= 4'b0010;
631.                                    end
632.                                16'b0xxx0xxx0xxx1000:
633.                                    begin
634.                                        master_slave_mux_master <=2'b0;
635.                                        slave_master_demux_slave<=master_correspondi
    ng_slave[0];
636.                                        mux_demux_slave_corresponding_master[master_
    corresponding_slave_3bit[0]] <= 4'b0001;
637.                                    end
638.                                16'b01000xxx0xxx0xxx:
639.                                    begin
640.                                        master_slave_mux_master <=2'b11;
641.                                        slave_master_demux_slave<=master_correspondi
    ng_slave[3];
642.                                        mux_demux_slave_corresponding_master[master_
    corresponding_slave_3bit[3]] <= 4'b1000;
643.                                    end
644.                                16'b00xx01000xxx0xxx:
645.                                    begin
646.                                        master_slave_mux_master <=2'b10;
647.                                        slave_master_demux_slave<=master_correspondi
    ng_slave[2];
648.                                        mux_demux_slave_corresponding_master[master_
    corresponding_slave_3bit[2]] <= 4'b0100;
649.                                    end
650.                                16'b00xx00xx01000xxx:
651.                                    begin
652.                                        master_slave_mux_master <=2'b01;
653.                                        slave_master_demux_slave<=master_correspondi
    ng_slave[1];
654.                                        mux_demux_slave_corresponding_master[master_
    corresponding_slave_3bit[1]] <= 4'b0010;
655.                                    end
656.
657.                                16'b00xx00xx00xx0100:
658.                                    begin
659.                                        master_slave_mux_master <=2'b0;
660.                                        slave_master_demux_slave<=master_correspondi
    ng_slave[0];
661.                                        mux_demux_slave_corresponding_master[master_
    corresponding_slave_3bit[0]] <= 4'b0001;
662.                                    end
663.                                16'b001000xx00xx00xx:
664.                                    begin
665.                                        master_slave_mux_master <=2'b11;
666.                                        slave_master_demux_slave<=master_correspondi
    ng_slave[3];
667.                                        mux_demux_slave_corresponding_master[master_
    corresponding_slave_3bit[3]] <= 4'b1000;
668.                                    end
669.                                16'b000x001000xx00xx:
670.                                    begin
671.                                        master_slave_mux_master <=2'b10;
672.                                        slave_master_demux_slave<=master_correspondi
    ng_slave[2];
```

```
673.                                        mux_demux_slave_corresponding_master[master_
    corresponding_slave_3bit[2]] <= 4'b0100;
674.                                    end
675.                                16'b000x000x001000xx:
676.                                    begin
677.                                        master_slave_mux_master <=2'b01;
678.                                        slave_master_demux_slave<=master_correspondi
    ng_slave[1];
679.                                        mux_demux_slave_corresponding_master[master_
    corresponding_slave_3bit[1]] <= 4'b0010;
680.                                    end
681.                                16'b000x000x000x0010:
682.                                    begin
683.                                        master_slave_mux_master <=2'b0;
684.                                        slave_master_demux_slave<=master_correspondi
    ng_slave[0];
685.                                        mux_demux_slave_corresponding_master[master_
    corresponding_slave_3bit[0]] <= 4'b0001;
686.                                    end
687.                                16'b0001000x000x000x:
688.                                    begin
689.                                        master_slave_mux_master <=2'b11;
690.                                        slave_master_demux_slave<=master_correspondi
    ng_slave[3];
691.                                        mux_demux_slave_corresponding_master[master_
    corresponding_slave_3bit[3]] <= 4'b1000;
692.                                    end
693.                                16'b00000001000x000x:
694.                                    begin
695.                                        master_slave_mux_master <=2'b10;
696.                                        slave_master_demux_slave<=master_correspondi
    ng_slave[2];
697.                                        mux_demux_slave_corresponding_master[master_
    corresponding_slave_3bit[2]] <= 4'b0100;
698.                                    end
699.                                16'b000000000001000x:
700.                                    begin
701.                                        master_slave_mux_master <=2'b01; // mux- 01,
     demux - 00000100
702.                                        slave_master_demux_slave<=master_correspondi
    ng_slave[1];
703.                                        mux_demux_slave_corresponding_master[master_
    corresponding_slave_3bit[1]] <= 4'b0010;
704.                                    end
705.                                16'b0000000000000001:
706.                                    begin
707.                                        master_slave_mux_master <=2'b0;
708.                                        slave_master_demux_slave<=master_correspondi
    ng_slave[0];
709.                                        mux_demux_slave_corresponding_master[master_
    corresponding_slave_3bit[0]] <= 4'b0001;
710.                                    end
711.                                default:
712.                                    begin
713.                                        master_slave_mux_master <=2'b0;
714.                                        slave_master_demux_slave<=8'b0;
715.                                    end
716.                            endcase
717.                        end
718.
719.                end
720.
721.            //*************************** PIPELINE STAGE 2: TRANSFER DATA TO SLAVE P
    ORTS WITH PARALLEL TO SERIAL DATA CONVERSION ************************//
722.
723.            always @(posedge clk or negedge reset_n)// stage 2
```

