HO CHI MINH CITY UNIVERSITY OF TECHNOLOGY AND EDUCATION

FACULTY OF INTERNATIONAL EDUCATION

INTEGRATED CIRCUITS AND SYSTEMS DESIGN

ICSD338164E_23_1_01FIE

Topic: RAM ARBITER

Members:

1. Đỗ Mạnh Dũng	**21119301**
2. Trịnh Lê Minh Hiếu	**21119011**
3. Phạm Trần Minh	**21119023**
4. Đặng Trung Nghĩa	**21119313**

Instructors:

Assoc. Prof. Võ Minh Huân

Ho Chi Minh City, 12/2023

Table of Contents

PART A: INTRODUCTION

1.1. Problem Statement:

In today's world, with the prevalence of electronic systems and computers, managing memory access becomes a challenge for designers. In an asynchronous environment, where there is no common clock to synchronize events, ensuring efficient and fair memory access becomes a complex issue.

One of the significant challenges arises from concurrently managing multiple memory access requests from different components. When multiple parts of the system simultaneously request access to the memory, deciding which request takes priority becomes crucial. This raises the issue of designing an effective Arbiter, a component that determines which component is allowed memory access and when.

1.2. Problem Resolution:

To address these challenges, the project "Designing RAM Arbiter" focuses on researching and developing an intelligent and flexible Arbiter. The Arbiter will ensure that every memory access request is handled fairly, stably, and efficiently.

Through the research process, we will explore methods and algorithms to make informed decisions when multiple requests coincide. The goal is to build an Arbiter with flexible responsiveness, minimizing conflicts and ensuring that every system component has an opportunity for efficient memory access.

This project is not only technically oriented but also an opportunity to contribute to the development of asynchronous systems and memory management. It provides a practical solution to one of the most critical issues in this field.

PART B: MAIN CONTENT

1. TOP LEVEL BLOCK DIAGRAM

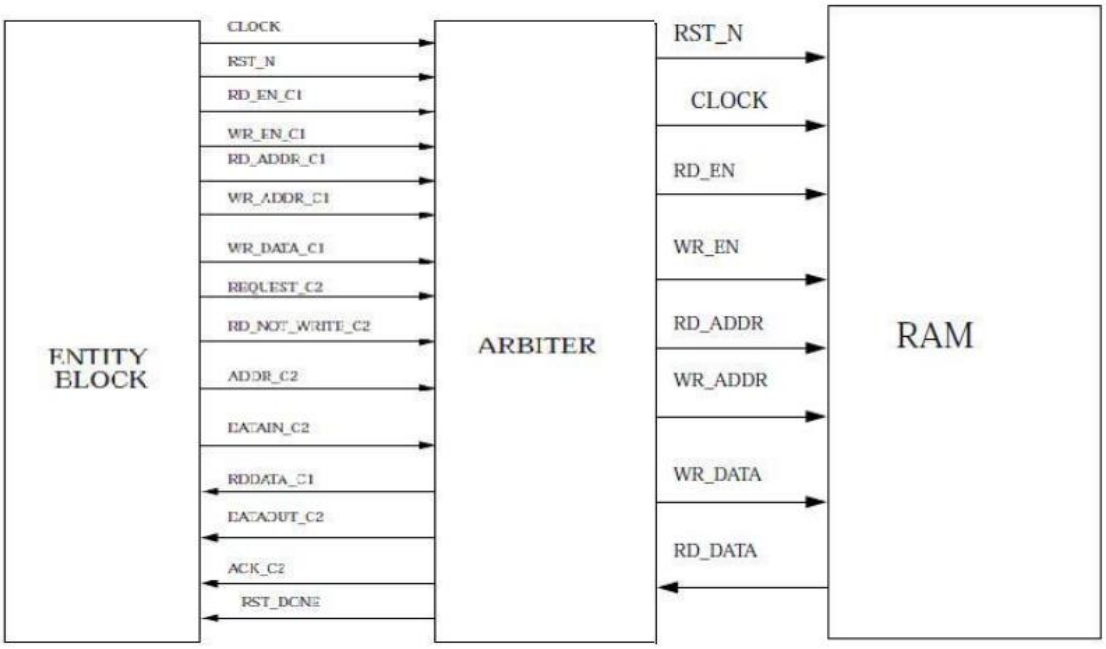
Top Level View Block Diagram Of Ram_Arbiter

