DEPARTMENT OF ELECTRONIC AND TELECOMMUNICATION

ENGINEERING

UNIVERSITY OF MORATUWA

EN4020 – ADVANCED DIGITAL SYSTEMS

SYSTEM BUS DESIGN

Team Members	*Index Number*
K. M. R. Wijitharathna	180717E
T. D. R. V. Perera	180468N
N. P. A. Mendis	180398A
P. D. Mahawela	180378M

This is submitted as a partial fulfillment for the module EN4020 : Advance Digital Systems

March 19, 2023

Table of Contents

1 Introduction **3**
1.1 Overview . . . 3
1.2 Block Design . . . 3

2 Methodology **4**
2.1 Master Module . . . 4
2.2 Slave Module . . . 5
2.3 Bus Arbitration . . . 6
2.4 Multiplexer . . . 7
2.5 Decoder . . . 8
2.6 Write Multiplexer . . . 8
2.7 Top level Module . . . 9

3 Simulation and Implementation **10**
3.1 Master Module Testbench . . . 10
3.2 Slave Module Testbench . . . 10
3.3 Slave Memory Storage . . . 11
3.4 Top level Module Testbench . . . 11

4 Conclusion **12**
4.1 Limitations of the Design . . . 12
4.2 Improvements . . . 12

5 Appendices **13**

List of Figures

1 System Block Diagram 3
2 Master Module 4
3 Slave Module 5
4 Arbiter Module 6
5 Master Module 7
6 Decoder Module 8
7 Write Multiplexer Module 8
8 Top Level Module 9
9 Master Testbench Waveforms 10
10 Slave Testbench Waveforms 10
11 System Block Diagram 11
12 dout Waveforms 11

List of Tables

1 Master Signal Descriptions 5
2 Slave Signal Descriptions 6
3 Arbiter Signal Descriptions 7
4 Multiplexer Signal Descriptions 7
5 Decoder Signal Descriptions 8
6 Write-mux Signal Descriptions 9

1 Introduction

1.1 Overview

A system bus is a communication pathway that enables communication between various components within a digital system. This report discusses implementing a custom system bus design and the design specifications, operating principles, design, and verification. The introduction section discusses the high-level block diagram to deliver an understanding of the implemented design. Next, the methodology section describes the operating principles of each system bus module and how all those components behave as a whole. The simulation section documents how each module was tested and verified the complete functionality. Finally, we discuss our proposed design's limitations and potential improvements.

We were tasked to create a custom system bus that had multiple slaves and masters, which could perform cross-communication between masters and slaves. First, we approached the design by specifying and defining the system specifications. Then by designing a block diagram of the structure, we implemented each module individually. Next, we wrote individual test benches for each module to ensure the proper functionality and behaviour of each module we designed. Then after this step, we created a top module that represents the complete bus design with all the modules connected. Finally, a testbench was written to test the system bus and demonstrate a data transaction between a master and a slave.

1.2 Block Design

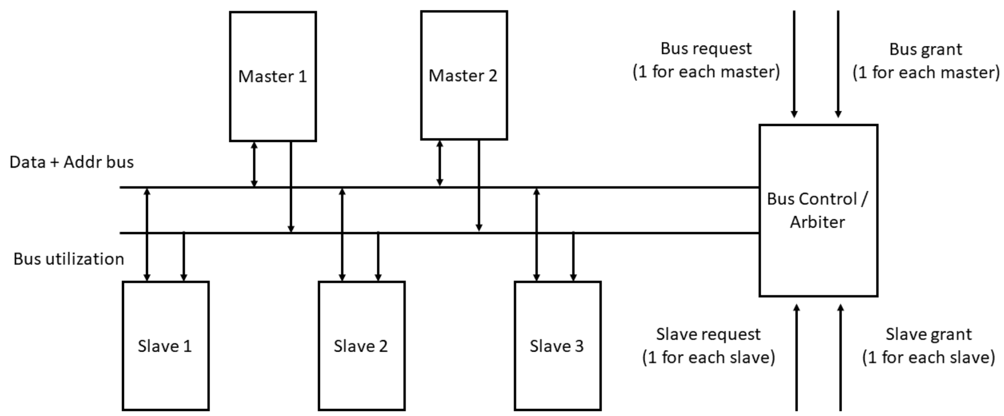

Fig. 1: System Block Diagram

The figure represents the high-level block diagram of our proposed design. Currently, the design holds two masters and three slaves interconnected through the shared bus. Each master and slave modules have separate interfaces enabling them to interconnect with the system bus. In addition, the bus controller module handles the requests made by the masters granting them access to utilize the system bus for communication with slaves.

The figure represents how the bus was implemented in our top Verilog module. This representation will help us better to understand the internal implementation of our proposed system bus.

2 Methodology

2.1 Master Module

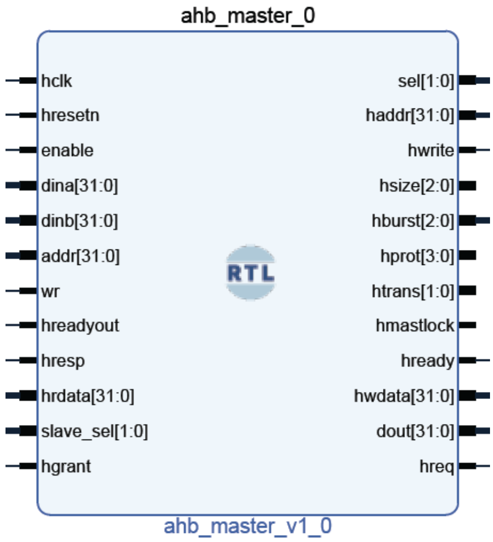

Fig. 2: Master Module

The master module uses a state machine implemented using two always blocks, one triggered by the clock and the reset inputs and the other for the state transition and output logic. The first always block loads the next state to the current state on the rising edge of the clock or the negative edge of the active low reset signal. All the output ports are set to their default values if the reset is low. Otherwise, the next state is assigned.

A case statement is used to implement the state transitions and output logic in the second always block. The initial state is the idle state where the master waits for an enable signal. When received, it will send a hreq to the arbiter to request a grant to transmit. When the arbiter issues the hgrant signal, the module checks for the write enable signal (wr) to determine whether it is a read or a write signal. If it is a write, the module transitions into read and write states, sets the corresponding output ports, and waits until the enable signal is set to low and returns to idle state.