```
724.                begin
725.                    if(~reset_n)
726.                        begin
727.                            slave_en_flag <= 8'b0;
728.                            s_p_conversion_counter <= 5'b0;
729.                            s_p_conversion_value<=5'b0;
730.                            shift_reg_master<={(DATA_WIDTH+ADDR_WIDTH+2){1'b0}};
731.                            last_state_stage_1_2<=4'b0;
732.                            stage_2_state<=1'b0;
733.                            free_slaves<=8'hff;
734.                        end
735.                    else
736.                        begin
737.                            case(stage_2_state)
738.                                1'b0: // writedata_avm_wire
739.                                    begin
740.                                        s_p_conversion_counter <= 5'b0;
741.                                        if (writeen_avm[master_slave_mux_master])
742.                                            begin
743.                                                s_p_conversion_value <= DATA_WIDTH+A
    DDR_WIDTH+1'b1; //
744.                                                                                    // 0
                                    1                                   8'b0
                                // 45              // 01
745.                                                shift_reg_master <= {readen_avm[mast
    er_slave_mux_master],writeen_avm[master_slave_mux_master] ,writedata_avm_wire[maste
    r_slave_mux_master], addr_avm_wire[master_slave_mux_master]};
746.                                            end
747.                                        else
748.                                            begin
749.                                                s_p_conversion_value <= ADDR_WIDTH+1
    'b1;
750.                                                shift_reg_master <= {readen_avm[mast
    er_slave_mux_master],writeen_avm[master_slave_mux_master] , addr_avm_wire[master_sl
    ave_mux_master], {DATA_WIDTH{1'b0}}};
751.                                            end
752.
753.                                        if(slave_master_demux_slave!=8'b0) // 000001
    00
754.                                            begin
755.                                                slave_en_flag <= slave_master_demux_
    slave;
756.                                                last_state_stage_1_2 <= (last_state_
    stage_1_2 ^ master_releaser_stage) | (1'b1 << master_slave_mux_master);
757.                                                stage_2_state <= 1'b1;
758.                                                free_slaves <= ((free_slaves|slave_r
    eleaser_stage) ^ slave_master_demux_slave);
759.                                            end
760.                                        else
761.                                            begin
762.                                                slave_en_flag<=8'b0;
763.
764.                                                stage_2_state <= stage_2_state;
765.                                                last_state_stage_1_2 <= (last_state_
    stage_1_2 ^ master_releaser_stage);
766.                                                free_slaves <=(free_slaves|slave_rel
    easer_stage);
767.                                            end
768.                                    end
769.                                1'b1:
770.                                    begin
771.                                        s_p_conversion_value<=s_p_conversion_value;
    //17
772.                                        slave_en_flag<=8'b0;
773.                                        last_state_stage_1_2 <= (last_state_stage_1_
    2 ^ master_releaser_stage);
```