PORT-NAME	DIRECTION	FUNCTIONALITY
RD_EN_C1	IN	Active high signal for read operation from RAM by client1
WR_EN_C1	IN	Active high signal for write operation into RAM by client1
RD_ADDR_C1	IN	RAM address(bus signal) from where client1 wants to read
WR_ADDR_C1	IN	RAM address(bus signal) where client1 wants to write
WR_DATA_C1	IN	The data(bus signal),client1 wants to write in WR_ADDR
REQUEST_C2	IN	Active high signal by client2 to access RAM
RD_NOT_WRITE_C2	IN	Active high for reading, Active low for writing by client2
ADDR_C2	IN	RAM address(bus signal) either for read or write by client2
DATAIN_C2	IN	The data (bus signal) if client2 wants to write in ADDR_C2
RD_DATA_C1	OUT	Output data(bus signal) got by client1 from RAM
DATA_OUT_C2	OUT	Output data(bus signal) got by client2 from RAM
ACK_C2	OUT	Active high signal to indicate that request by client2 is done
CLOCK	IN	Positive edge input for valid operation (global signal)
RST_N	IN	Active low signal to clear all registers (global signal)
RST_DONE	OUT	Active low signal to indicate that RAM is being initialized

Port List and their Functionality

This is the top level block diagram which can interface with two clients (client1 and client2) with their respective input and output ports. But client1 has a higher priority than client2. So client1 can have the access of the RAM any time it wants. But if client2 wants to access the RAM then it must send a request first and according to the following condition Client2 can get the access of the RAM:

1) If client1 is only writing then Client2 can get the access of the RAM for reading only
2) If client1 is only reading then client2 can get the access of the RAM for writing only
3) If client1 is reading and also writing then client2 cannot get any access of the RAM
4) If client1 is not doing anything then Client2 can get the access of the RAM either for reading or writing.



Internal Architecture of Ram_Arbiter

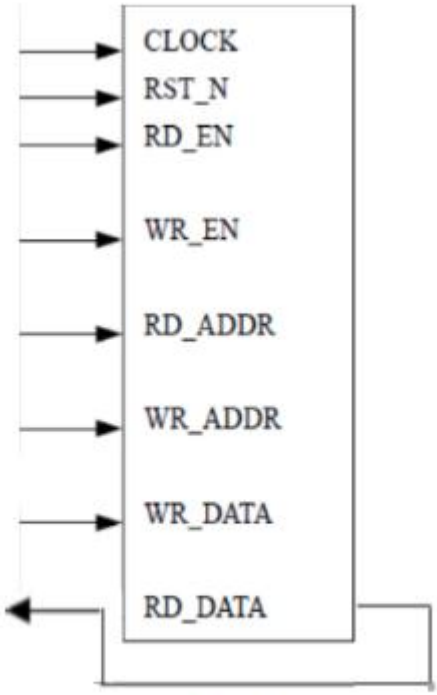
2. RAM

RAM stands for "Random Access Memory." It is a type of computer memory used to temporarily store data and provide quick access to the processor. Data in RAM can be read and written quickly, and it retains data only as long as the computer is powered on. When the computer is turned off or restarted, the data in RAM is lost.

RAM is primarily used to store programs and data that the processor is currently or will soon be using, and it is designed to provide fast access to this information. In modern applications and operating systems, having a larger RAM capacity often contributes to smoother computer performance and faster processing.

2.1. BLOCK DIAGRAM OF RAM

In Random Access Memory (RAM), the memory cells can be accessed for information transfer from any desired random location. That is the process of locating a word in memory is the same and requires an equal amount of time no matter where the cells are located physically in memory: thus the name “Random Access”. Communication between memory and its environment is achieved through data input and output lines, address selection lines, and control lines that specify the direction of transfer. A block diagram of a RAM unit is shown below figure:



RAM
COMPONENET

A random-access memory can perform two main operations: write and read. The write signal indicates a transfer-in operation, while the read signal indicates a transfer-out operation. Upon receiving one of these control signals, the internal circuits within the memory execute the desired function. The steps for transferring a new word into memory are as follows:

- Apply the binary address of the desired word to the address lines.
- Input the data bits to be stored in memory into the data input lines.
- Activate the write input.

Subsequently, the memory unit takes the bits currently available in the input data lines and stores them in the word specified by the address lines. The steps for transferring a stored word out of memory are as follows:

- Apply the binary address of the desired word to the address lines.
- Activate the read input.