Table 1: Master Signal Descriptions

Signal	I/O	Width	Description
hclk	input	1	Global clock signal
hresetn	input	1	Global reset signal
enable	input	1	Global enable signal
dina	input	32	Data input port a
dinb	input	32	Data input port b
addr	input	32	Address input port
wr	input	1	Enable write operation
hreadyout	input	1	Slave ready
hresp	input	1	Slave response
hrdata	input	32	Data in from slave
slave sel	input	2	Slave select
hgrant	input	1	Bus access grant indication
sel	output	2	Slave select output
haddr	output	32	Slave address output
hwrite	output	1	Write operation indicator
hsize	output	3	Size of the transfer
hburst	output	3	Write operation burst size
hprot	output	4	Protection information
htrans	output	2	Transfer type
hmastlock	output	1	Master lock output
hready	output	1	Master busy/complete indication
hwdata	output	32	Master data output
dout	output	32	External debug port for data
hreq	output	1	Bus access request signal

2.2 Slave Module

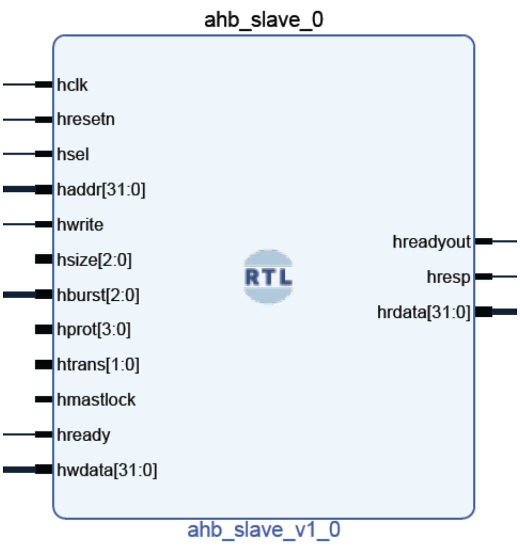

Fig. 3: Slave Module

The slave module is implemented using a state machine. Initially the module will be in the idle state with the output ports set to default values until the slave select(hsel) signal is set high. Then it checks whether the slave is ready(hready signal is high) and the state of the slave write(hwrite) signal. If hwrite is high then the module transitions to the write state(s2); otherwise to the read state(s3). In these write and read states, the output ports will be set accordingly and wait as long as the slave select is high. When the slave select goes low, the module transitions to the idle state and stay there until the slave

Table 2: Slave Signal Descriptions

Signal	I/O	Width	Description
hclk	input	1	Global clock signal
hresetn	input	1	Global reset signal
hsel	input	1	Slave select signal
haddr	input	32	Slave address port in
hwrite	input	1	Write signal for slave
hsize	input	3	Size of the transfer
hburst	input	3	Burst type
hmastlock	input	1	Master lock signal
hready	input	1	Slave ready signal
hwdata	input	32	Write data
hreadyout	output	1	Indicates whether transfer is pending or complete
hresp	output	1	Indicates whether transfer is okay or error
hrdata	output	32	Read data

select becomes high again to initiate a transaction.

2.3 Bus Arbitration

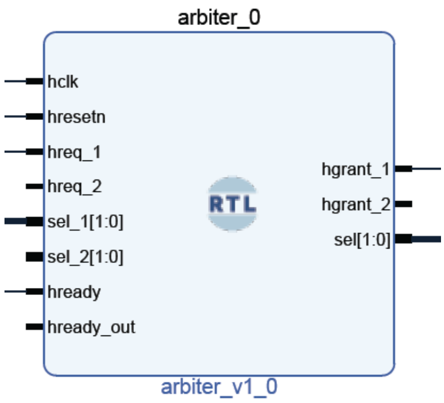

Fig. 4: Arbiter Module

The arbiter module arbitrates between multiple masters contending to access the shared bus. This implementation receives requests from two masters through the hreq_n signals. The module is implemented using a state machine. When the active-low reset signal is set low, the module is set to the idle state. At every positive edge of the clock, if the reset is not asserted, the current state is synchronized with the next state.

In the idle state, the module checks whether one of the hreq signals is set to high and transitions the module into the respective GRANT_n state. If both are low, it will stay in an idle state. In grant states, the module grants access to the respective master by setting the respective hgrant signal high and passing its selected slave through the sel signal. If the other hreq goes high when the module is in one grant state, it will switch to the other grant state. If both hreq signals go low, the module switches back to idle.

Table 3: Arbiter Signal Descriptions

Signal	I/O	Width	Description
hclk	input	1	Global clock signal
hresetn	input	1	Global reset signal
hreq_n	input	1	Bus request signal from master n
sel_n	input	2	Slave select signal from master n
hready	input	1	Slave ready signal
hgrant_n	output	1	Bus grant signal for master n
sel	output	2	Slave select signal

2.4 Multiplexer

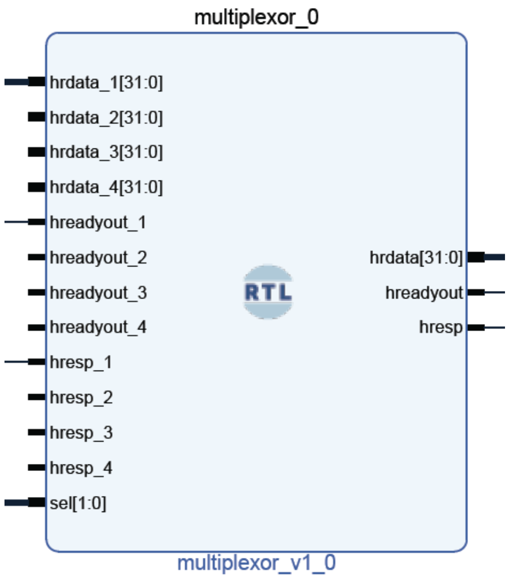

Fig. 5: Master Module

Table 4: Multiplexer Signal Descriptions

Signal	I/O	Width	Description
hrdata_n	input	32	Data in from slave n
hready_out_n	input	1	Slave ready signal from slave n
hresp_n	input	1	Slave response signal from slave n
sel	input	2	Slave select signal from arbitor
hrdata	output	32	Selected slave data out signal
hreadyout	output	1	Selected slave ready out signal
hresp	output	1	Selected slave response out signal

The multiplexer module selects one of multiple set of signals from different slaves. This implementation provides multiplexing between signals from three slaves. The slave is chosen using the 2-bit wide sel signal whenever any input port changes. Depending on the sel signal, respective data and the ready signals from the slave is passed on to the module's output.

2.5 Decoder

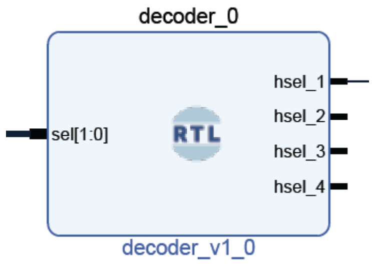

Fig. 6: Decoder Module

The 2-to-4 decoder generates four output signals corresponding to a 2-bit input signal. The selected output is set to '1' while the other output signals are '0'. This module is implemented using a case statement and triggered by a change in the input sel signal.