```
774.                                        free_slaves<=(free_slaves|slave_releaser_sta
    ge);
775.                                        shift_reg_master<= {shift_reg_master[DATA_WI
    DTH+ADDR_WIDTH:0],1'b0};
776.                                        if(s_p_conversion_counter == s_p_conversion_
    value)
777.                                            begin
778.                                                stage_2_state <= 1'b0;
779.                                                s_p_conversion_counter <= 5'b0;
780.                                            end
781.                                        else
782.                                            begin
783.                                                stage_2_state <= stage_2_state;
784.                                                s_p_conversion_counter <= s_p_conver
    sion_counter + 5'b1;
785.                                            end
786.                                    end
787.                                default:
788.                                    begin
789.                                        slave_en_flag <= 8'b0;
790.                                        s_p_conversion_counter <= 5'b0;
791.                                        s_p_conversion_value<=5'b0;
792.                                        shift_reg_master<={(DATA_WIDTH+ADDR_WIDTH+2)
    {1'b0}};
793.                                        last_state_stage_1_2<=4'b0;
794.                                        stage_2_state<=1'b0;
795.                                        free_slaves<=8'hff;
796.                                    end
797.                            endcase
798.                        end
799.
800.                end
801.
802.            always @(posedge clk or negedge reset_n)
803.                begin
804.                    if (~reset_n)
805.                        begin
806.                            slave_active_stage_3=8'b0;
807.                        end
808.                    else
809.                        begin
810.                            slave_active_stage_3=(slave_active_stage_3|(((~readvalid
    _avs_last) & readvalid_avs) |  ((~writevalid_avs_last) & writevalid_avs))) & (~last
    _state_stage_3_4);
811.                        end
812.
813.                end
814.
815.            //*************************** PIPELINE STAGE3: SELECT READY SLAVE PORTS
    AND SET CONTROL SIGNALS TO MUX AND DEMUX ************************//
816.
817.            always @(posedge clk or negedge reset_n)
818.                begin
819.                    if (~reset_n)
820.                        begin
821.                            slave_master_mux_slave<=8'b0;
822.                            slave_master_mux_slave_3b <= 4'b0;
823.                        end
824.                    else
825.                        begin         // 0001                   00000100
826.                            casex({slave_master_priority_with_active})
827.                                32'b1000xxxxxxxxxxxxxxxxxxxxxxxxxxxx:
828.                                    begin
829.                                        slave_master_mux_slave<=8'b10000000;
830.                                        slave_master_mux_slave_3b <= 4'b111;
831.                                    end
```