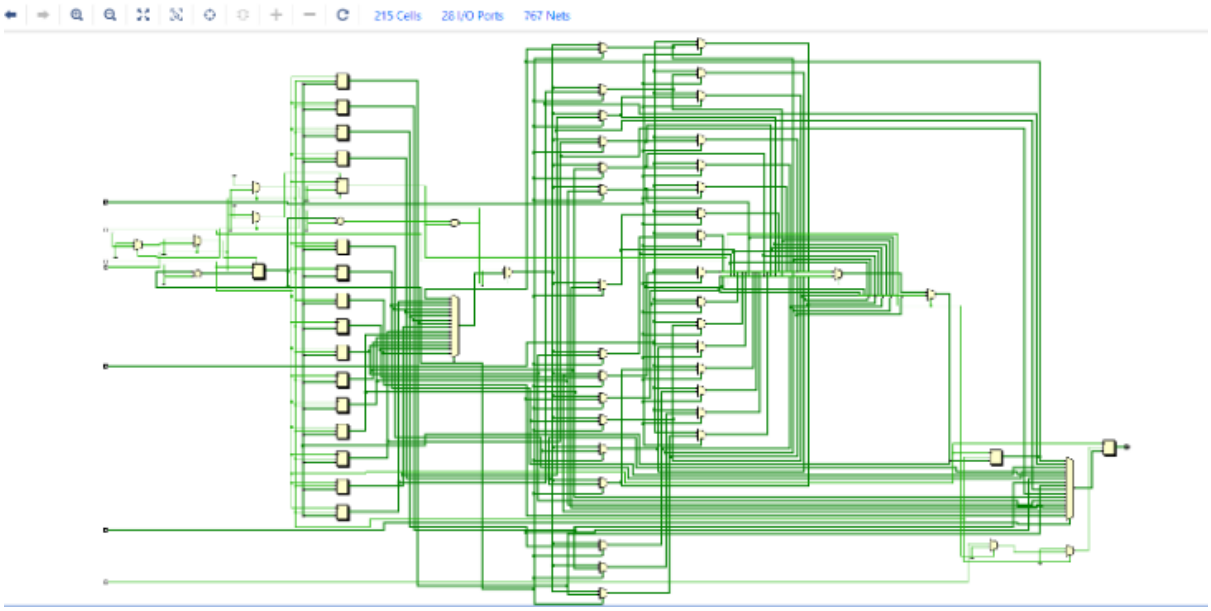
The memory unit will then take the bits from the word that has been selected by the address and apply them to the output data lines. The content of the selected word does not change after reading.

2.2. CODING

```
            //Declare Module and Parameters.

module RAM #(parameter G_ADDR_WIDTH = 4, parameter G_DATA_WIDTH = 8) ;     //Declare a Verilog module with the name "RAM."

  (input CLOCK, RST_N, RD_EN, WR_EN,

   input [G_ADDR_WIDTH-1:0] RD_ADDR, WR_ADDR,

   input [G_DATA_WIDTH-1:0] WR_DATA,

   output reg [G_DATA_WIDTH-1:0] RD_DATA);                //Define module parameters (G_ADDR_WIDTH and G_DATA_WIDTH) with default values of 4 and 8.

            //Declare Constants and Variables.

  localparam RAM_DEPTH = 2**G_ADDR_WIDTH;  // Declare a local constant (RAM_DEPTH) with a value equal to 2 raised to the power of G_ADDR_WIDTH.

  reg [G_DATA_WIDTH-1:0] MEMORY [0:RAM_DEPTH-1];  // Declare an array named MEMORY as a two-dimensional array of registers with a length of RAM_DEPTH and a width of G_DATA_WIDTH.

  integer count = 0;

  reg reset_done;



  always @(negedge CLOCK) begin

            // Reset Process

    if (!RST_N)

      reset_done <= 1'b1;

    else begin   // If the signal RST_N (inverse of reset) is 0, set reset_done to 1; otherwise, set reset_done to 0.

              // Initialization Process

     if ((count < (2**G_ADDR_WIDTH)) && (reset_done == 1'b1)) begin // If count is less than 2^G_ADDR_WIDTH and reset_done is 1,

MEMORY[count] <= 0;

       count <= count + 1; // initialize MEMORY[count] to 0 and increment count by 1.

     end

     else begin
```

```
        count <= 0;

        reset_done <= 1'b0;  // Otherwise, set count to 0 and reset_done to 0.

      end

            // Main Process (Read and Write)

 if (!reset_done) begin // Check if the module is not in the reset state (reset_done is 1). If not in the reset state (reset_done is 1), the main process will be executed.

            // Write Process

 if (WR_EN) // If WR_EN is 1, indicating a write request, the write process is executed.          MEMO  // Write data from WR_DATA to the position in memory (MEMORY) corresponding to the address WR_ADDR.

          if (RD_EN)

             RD_DATA <= MEMORY[RD_ADDR]; // Assign the value of the memory cell at address RD_ADDR to the output signal RD_DATA.

 end

    end

  end

endmodule
```