Table 5: Decoder Signal Descriptions

Signal	I/O	Width	Description
sel	input	2	Slave select signal from arbiter
hsel_n	output	1	Demultiplexed output from the decoder to select the slave

2.6 Write Multiplexer

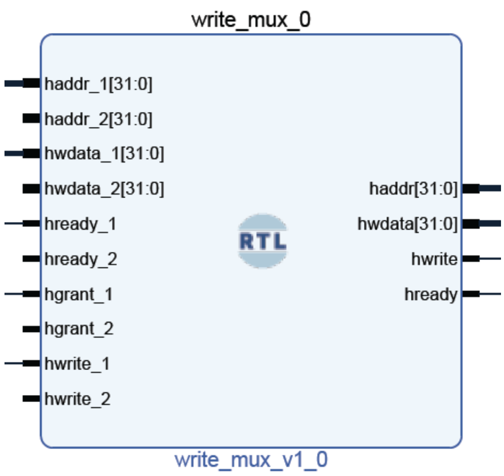

Fig. 7: Write Multiplexer Module

This write multiplexer module selects the signals from the selected master based on the state of the hgrant signals. When the master is selected by the hgrant signal, its slave address, data, ready and write signals are passed on the outputs.

Table 6: Write-mux Signal Descriptions

Signal	I/O	Width	Description
haddr_n	input	32	Slave address input
hwdata_n	input	32	Write data to slave
hready_n	input	1	Master ready signal
hgrant_n	input	1	Master n bus grant signal
hwrite_n	input	1	Write signal input from master n
haddr	output	32	Slave address output
hwdata	output	32	Write data
hwrite	output	1	Write data signal to slave
hready	output	1	Master ready signal to slave

2.7 Top level Module

Figure 8 represents the block diagram of our top-level module. It includes the main components of our system bus and the signal lines interconnecting each module. The interconnect has 2 masters and 3 slaves. It also contains the connections for the external pins which is used to give signals to trigger the system bus.

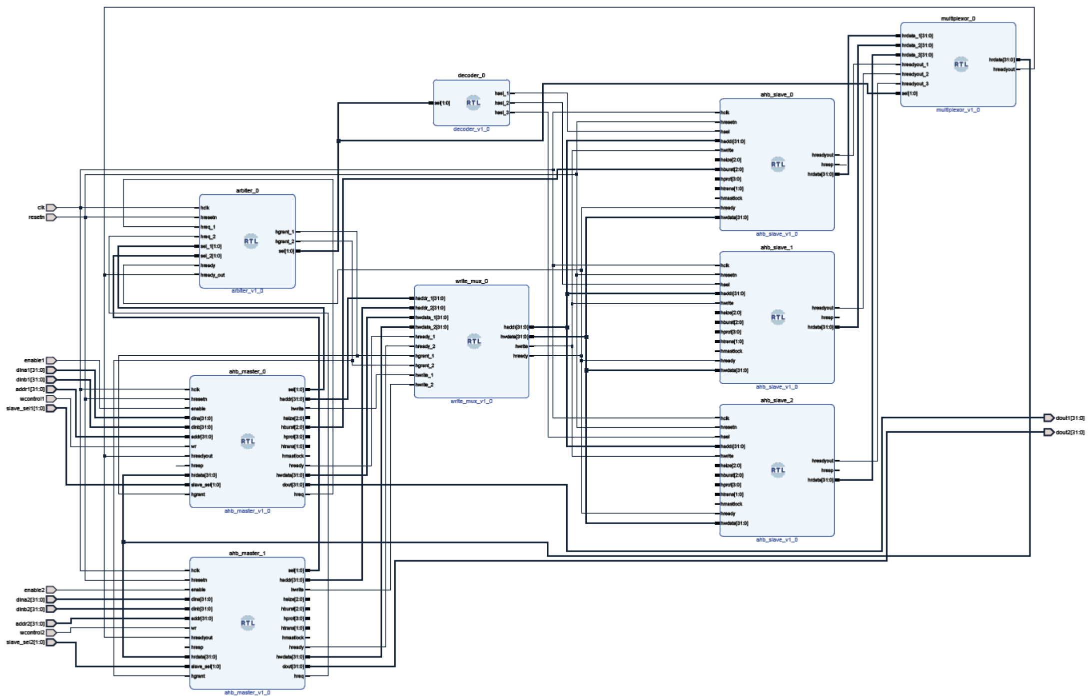

Fig. 8: Top Level Module

3 Simulation and Implementation

3.1 Master Module Testbench

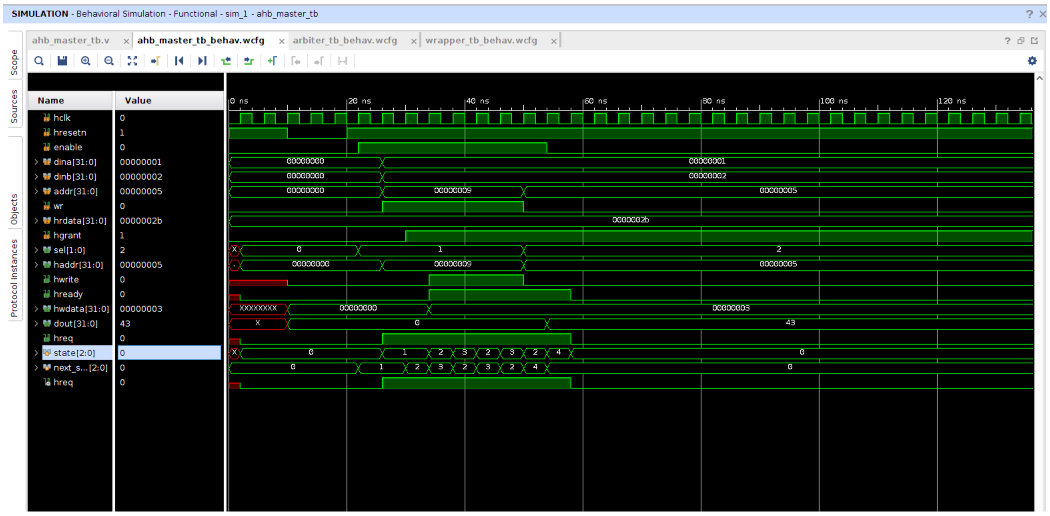

Fig. 9: Master Testbench Waveforms

This simulation waveform clearly shows the functionality of the master module. It first sends a request signal to initiate the transaction. When the arbiter grants access to the master to use the bus, it proceeds with the transaction with read or write.