```
                                32'b0xxx1000xxxxxxxxxxxxxxxxxxxxxxxx:
                                    begin
                                        slave_master_mux_slave<=8'b01000000;
                                        slave_master_mux_slave_3b <= 4'b110;
                                    end
                                32'b0xxx0xxx1000xxxxxxxxxxxxxxxxxxxx:
                                    begin
                                        slave_master_mux_slave<=8'b00100000;
                                        slave_master_mux_slave_3b <= 4'b101;
                                    end
                                32'b0xxx0xxx0xxx1000xxxxxxxxxxxxxxxx:
                                    begin
                                        slave_master_mux_slave<=8'b00010000;
                                        slave_master_mux_slave_3b <= 4'b100;
                                    end
                                32'b0xxx0xxx0xxx0xxx1000xxxxxxxxxxxx:
                                    begin
                                        slave_master_mux_slave<=8'b00001000;
                                        slave_master_mux_slave_3b <= 4'b011;
                                    end
                                32'b0xxx0xxx0xxx0xxx0xxx1000xxxxxxxx:
                                    begin
                                        slave_master_mux_slave<=8'b00000100;
                                        slave_master_mux_slave_3b <= 4'b010;
                                    end
                                32'b0xxx0xxx0xxx0xxx0xxx0xxx1000xxxx:
                                    begin
                                        slave_master_mux_slave<=8'b00000010;
                                        slave_master_mux_slave_3b <= 4'b001;
                                    end
                                32'b0xxx0xxx0xxx0xxx0xxx0xxx0xxx1000:
                                    begin
                                        slave_master_mux_slave<=8'b00000001;
                                        slave_master_mux_slave_3b <= 4'b000;
                                    end
                                32'b01000xxx0xxx0xxx0xxx0xxx0xxx0xxx:
                                    begin
                                        slave_master_mux_slave<=8'b10000000;
                                        slave_master_mux_slave_3b <= 4'b111;
                                    end
                                32'b00xx01000xxx0xxx0xxx0xxx0xxx0xxx:
                                    begin
                                        slave_master_mux_slave<=8'b01000000;
                                        slave_master_mux_slave_3b <= 4'b110;
                                    end
                                32'b00xx00xx01000xxx0xxx0xxx0xxx0xxx:
                                    begin
                                        slave_master_mux_slave<=8'b00100000;
                                        slave_master_mux_slave_3b <= 4'b101;
                                    end
                                32'b00xx00xx00xx01000xxx0xxx0xxx0xxx:
                                    begin
                                        slave_master_mux_slave<=8'b00010000;
                                        slave_master_mux_slave_3b <= 4'b100;
                                    end
                                32'b00xx00xx00xx00xx01000xxx0xxx0xxx:
                                    begin
                                        slave_master_mux_slave<=8'b00001000;
                                        slave_master_mux_slave_3b <= 4'b011;
                                    end
                                32'b00xx00xx00xx00xx00xx01000xxx0xxx:
                                    begin
                                        slave_master_mux_slave<=8'b00000100;
                                        slave_master_mux_slave_3b <= 4'b010;
                                    end
                                32'b00xx00xx00xx00xx00xx00xx01000xxx:
```

```
898.                                    begin
899.                                        slave_master_mux_slave<=8'b00000010;
900.                                        slave_master_mux_slave_3b <= 4'b001;
901.                                    end
902.                                32'b00xx00xx00xx00xx00xx00xx00xx0100:
903.                                    begin
904.                                        slave_master_mux_slave<=8'b00000001;
905.                                        slave_master_mux_slave_3b <= 4'b000;
906.                                    end
907.
908.
909.                                32'b001000xx00xx00xx00xx00xx00xx00xx:
910.                                    begin
911.                                        slave_master_mux_slave<=8'b10000000;
912.                                        slave_master_mux_slave_3b <= 4'b111;
913.                                    end
914.                                32'b000x001000xx00xx00xx00xx00xx00xx:
915.                                    begin
916.                                        slave_master_mux_slave<=8'b01000000;
917.                                        slave_master_mux_slave_3b <= 4'b110;
918.                                    end
919.                                32'b000x000x001000xx00xx00xx00xx00xx:
920.                                    begin
921.                                        slave_master_mux_slave<=8'b00100000;
922.                                        slave_master_mux_slave_3b <= 4'b101;
923.                                    end
924.                                32'b000x000x000x001000xx00xx00xx00xx:
925.                                    begin
926.                                        slave_master_mux_slave<=8'b00010000;
927.                                        slave_master_mux_slave_3b <= 4'b100;
928.                                    end
929.                                32'b000x000x000x000x001000xx00xx00xx:
930.                                    begin
931.                                        slave_master_mux_slave<=8'b00001000;
932.                                        slave_master_mux_slave_3b <= 4'b011;
933.                                    end
934.                                32'b000x000x000x000x000x001000xx00xx:
935.                                    begin
936.                                        slave_master_mux_slave<=8'b00000100;
937.                                        slave_master_mux_slave_3b <= 4'b010;
938.                                    end
939.                                32'b000x000x000x000x000x000x001000xx:
940.                                    begin
941.                                        slave_master_mux_slave<=8'b00000010;
942.                                        slave_master_mux_slave_3b <= 4'b001;
943.                                    end
944.                                32'b000x000x000x000x000x000x000x0010:
945.                                    begin
946.                                        slave_master_mux_slave<=8'b00000001;
947.                                        slave_master_mux_slave_3b <= 4'b000;
948.                                    end
949.                                32'b0001000x000x000x000x000x000x000x:
950.                                    begin
951.                                        slave_master_mux_slave<=8'b10000000;
952.                                        slave_master_mux_slave_3b <= 4'b111;
953.                                    end
954.                                32'b00000001000x000x000x000x000x000x:
955.                                    begin
956.                                        slave_master_mux_slave<=8'b01000000;
957.                                        slave_master_mux_slave_3b <= 4'b110;
958.                                    end
959.                                32'b0000000000001000x000x000x000x000x:
960.                                    begin
961.                                        slave_master_mux_slave<=8'b00100000;
962.                                        slave_master_mux_slave_3b <= 4'b101;
963.                                    end
```