2.3. SCHEMATIC OF RAM



Schematic of RAM

2.4. TESTCASES FOR RAM

Case 1: RAM write operation
Case 2: RAM read operation
Case 3: RAM read & write operation

2.5. VERILOG TESTBENCH

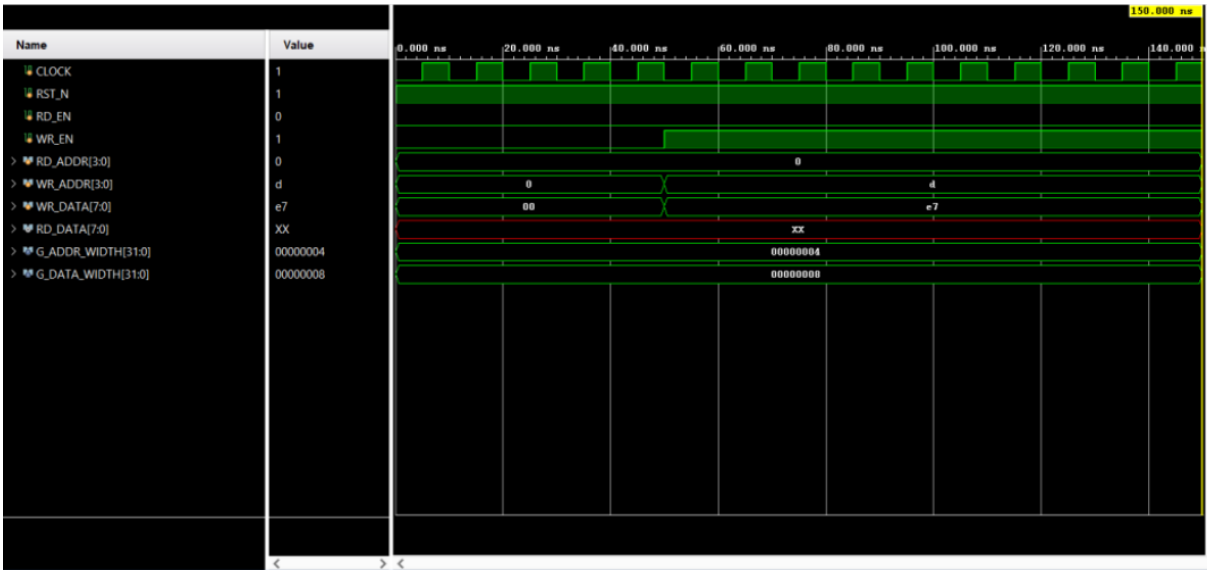
2.5.1. TESTBENCH CODE

```
`timescale 1ns / 1ps

    module tb_RAM;

// Parameters

    parameter G_ADDR_WIDTH = 4;

    parameter G_DATA_WIDTH = 8;

// Inputs

    reg CLOCK;

    reg RST_N;

    reg RD_EN;

    reg WR_EN;

    reg [G_ADDR_WIDTH-1:0] RD_ADDR;

    reg [G_ADDR_WIDTH-1:0] WR_ADDR;

    reg [G_DATA_WIDTH-1:0] WR_DATA;

// Outputs

   wire [G_DATA_WIDTH-1:0] RD_DATA;

// Instantiate the RAM module

  RAM #(G_ADDR_WIDTH, G_DATA_WIDTH) uut (

  .CLOCK(CLOCK),

  .RST_N(RST_N),

  .RD_EN(RD_EN),

  .WR_EN(WR_EN),
```