3.2 Slave Module Testbench

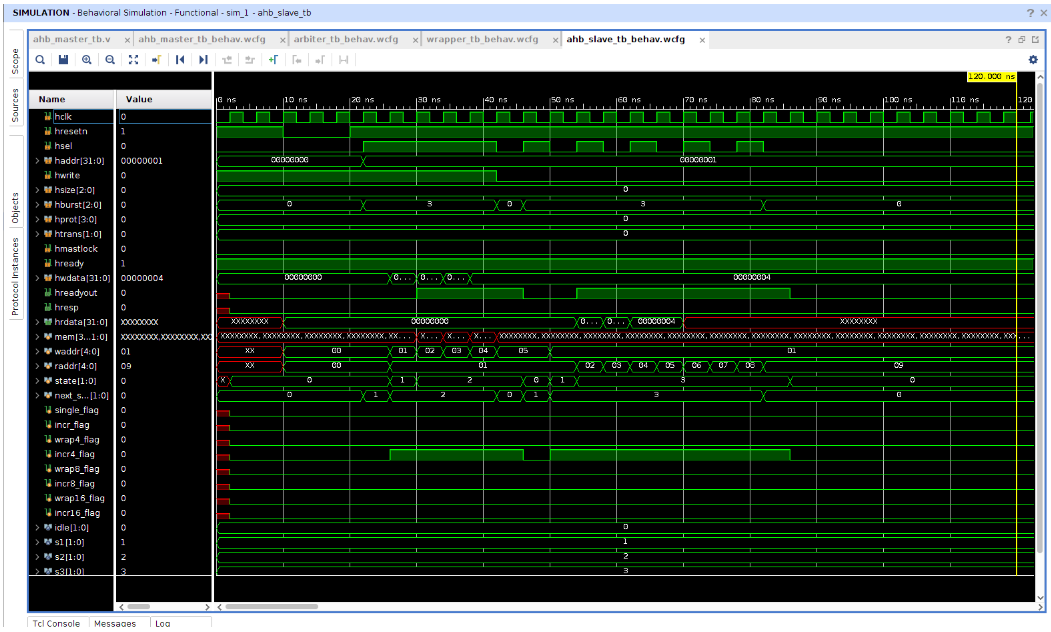

Fig. 10: Slave Testbench Waveforms

This simulation waveform shows the correct functionality of the slave module. When the master sends the data to the data, the slave module waits until the ready signal is issued. When the ready signal is issued, the slave module sends or receives data depending on the level of the slave write signal.

3.3 Slave Memory Storage

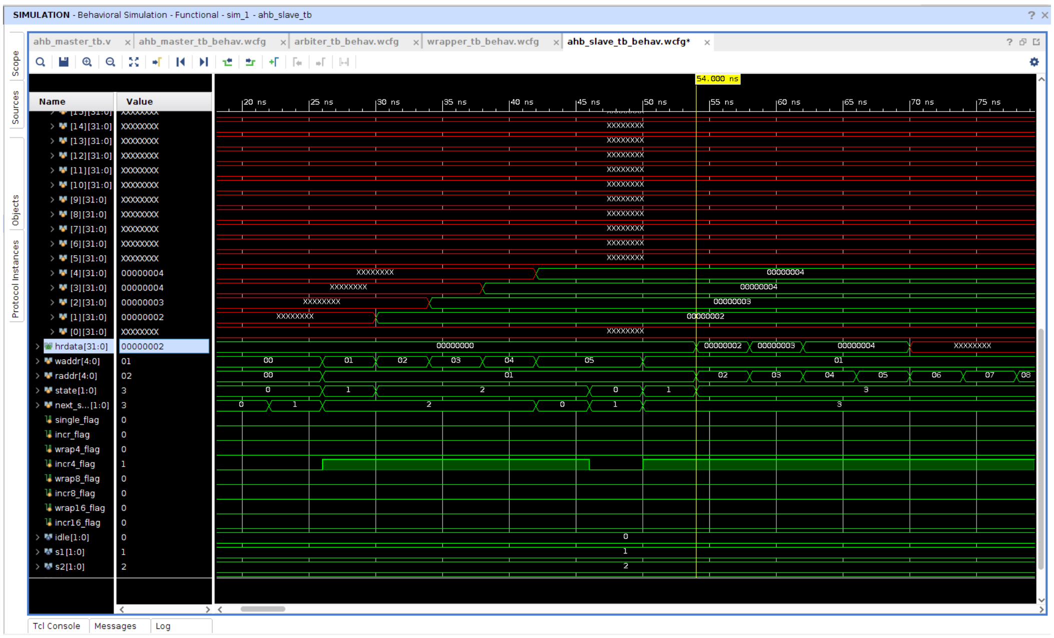

Fig. 11: System Block Diagram

3.4 Top level Module Testbench

The top-level module testbench uses the masters to write data to slaves. Then it read data back from slaves to dout port. The dout waveform represents the read data.

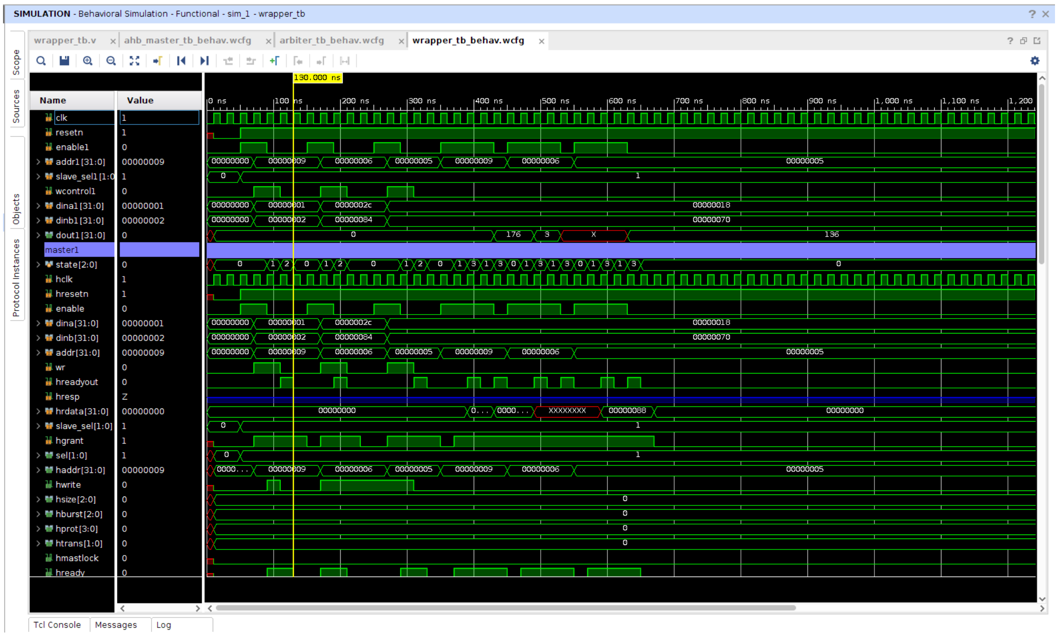

Fig. 12: dout Waveforms

4 Conclusion

4.1 Limitations of the Design

A few of the limitations of our current implementation are listed below.