```
                                    32'b00000000000000001000x000x000x000x:
                                        begin
                                            slave_master_mux_slave<=8'b00010000;
                                            slave_master_mux_slave_3b <= 4'b100;
                                        end
                                    32'b00000000000000000001000x000x000x:
                                        begin
                                            slave_master_mux_slave<=8'b00001000;
                                            slave_master_mux_slave_3b <= 4'b011;
                                        end
                                    32'b00000000000000000000001000x000x: //// b00000100 // b010
                                        begin
                                            slave_master_mux_slave<=8'b00000100;
                                            slave_master_mux_slave_3b <= 4'b010;
                                        end
                                    32'b00000000000000000000000001000x:
                                        begin
                                            slave_master_mux_slave<=8'b00000010;
                                            slave_master_mux_slave_3b <= 4'b001;
                                        end
                                    32'b00000000000000000000000000000001:
                                        begin
                                            slave_master_mux_slave<=8'b00000001;
                                            slave_master_mux_slave_3b <= 4'b000;
                                        end
                                    default
                                        begin
                                            slave_master_mux_slave<=8'b0;
                                            slave_master_mux_slave_3b <= 4'b1000;
                                        end
                                endcase
                            end
                        end

            //*************************** PIPELINE STAGE 4: TRANSFER ACK AND DATA RO RESPECTIV MASTER FROM SLAVE PORTS WITH PARALLEL TO SERIAL CONVERSION ************ ************//

            always @(posedge clk or negedge reset_n)
                begin
                    if(!reset_n)
                        begin
                            s_p_conversion_counter_4 <= 4'b0;
                            s_p_conversion_value_4 <= 4'b0;
                            stage_4_state <= 1'b0;
                            shift_reg_slave <= {(DATA_WIDTH + 2){1'b0}};//(1'b0)*(DATA_WIDTH + 2);
                            last_state_stage_3_4<=8'b0;
                            last_state_stage_3_4_single<=8'b0;
                            master_enable<=4'b0;
                            slave_master_mux_slave_3b_tem<=4'b0;
                        end
                    else
                        begin
                            case(stage_4_state)
                                1'b0:
                                    begin
                                        last_state_stage_3_4_single<=8'b0;
                                        s_p_conversion_counter_4 <= 4'b0;
                                        if (readvalid_avs_reg[slave_master_mux_slave_3b])
                                            begin
                                                shift_reg_slave <= {readvalid_avs_reg[slave_master_mux_slave_3b], writevalid_avs_reg[slave_master_mux_slave_3b], readdata_avs_reg[slave_master_mux_slave_3b]};
```