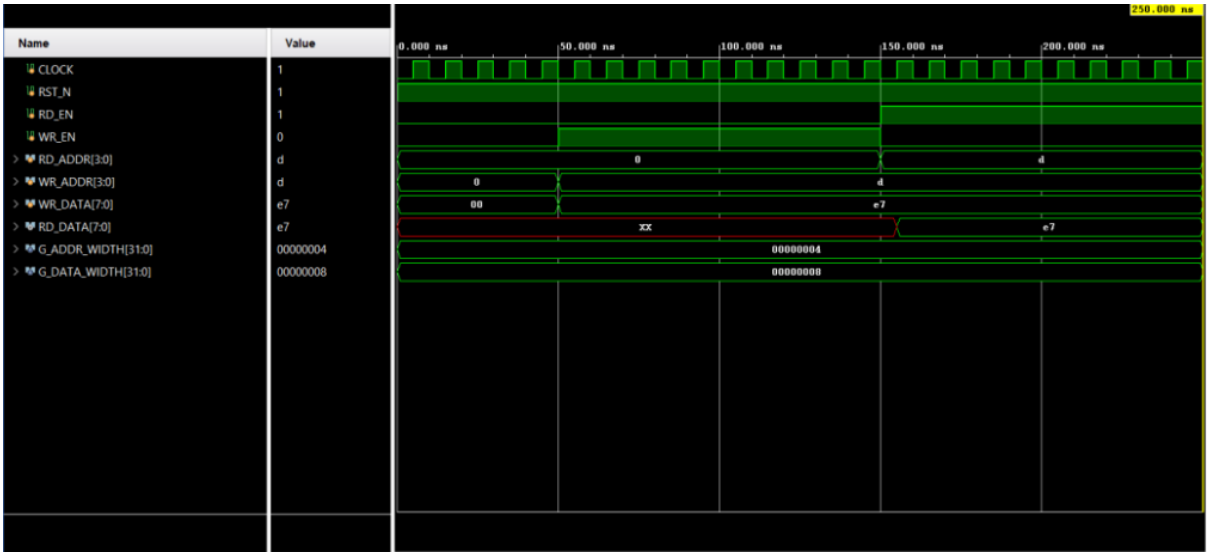
```
  .RD_ADDR(RD_ADDR),
  .WR_ADDR(WR_ADDR),
  .WR_DATA(WR_DATA),
  .RD_DATA(RD_DATA)
 );
 // Clock generation process
always #5 CLOCK = ~CLOCK;
 // Initial stimulus
 initial begin
  // Initialize inputs
  RST_N = 1;
  CLOCK = 0;
  RD_EN = 0;
  WR_EN = 0;
  RD_ADDR = 0;
  WR_ADDR = 0;
  WR_DATA = 0;
  // Apply reset for a short duration
  #50 RST_N = 1;
   // Test Case 1: RAM Write Operation
  WR_EN = 1;
  RD_EN = 0;
  WR_ADDR = 4'b1101;
  WR_DATA = 8'b11100111;
  #100;
   // Test Case 2: RAM Read Operation
  WR_EN = 0;
  RD_EN = 1;
```

```
    RD_ADDR = 4'b1101;
    #100;
     // Test Case 3: RAM Read & Write Operation
    WR_EN = 1;
    RD_EN = 1;
    RD_ADDR = 4'b1101; //1101 = d is address
    WR_ADDR = 4'b1011; //1011=b is address
    WR_DATA = 8'b10111001; // 10111001 = b9 is data
    #100;
    // Finish simulation
    $finish;
  end
endmodule
```

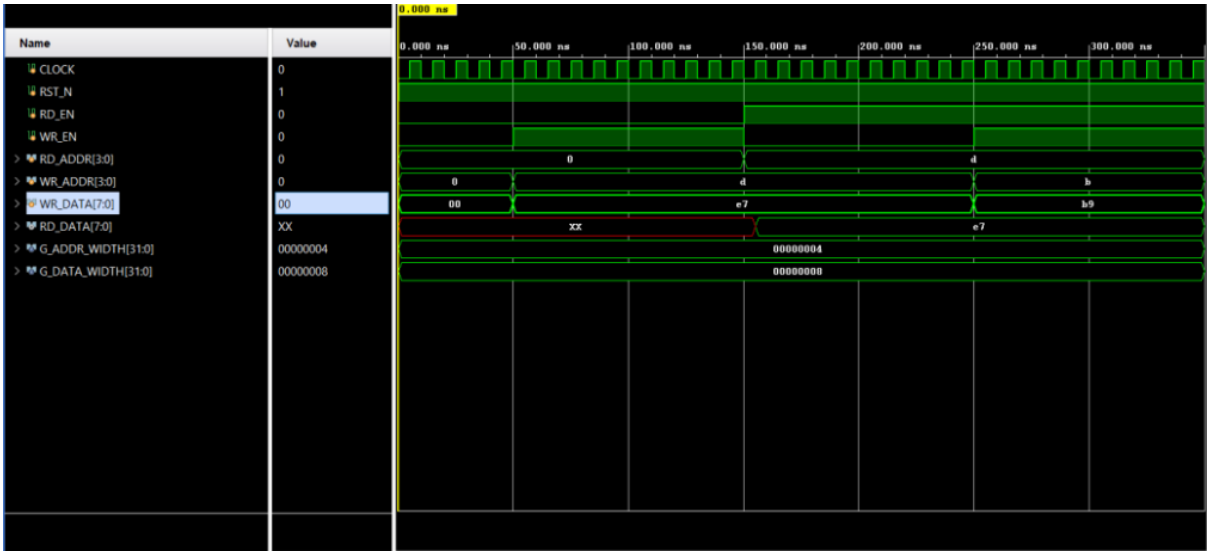
Waveforms:



RAM WRITE OPERATION (CASE 1)



RAM READ OPERATION (CASE 2)



RAM READ AND WRITE OPERATION (CASE 3)

2.5.2. ANALYSIS OF THE TEST CASE

- **Case 1: Write Operation**

In this case, within the first RAM cycle, it is initialized and the write operation is performed at the address location “1101 (binary) = d (hex)”. The data written is “11100111 = e7 (hex)”. The RAM starts working when the RST_N pin is enabled.

- **Case 2: Read Operation**

Similar to the previous condition, when the RST_N pin is enabled within the first RAM cycle, data is read from the address location “1101 (binary) = d (hex)”.

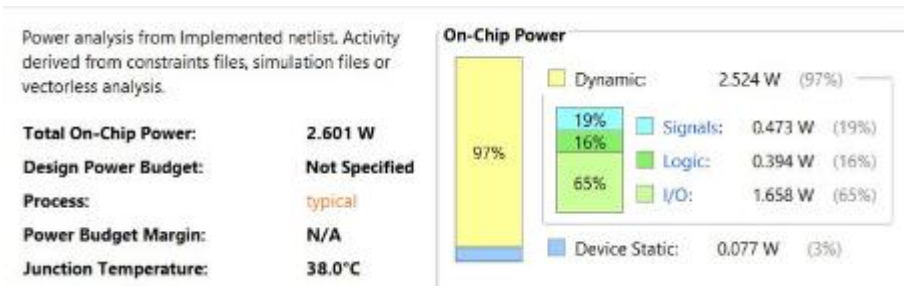
- **Case 3: Read and Write Operation**

In this instance, a HIGH setting on the reset pin activates the RAM. The data "11100111 (binary) = e7 (hex)" is written to the address "1101 (binary) = d (hex)" in the first RAM cycle. We aim to demonstrate that the RAM design could handle both read and write operations in a single RAM cycle in the second RAM cycle. While data "10111001 (binary) = b9 (hex)” was being written at the address location "1011 (binary) = b (hex)," data was being read from location "1101".

2.5.3. POWER & TIMING

Power:

- Dynamic Power: It is the energy consumed during the transition of states from 0 to 1 or vice versa. This occurs when flip-flops change states or when transistors switch states during computation.
- Static Power: It is the energy consumed even when the circuit is not active (no dynamic transitions). The main cause is due to the leakage current of transistors and other energy loss.



POWER OF RAM

Setup Time and Hold Time:

- Setup time is the time before a synchronous signal changes that the input data needs to be stable.
- Hold time is the time after a synchronous signal changes that the input data needs to be held stable.

SET UP TIME

Name	Slack	Levels	Routes	High Fanout	From	To	Total Delay	Logic Delay	Net Delay	Logic %	Net %	Requirement	Source Clock
Unconstrained Paths (1)													
(none) (10)													
Path 11	∞	4	3	43	RD_ADDR[0]	RD_DATA_reg[0]/D	7.127	1.549	5.578	21.7	78.3	∞	input port clock
Path 12	∞	4	3	43	RD_ADDR[0]	RD_DATA_reg[4]/D	7.054	1.549	5.505	22.0	78.0	∞	input port clock
Path 13	∞	5	4	22	RD_ADDR[2]	RD_DATA_reg[7]/D	6.821	1.673	5.149	24.5	75.5	∞	input port clock
Path 14	∞	5	4	22	RD_ADDR[2]	RD_DATA_reg[1]/D	6.727	1.673	5.054	24.9	75.1	∞	input port clock
Path 15	∞	4	3	43	RD_ADDR[0]	RD_DATA_reg[3]/D	6.676	1.549	5.127	23.2	76.8	∞	input port clock
Path 16	∞	4	3	43	RD_ADDR[0]	RD_DATA_reg[2]/D	6.606	1.549	5.057	23.4	76.6	∞	input port clock
Path 17	∞	4	3	43	RD_ADDR[0]	RD_DATA_reg[5]/D	6.519	1.549	4.970	23.8	76.2	∞	input port clock
Path 18	∞	5	4	22	RD_ADDR[2]	RD_DATA_reg[6]/D	6.484	1.673	4.811	25.8	74.2	∞	input port clock
Path 19	∞	8	3	18	count_reg[7]/C	memory_reg[15][0]/D	6.404	2.030	4.374	31.7	68.3	∞	
Path 20	∞	4	3	20	WR_ADDR[1]	memory_reg[6][0]/D	6.378	1.528	4.850	24.0	76.0	∞	input port clock