- The current implementation can only communicate within a single bus. This implementation cannot communicate between multiple buses.
- The current implementation cannot perform bursts. Therefore, we can only send data at a fixed data rate.
- Since the masters are directly connected to the module, it will be difficult to add more masters conveniently. This implementation could be improved to be more modular.

4.2 Improvements

A few of the improvements that can be made to address the above limitations are listed below.

- We can create a bus bridge and build the interface to communicate between multiple buses.
- We can implement burst to increase the bit rate written at a time. This will improve the data throughput.

5 Appendices

Top wrapper testbench

```
    module wrapper_tb(

    );
reg clk;
reg resetn;
reg [31:0] dina1;
reg [31:0] dinb1;
reg enable1;
reg [1:0] slave_sel1;
reg [31:0] addr1;
reg wcontrol1;

wire [31:0] dout1;

design_1_wrapper uut (
    .addr1(addr1),
    .clk(clk),
    .dina1(dina1),
    .dinb1(dinb1),
    .dout1(dout1),
    .enable1(enable1),
    .resetn(resetn),
    .slave_sel1(slave_sel1),
    .wcontrol1(wcontrol1)
);

//clock generation
initial begin
    clk = 0;
    forever begin
        #10
        clk = ~clk;
    end
end


task write( input [1:0] sel, input [31:0] address, input [31:0] a, input [31:0] b);
begin
  @(posedge clk)
  slave_sel1 = sel;
  enable1 = 1'b1;
  @(posedge clk)
  dina1 = a;
  dinb1 = b;
  addr1 = address;
  wcontrol1 = 1'b1;
  enable1 = 1'b1;
  @(posedge clk) //STATE SHOULD BE IDLE
  enable1 = 1'b0;
  @(posedge clk)
  wcontrol1 = 1'b0;
  #40;
end
endtask
```

```
task read(input [1:0] sel, input [31:0] address);
begin
  @(posedge clk)
  slave_sel1 = sel;
  enable1 = 1'b1;
  addr1 = address;
  @(posedge clk)
  wcontrol1 = 1'b0;
  @(posedge clk)
  wcontrol1 = 1'b0;
  @(posedge clk)
  wcontrol1 = 1'b0;
  @(posedge clk)
  enable1 = 1'b0;
end
endtask

initial begin
  enable1 = 1'b0;
  dina1 = 32'd0;
  dinb1 = 32'd0;
  addr1 = 32'd0;
  wcontrol1 = 1'b0;
  //hrdata = 32'd43;
  slave_sel1 = 2'b00;
  #10 resetn = 0;
  #40 resetn = 1;


  // write
  write(2'b01, 32'd9, 32'd1, 32'd2);  //write slave1 addr 9 => 3
  //write(2'b10, 32'd8, 32'd32, 32'd3); //write slave2 addr 8 => 35
  //write(2'b10, 32'd7, 32'd7, 32'd11); //write slave2 addr 7 => 18
  write(2'b01, 32'd6, 32'd44, 32'd132); //write slave1 addr 6 => 176
  // read
  //read(2'b10, 32'd8);                   //read slave2 addr8
  write(2'b01, 32'd5, 32'd24, 32'd112); //write slave1 addr 5 => 136
  read(2'b01, 32'd9);                   //read slave0 add9
  read(2'b01, 32'd6);                   //read slave1 addr7
  read(2'b01, 32'd5);                   //read slave0 addr6
  //read(2'b01, 32'd5);                   //read slave0 addr5

end

endmodule
```

master module code

```
  module ahb_master(
  input hclk,
  input hresetn,
  input enable,
  input [31:0] dina,
  input [31:0] dinb,
  input [31:0] addr,
  input wr,
  input hreadyout,
  input hresp,
  input [31:0] hrdata,
```

```
  input [1:0] slave_sel,
  input hgrant,

  output reg [1:0] sel,
  output reg [31:0] haddr,
  output reg hwrite,
  output reg [2:0] hsize,
  output reg [2:0] hburst,
  output reg [3:0] hprot,
  output reg [1:0] htrans,
  output reg hmastlock,
  output reg hready,
  output reg [31:0] hwdata,
  output reg [31:0] dout,
  output reg hreq
);

//------------------------------------------------------
// The definitions for state machine
//------------------------------------------------------

reg [2:0] state, next_state;
parameter idle = 2'b00, /*await = 3'b001*/ s1 = 2'b01, s2 = 2'b10, s3 = 2'b11;


//------------------------------------------------------
// The state machine
//------------------------------------------------------

always @(posedge hclk, negedge hresetn) begin
  if(!hresetn) begin
    state <= idle;
  end
  else begin
    state <= next_state;
  end
end


always @(*) begin
  case(state)
    idle: begin
      if(enable == 1'b1) begin
        hreq <= 1;
        if((enable ==1'b1) && (hgrant==1'b1))
            next_state = s1;
      end
      /*if(enable ==1)
        next_state =s1;
      */
      else begin
        next_state = idle;
      end
    end
    /*await: begin
        if(hgrant == 1) begin
            next_state = s1;
        end
```

```
        else begin
            next_state = await;
        end
    end */
    s1: begin
      if(wr == 1'b1) begin
        next_state = s2;
      end
      else begin
        next_state = s3;
      end
    end
    s2: begin
      if(enable == 1'b1) begin
        next_state = s1;
      end
      else begin
        next_state = idle;
      end
    end
    s3: begin
      if(enable == 1'b1) begin
        next_state = s1;
      end
      else begin
        next_state = idle;
      end
    end
    default: begin
      next_state = idle;
    end
  endcase
end

always @(posedge hclk, negedge hresetn) begin
  if(!hresetn) begin
    sel <= 2'b00;
    haddr <= 32'h0000_0000;
    hwrite <= 1'b0;
    hsize <= 3'b000;
    hburst <= 3'b000;
    hprot <= 4'b0000;
    htrans <= 2'b00;
    hmastlock <= 1'b0;
    hready <= 1'b0;
    hwdata <= 32'h0000_0000;
    dout <= 32'h0000_0000;
  end
  else begin
    case(next_state)
      idle: begin
        sel <= slave_sel;
        haddr <= addr;
        hwrite <= hwrite;
        hburst <= hburst;
        hready <= 1'b0;
        hwdata <= hwdata;
        dout <= dout;
```

```
      hreq <=0;
    end
    /*await: begin
      sel <= slave_sel;
      haddr <= addr;
      hwrite <= wr; //caution earlier hwrite
      hburst <= hburst;
      hready <= 1'b1; //caution
      hwdata <= hwdata;
      dout <= dout;
      hreq <= 1;
    end*/

    s1: begin
      sel <=  slave_sel;
      haddr <= addr;
      hwrite <= wr;
      hburst <= 3'b000;
      hready <= 1'b1;
      hwdata <= dina+dinb;
      dout <= dout;
    end
    s2: begin
      sel <= sel;
      haddr <= addr;
      hwrite <= wr;
      hburst <= 3'b000;
      hready <= 1'b1;
      hwdata <= dina+dinb;
      dout <= dout;
    end
    s3: begin
      sel <= sel;
      haddr <= addr;
      hwrite <= wr;
      hburst <= 3'b000;
      hready <= 1'b1;
      hwdata <= hwdata;
      dout <= hrdata;
    end
    default: begin
      sel <= slave_sel;
      haddr <= haddr;
      hwrite <= hwrite;
      hburst <= hburst;
      hready <= 1'b0;
      hwdata <= hwdata;
      dout <= dout;
    end
  endcase
 end
end


endmodule
```