```
1023.                                           s_p_conversion_value_4 <= DATA_WIDTH
    +1'b1;
1024.                                       end
1025.                                   else
1026.                                       begin
1027.                                           shift_reg_slave <= {readvalid_avs_re
    g[slave_master_mux_slave_3b], writevalid_avs_reg[slave_master_mux_slave_3b],{DATA_W
    IDTH{1'b0}}};
1028.                                           s_p_conversion_value_4 <= 4'd1;
1029.                                       end
1030.
1031.                                   if(slave_master_mux_slave!=8'b0)
1032.                                       begin
1033.                                           stage_4_state <= 1'b1;
1034.                                           last_state_stage_3_4<=(last_state_st
    age_3_4^slave_releaser_stage)| (1'b1 << slave_master_mux_slave_3b);
1035.                                           master_enable<=mux_demux_slave_corre
    sponding_master[slave_master_mux_slave_3b];
1036.                                       end
1037.                                   else
1038.                                       begin
1039.                                           stage_4_state <= stage_4_state;
1040.                                           last_state_stage_3_4<=(last_state_st
    age_3_4^slave_releaser_stage);
1041.                                           master_enable<=4'b0;
1042.                                       end
1043.                                   slave_master_mux_slave_3b_tem<=slave_master_
    mux_slave_3b;
1044.                               end
1045.                           1'b1:
1046.                               begin
1047.                                   slave_master_mux_slave_3b_tem<=slave_master_
    mux_slave_3b_tem;
1048.                                   master_enable<=4'b0;
1049.                                   shift_reg_slave[DATA_WIDTH + 1:0] <= {shift_
    reg_slave[DATA_WIDTH:0],1'b0};
1050.                                   last_state_stage_3_4<=(last_state_stage_3_4^
    slave_releaser_stage);
1051.                                   if(s_p_conversion_counter_4 == s_p_conversio
    n_value_4)
1052.                                       begin
1053.                                           stage_4_state <= 1'b0;
1054.                                           s_p_conversion_counter_4 <= 4'b0;
1055.                                           last_state_stage_3_4_single<=(1'b1 <
    < slave_master_mux_slave_3b_tem);
1056.                                       end
1057.                                   else
1058.                                       begin
1059.                                           stage_4_state <= stage_4_state;
1060.                                           s_p_conversion_counter_4 <= s_p_conv
    ersion_counter_4 + 4'b1;
1061.                                           last_state_stage_3_4_single<=8'b0;
1062.                                       end
1063.                               end
1064.                           endcase
1065.                       end
1066.               end
1067.
1068.               //************************** PIPELINE STAGE 5: RELeasE THE MASTER A
    ND SLAVE ************************//
1069.
1070.               always @(posedge clk or negedge reset_n) // stage 5 (releaser)
1071.                   begin
1072.                       if (~reset_n)
1073.                           begin
1074.                               master_releaser_stage<=4'b0;
```

```
                                slave_releaser_stage<=8'b0;
                            end
                        else
                            begin
                                case(last_state_stage_3_4_single)
                                    8'b10000000:
                                        begin
                                            master_releaser_stage<=mux_demux_slave_c
    orresponding_master[7];
                                            slave_releaser_stage<=8'b10000000;
                                        end
                                    8'b01000000:
                                        begin
                                            master_releaser_stage<=mux_demux_slave_c
    orresponding_master[6];
                                            slave_releaser_stage<=8'b01000000;
                                        end
                                    8'b00100000:
                                        begin
                                            master_releaser_stage<=mux_demux_slave_c
    orresponding_master[5];
                                            slave_releaser_stage<=8'b00100000;
                                        end
                                    8'b00010000:
                                        begin
                                            master_releaser_stage<=mux_demux_slave_c
    orresponding_master[4];
                                            slave_releaser_stage<=8'b00010000;
                                        end
                                    8'b00001000:
                                        begin
                                            master_releaser_stage<=mux_demux_slave_c
    orresponding_master[3];
                                            slave_releaser_stage<=8'b00001000;
                                        end
                                    8'b00000100:
                                        begin
                                            master_releaser_stage<=mux_demux_slave_c
    orresponding_master[2];
                                            slave_releaser_stage<=8'b00000100;
                                        end
                                    8'b00000010:
                                        begin
                                            master_releaser_stage<=mux_demux_slave_c
    orresponding_master[1];
                                            slave_releaser_stage<=8'b00000010;
                                        end
                                    8'b00000001:
                                        begin
                                            master_releaser_stage<=mux_demux_slave_c
    orresponding_master[0];
                                            slave_releaser_stage<=8'b00000001;
                                        end
                                    default:
                                        begin
                                            master_releaser_stage<=4'b0;
                                            slave_releaser_stage<=8'b0;
                                        end
                                endcase
                            end
                    end

        endmodule
```