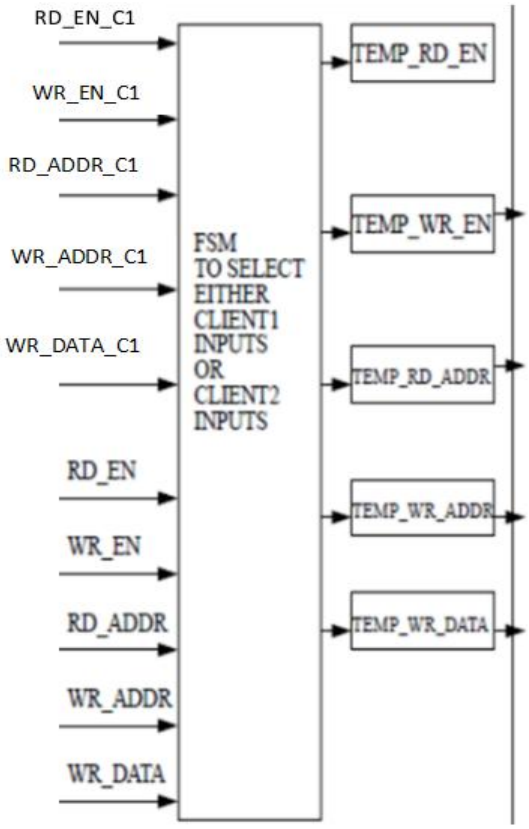
HOLD TIME

Name	Slack	Levels	Routes	High Fanout	From	To	Total Delay	Logic Delay	Net Delay	Logic %	Net %	Requirement
Unconstrained Paths (1)												
(none) (10)												
Path 1	∞	2	1	2	memory_reg[7][6]/C	RD_DATA_reg[6]/D	0.248	0.186	0.062	75.0	25.0	-∞
Path 2	∞	2	1	2	memory_reg[14][6]/C	memory_reg[14][6]/D	0.300	0.186	0.114	62.0	38.0	-∞
Path 3	∞	2	1	2	memory_reg[2][0]/C	memory_reg[2][0]/D	0.308	0.186	0.122	60.4	39.6	-∞
Path 4	∞	2	1	2	memory_reg[3][5]/C	memory_reg[3][5]/D	0.335	0.209	0.126	62.4	37.6	-∞
Path 5	∞	2	1	2	memory_reg[7][7]/C	RD_DATA_reg[7]/D	0.337	0.186	0.151	55.2	44.8	-∞
Path 6	∞	2	1	2	memory_reg[13][0]/C	memory_reg[13][0]/D	0.353	0.186	0.167	52.7	47.3	-∞
Path 7	∞	2	1	2	memory_reg[1][0]/C	memory_reg[1][0]/D	0.353	0.186	0.167	52.7	47.3	-∞
Path 8	∞	2	1	2	memory_reg[1][4]/C	memory_reg[1][4]/D	0.353	0.186	0.167	52.7	47.3	-∞
Path 9	∞	2	1	2	memory_reg[4][3]/C	memory_reg[4][3]/D	0.353	0.186	0.167	52.7	47.3	-∞
Path 10	∞	2	1	2	memory_reg[0][0]/C	memory_reg[0][0]/D	0.353	0.186	0.167	52.7	47.3	-∞