slave module code

```
module ahb_slave(
```

```
  input hclk,
  input hresetn,
  input hsel,
  input [31:0] haddr,
  input hwrite,
  input [2:0] hsize,
  input [2:0] hburst,
  input [3:0] hprot,
  input [1:0] htrans,
  input hmastlock,
  input hready,
  input [31:0] hwdata,
  output reg hreadyout,
  output reg hresp,
  output reg [31:0] hrdata
);

//----------------------------------------------------------------------
// The definitions for intern registers for data storge
//----------------------------------------------------------------------
reg [31:0] mem [31:0];
reg [4:0] waddr;
reg [4:0] raddr;

//----------------------------------------------------------------------
// The definition for state machine
//----------------------------------------------------------------------
reg [1:0] state;
reg [1:0] next_state;
localparam idle = 2'b00,s1 = 2'b01,s2 = 2'b10,s3 = 2'b11;


//----------------------------------------------------------------------
// The definition for burst feature
//----------------------------------------------------------------------
reg single_flag;
reg incr_flag;
reg wrap4_flag;
reg incr4_flag;
reg wrap8_flag;
reg incr8_flag;
reg wrap16_flag;
reg incr16_flag;




//----------------------------------------------------------------------
// The state machine
//----------------------------------------------------------------------

always @(posedge hclk, negedge hresetn) begin
  if(!hresetn) begin
    state <= idle;
  end
  else begin
    state <= next_state;
  end
```

```
end

always @(*) begin
  case(state)
    idle: begin
      single_flag = 1'b0;
      incr_flag = 1'b0;
      wrap4_flag = 1'b0;
      incr4_flag = 1'b0;
      wrap8_flag = 1'b0;
      incr8_flag = 1'b0;
      wrap16_flag = 1'b0;
      incr16_flag = 1'b0;
      if(hsel == 1'b1) begin
        next_state = s1;
      end
      else begin
        next_state = idle;
      end
    end
    s1: begin
      case(hburst)
        // single transfer burst
        3'b000: begin
          single_flag = 1'b1;
          incr_flag = 1'b0;
          wrap4_flag = 1'b0;
          incr4_flag = 1'b0;
          wrap8_flag = 1'b0;
          incr8_flag = 1'b0;
          wrap16_flag = 1'b0;
          incr16_flag = 1'b0;
        end
        // incrementing burst of undefined length
        3'b001: begin
          single_flag = 1'b0;
          incr_flag = 1'b1;
          wrap4_flag = 1'b0;
          incr4_flag = 1'b0;
          wrap8_flag = 1'b0;
          incr8_flag = 1'b0;
          wrap16_flag = 1'b0;
          incr16_flag = 1'b0;
        end
        // 4-beat wrapping burst
        3'b010: begin
          single_flag = 1'b0;
          incr_flag = 1'b0;
          wrap4_flag = 1'b1;
          incr4_flag = 1'b0;
          wrap8_flag = 1'b0;
          incr8_flag = 1'b0;
          wrap16_flag = 1'b0;
          incr16_flag = 1'b0;
        end
        // 4-beat incrementing burst
        3'b011: begin
          single_flag = 1'b0;
```

```
        incr_flag = 1'b0;
        wrap4_flag = 1'b0;
        incr4_flag = 1'b1;
        wrap8_flag = 1'b0;
        incr8_flag = 1'b0;
        wrap16_flag = 1'b0;
        incr16_flag = 1'b0;
      end
      // 8-beat wrapping burst
      3'b100: begin
        single_flag = 1'b0;
        incr_flag = 1'b0;
        wrap4_flag = 1'b0;
        incr4_flag = 1'b0;
        wrap8_flag = 1'b1;
        incr8_flag = 1'b0;
        wrap16_flag = 1'b0;
        incr16_flag = 1'b0;
      end
      // 8-beat incrementing burst
      3'b101: begin
        single_flag = 1'b0;
        incr_flag = 1'b0;
        wrap4_flag = 1'b0;
        incr4_flag = 1'b0;
        wrap8_flag = 1'b0;
        incr8_flag = 1'b1;
        wrap16_flag = 1'b0;
        incr16_flag = 1'b0;
      end
      // 16-beat wrapping burst
      3'b110: begin
        single_flag = 1'b0;
        incr_flag = 1'b0;
        wrap4_flag = 1'b0;
        incr4_flag = 1'b0;
        wrap8_flag = 1'b0;
        incr8_flag = 1'b0;
        wrap16_flag = 1'b1;
        incr16_flag = 1'b0;
      end
      // 16-beat incrementing burst
      3'b111: begin
        single_flag = 1'b0;
        incr_flag = 1'b0;
        wrap4_flag = 1'b0;
        incr4_flag = 1'b0;
        wrap8_flag = 1'b0;
        incr8_flag = 1'b0;
        wrap16_flag = 1'b0;
        incr16_flag = 1'b1;
      end
      // default
      default: begin
        single_flag = 1'b0;
        incr_flag = 1'b0;
        wrap4_flag = 1'b0;
        incr4_flag = 1'b0;
```

```
          wrap8_flag = 1'b0;
          incr8_flag = 1'b0;
          wrap16_flag = 1'b0;
          incr16_flag = 1'b0;
        end
      endcase


      if((hwrite == 1'b1) && (hready == 1'b1)) begin
        next_state = s2;
      end
      else if((hwrite == 1'b0) && (hready == 1'b1)) begin
        next_state = s3;
      end
      else begin
        next_state = s1;
      end
    end
    s2: begin
      case(hburst)
        // single transfer burst
        3'b000: begin
          if(hsel == 1'b1) begin
            next_state = s1;
          end
          else begin
            next_state = idle;
          end
        end
        // incrementing burst of undefined length
        3'b001: begin
          next_state = s2;
        end
        // 4-beat wrapping burst
        3'b010: begin
          next_state = s2;
        end
        // 4-beat incrementing burst
        3'b011: begin
          next_state = s2;
        end
        // 8-beat wrapping burst
        3'b100: begin
          next_state = s2;
        end
        // 8-beat incrementing burst
        3'b101: begin
          next_state = s2;
        end
        // 16-beat wrapping burst
        3'b110: begin
          next_state = s2;
        end
        // 16-beat incrementing burst
        3'b111: begin
          next_state = s2;
        end
        // default
```

```
        default: begin
          if(hsel == 1'b1) begin
            next_state = s1;
          end
          else begin
            next_state = idle;
          end
        end
      endcase
    end
    s3: begin
      case(hburst)
        // single transfer burst
        3'b000: begin
          if(hsel == 1'b1) begin
            next_state = s1;
          end
          else begin
            next_state = idle;
          end
        end
        // incrementing burst of undefined length
        3'b001: begin
          next_state = s3;
        end
        // 4-beat wrapping burst
        3'b010: begin
          next_state = s3;
        end
        // 4-beat incrementing burst
        3'b011: begin
          next_state = s3;
        end
        // 8-beat wrapping burst
        3'b100: begin
          next_state = s3;
        end
        // 8-beat incrementing burst
        3'b101: begin
          next_state = s3;
        end
        // 16-beat wrapping burst
        3'b110: begin
          next_state = s3;
        end
        // 16-beat incrementing burst
        3'b111: begin
          next_state = s3;
        end
        // default
        default: begin
          if(hsel == 1'b1) begin
            next_state = s1;
          end
          else begin
            next_state = idle;
          end
        end
```

```
      endcase
    end
    default: begin
      next_state = idle;
    end
  endcase
end

always @(posedge hclk, negedge hresetn) begin
  if(!hresetn) begin
    hreadyout <= 1'b0;
    hresp <= 1'b0;
    hrdata <= 32'h0000_0000;
    waddr <= 5'b0000_0;
    raddr <= 5'b0000_0;
  end
  else begin
    case(next_state)
      idle: begin
        hreadyout <= 1'b0;
        hresp <= 1'b0;
        hrdata <= hrdata;
        waddr <= waddr;
        raddr <= raddr;
      end
      s1: begin
        hreadyout <= 1'b0;
        hresp <= 1'b0;
        hrdata <= hrdata;
        waddr <= haddr;
        raddr <= haddr;
      end
      s2: begin
      case({single_flag,incr_flag,wrap4_flag,incr4_flag,wrap8_flag,incr8_flag,wrap16_flag,incr16_flag})
          // single transfer
          8'b1000_0000: begin
            hreadyout <= 1'b1;
            hresp <= 1'b0;
            mem[waddr] <= hwdata;
          end
          // incre
          8'b0100_0000: begin
            hreadyout <= 1'b1;
            hresp <= 1'b0;
            mem[waddr] <= hwdata;
            waddr <= waddr + 1'b1;
          end
          // wrap 4
          8'b0010_0000: begin
            hreadyout <= 1'b1;
            hresp <= 1'b0;
            if(waddr < (haddr + 2'd3)) begin
              mem[waddr] <= hwdata;
              waddr <= waddr + 1'b1;
            end
            else begin
              mem[waddr] <= hwdata;
              waddr <= haddr;
```

```
        end
      end
      // incre 4
      8'b0001_0000: begin
        hreadyout <= 1'b1;
        hresp <= 1'b0;
        mem[waddr] <= hwdata;
        waddr <= waddr + 1'b1;
      end
      // wrap 8
      8'b0000_1000: begin
        hreadyout <= 1'b1;
        hresp <= 1'b0;
        if(waddr < (haddr + 3'd7)) begin
          mem[waddr] <= hwdata;
          waddr <= waddr + 1'b1;
        end
        else begin
          mem[waddr] <= hwdata;
          waddr <= haddr;
        end
      end
      // incre 8
      8'b0000_0100: begin
        hreadyout <= 1'b1;
        hresp <= 1'b0;
        mem[waddr] <= hwdata;
        waddr <= waddr + 1'b1;
      end
      // wrap 16
      8'b0000_0010: begin
        hreadyout <= 1'b1;
        hresp <= 1'b0;
        if(waddr < (haddr + 4'd15)) begin
          mem[waddr] <= hwdata;
          waddr <= waddr + 1'b1;
        end
        else begin
          mem[waddr] <= hwdata;
          waddr <= haddr;
        end
      end
      // incre 16
      8'b0000_0001: begin
        hreadyout <= 1'b1;
        hresp <= 1'b0;
        mem[waddr] <= hwdata;
        waddr <= waddr + 1'b1;
      end
      // default
      default: begin
        hreadyout <= 1'b1;
        hresp <= 1'b0;
      end
    endcase
  end
  s3: begin
case({single_flag,incr_flag,wrap4_flag,incr4_flag,wrap8_flag,incr8_flag,wrap16_flag,incr16_flag})
```

```
// single transfer
8'b1000_0000: begin
  hreadyout <= 1'b1;
  hresp <= 1'b0;
  hrdata <= mem[raddr];
end
// incre
8'b0100_0000: begin
  hreadyout <= 1'b1;
  hresp <= 1'b0;
  hrdata <= mem[raddr];
  raddr <= raddr + 1'b1;
end
// wrap 4
8'b0010_0000: begin
  hreadyout <= 1'b1;
  hresp <= 1'b0;
  if(raddr < (haddr + 2'd3)) begin
    hrdata <= mem[raddr];
    raddr <= raddr + 1'b1;
  end
  else begin
    hrdata <= mem[raddr];
    raddr <= haddr;
  end
end
// incre 4
8'b0001_0000: begin
  hreadyout <= 1'b1;
  hresp <= 1'b0;
  hrdata <= mem[raddr];
  raddr <= raddr + 1'b1;
end
// wrap 8
8'b0000_1000: begin
  hreadyout <= 1'b1;
  hresp <= 1'b0;
  if(raddr < (haddr + 3'd7)) begin
    hrdata <= mem[raddr];
    raddr <= raddr + 1'b1;
  end
  else begin
    hrdata <= mem[raddr];
    raddr <= haddr;
  end
end
// incre 8
8'b0000_0100: begin
  hreadyout <= 1'b1;
  hresp <= 1'b0;
  hrdata <= mem[raddr];
  raddr <= raddr + 1'b1;
end
// wrap 16
8'b0000_0010: begin
  hreadyout <= 1'b1;
  hresp <= 1'b0;
  if(raddr < (haddr + 4'd15)) begin
```

```
              hrdata <= mem[raddr];
              raddr <= raddr + 1'b1;
            end
            else begin
              hrdata <= mem[raddr];
              raddr <= haddr;
            end
          end
          // incre 16
          8'b0000_0001: begin
            hreadyout <= 1'b1;
            hresp <= 1'b0;
            hrdata <= mem[raddr];
            raddr <= raddr + 1'b1;
          end
          // default
          default: begin
            hreadyout <= 1'b1;
            hresp <= 1'b0;
          end
        endcase
      end
      default: begin
        hreadyout <= 1'b0;
        hresp <= 1'b0;
        hrdata <= hrdata;
        waddr <= waddr;
        raddr <= raddr;
      end
    endcase
  end
end
```

arbiter module code

```
module arbiter(

input hclk,
input hresetn,
input hreq_1,
input hreq_2,
input [1:0] sel_1,
input [1:0]sel_2,
input hready,
input hready_out,

output reg hgrant_1,
output reg hgrant_2,
output reg [1:0] sel


);

//----------------------------
parameter IDLE = 2'b00;
parameter GRANT1 = 2'b01;
parameter GRANT2 = 2'b10;

reg [1:0]  state;
```

```
    reg [1:0]  next_state;

    wire busy;

    assign busy = hready | hready_out;

always @(posedge hclk, negedge hresetn) begin
  if(!hresetn) begin
    state <= IDLE;
  end
  else begin
    state <= next_state;
  end
end

always@(negedge hready_out, hreq_1, hreq_2)begin
case(state)
    IDLE:begin
        if(hreq_1 == 1) // normal request only from master 1
begin
 next_state <= GRANT1;
end
else if(hreq_2 == 1) // normal request only from master 2
begin
 next_state <= GRANT2;
end
else if((hreq_1 == 0) && (hreq_2==0))
begin
 next_state <= IDLE;
end
end
GRANT1:begin
   if((hreq_1 == 0) && (hreq_2==0))
begin
next_state <= IDLE;
end
   else if((hreq_1 == 0) && (hreq_2==1))
        begin
next_state <= GRANT2;
end
       else //hreq_1 ==1 && hreq_2 == 1 OR hreq_1 == 1 && hreq_2 ==0
            begin
        next_state <= GRANT1;
            end
        end
    GRANT2:begin
        if((hreq_1 == 0) && (hreq_2==0))
begin
next_state <= IDLE;
end
        else if((hreq_2 == 0) && (hreq_1 ==1))
        begin
                next_state <= GRANT1;
        end

        else if((hreq_2 == 1) && (hreq_1 ==0))
        begin
                next_state <= GRANT2;
```

```
        end
        end
    default:begin
            next_state <= IDLE;
    end
    endcase
end

always @(posedge hclk, negedge hresetn) begin
if(!hresetn) begin
        hgrant_1 <= 0;
        hgrant_2 <= 0;
        sel <= 2'b00;
        state <= IDLE;
        next_state <= IDLE;
    end
else begin
    case(next_state)
IDLE:begin
hgrant_1 <= 0;
hgrant_2 <= 0;
sel <= 2'b00;
end

GRANT1:begin // access given to Master 1

hgrant_1 <= 1;
hgrant_2 <= 0;
sel <= sel_1; // goes to the address and write muxes

//state <= (busy == 1'b1)?GRANT1:IDLE; // (some condition)?if_TRUE:if_FALSE Ternery operator
end
GRANT2:begin

hgrant_1 <= 0;
hgrant_2 <= 1;
sel <= sel_2;


//state <= (busy == 1'b1)?GRANT2:IDLE;
end
endcase
end
end

endmodule
```

decoder module code

```
    module decoder(
  input [1:0] sel,
  output reg hsel_1,
  output reg hsel_2,
  output reg hsel_3
);

always @(*) begin
  case(sel)
```

```
    2'b01: begin
      hsel_1 = 1'b1;
      hsel_2 = 1'b0;
      hsel_3 = 1'b0;
    end
    2'b10: begin
      hsel_1 = 1'b0;
      hsel_2 = 1'b1;
      hsel_3 = 1'b0;
    end
    2'b11: begin
      hsel_1 = 1'b0;
      hsel_2 = 1'b0;
      hsel_3 = 1'b1;
    end
    default: begin
      hsel_1 = 1'b0;
      hsel_2 = 1'b0;
      hsel_3 = 1'b0;
    end
  endcase
end


endmodule
```

multiplexor module code

```
    module multiplexor(
  input [31:0] hrdata_1,
  input [31:0] hrdata_2,
  input [31:0] hrdata_3,
  input hreadyout_1,
  input hreadyout_2,
  input hreadyout_3,
  input [1:0] sel,
  output reg [31:0] hrdata,
  output reg hreadyout
);

always@(*) begin
  case(sel)
    2'b01: begin
      hrdata = hrdata_1;
      hreadyout = hreadyout_1;
    end
    2'b10: begin
      hrdata = hrdata_2;
      hreadyout = hreadyout_2;
    end
    2'b11: begin
      hrdata = hrdata_3;
      hreadyout = hreadyout_3;
    end
    default: begin
      hrdata = 32'h0000_0000;
      hreadyout = 1'b0;
    end
```

```
  endcase
end

endmodule
```

write mux module code

```
    module write_mux(
  input [31:0] haddr_1,
  input [31:0] haddr_2,
  input [31:0] hwdata_1,
  input [31:0] hwdata_2,
  input hready_1,
  input hready_2,
  input hgrant_1,
  input hgrant_2,
  input hwrite_1,
  input hwrite_2,
  output reg [31:0] haddr,
  output reg [31:0] hwdata,
  output reg hwrite,
  output reg hready
);

always @(*) begin
    if(hgrant_1 == 1)
        begin
            haddr = haddr_1;
            hwdata = hwdata_1;
            hready = hready_1;
            hwrite = hwrite_1;
        end

else if(hgrant_2 == 1)
       begin
            haddr = haddr_2;
            hwdata = hwdata_2;
            hready = hready_2;
            hwrite = hwrite_2;
        end
    else
        begin
            haddr = 32'h0000_0000;
            hwdata = 32'h0000_0000;
            hready = 1'b0;
            hwrite = 1'b0;
        end
end

endmodule
```