3. ARBITER

3.1. FSM

3.1.1. BLOCK DIAGRAM



3.1.2. FSM DESIGN BLOCK

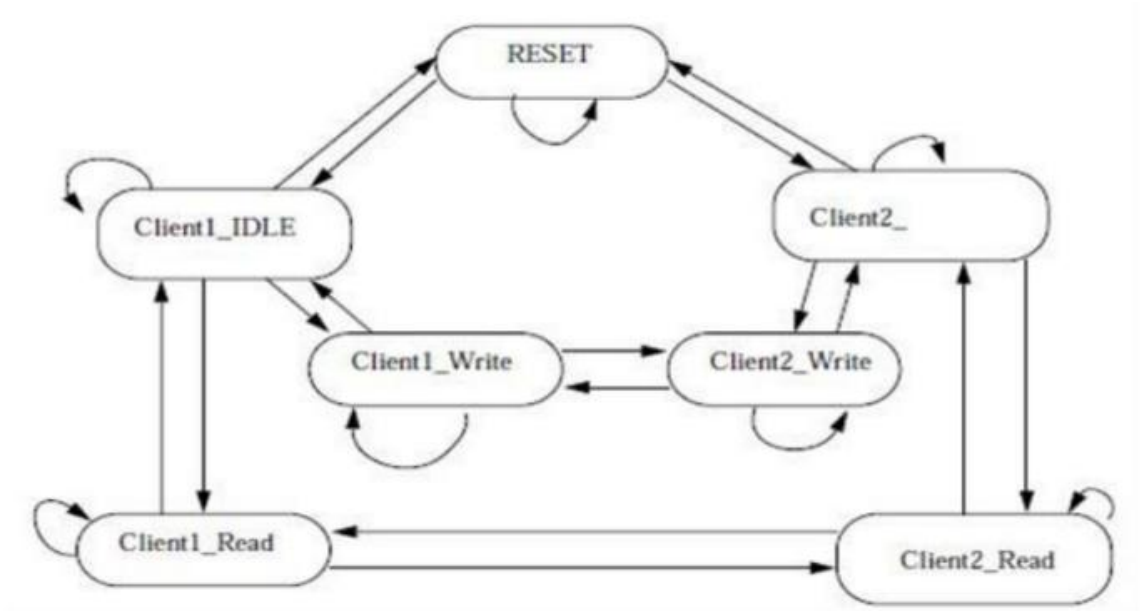
This FSM determines which client gets access to the specified RAM depending on the following conditions:

1) If Client1 is writing into RAM but not reading, then Client2 can read but can’t write.

2) If Client1 is reading from RAM but not writing, then Client2 can write but can’t read.

3) If Client2 is reading and Client1 wants to read then Client1 gets access to RAM.

4) If Client2 is writing and Client1 wants to write then Client1 gets access to RAM.



On applying RST_N=0 FSM will be in RESET state and after RST_N=1 it will take RAM_DEPTH cycles to reach in IDLE state and any input can be accepted after reaching IDLE state.

Transition of states depending upon different input conditions from the clients are given below:

1. IDLE----->Client1_Read==> RD_EN_C1=1.
2. Client1_Read----->IDLE==> RD_EN_C1=0.
3. IDLE---->Client1_Write==> WR_EN_C1=1.
4. Client1_Write---->IDLE==> WR_EN_C1=0.
5. IDLE--->Client2_Read==> RD_EN_C1=0 and REQUEST_C2=1 and D_NOT_WRITE_C2=1.
6. Client2_Read--->IDLE==>REQUEST_C2=0 or REQUEST_C2=1 and D_NOT_WRITE_C2=0.
7. IDLE--->Client2_Write==>WR_EN_C1=0 and REQUEST_C2=1 and D_NOT_WRITE_C2=0.
8. Client2_Write-->IDLE==>REQUEST_C2=0 or REQUEST_C2=1 and RD_NOT_WRITE_C2=1.
9. Client2_Read---->Client1_Read===>RD_EN_C1=1.
10. Client1_Read----->Client2_Read===>RD_EN_C1=0 and REQUEST_C2=1 and RD_NOT_WRITE_C2=1.
11. Client2_Write----->Client1_Write==>WR_EN_C1=1.
12. Client1_Write----->Client2_Write==>WR_EN_C1=0 and REQUEST_C2=1 and RD_NOT_WRITE_C2=0.
13. Client1_Read--->Client1_Read==>RD_EN_C1=1.
14. Client1_Write--->Client1_Write==>WR_EN_C1=1.
15. Client2_Read--->Client2_Read===>REQUEST_C2=1 and RD_NOT_WRITE_C2=1 and RD_EN_C1=0.
16. Client2_Write---->Client2_Write===>REQUEST_C2=1 and RD_NOT_WRITE_C2=0 and WR_EN_C1=0.

17. IDLE---->IDLE===>RD_EN_C1=0 and WR_EN_C1=0 and REQUEST_C2=0.
18. RESET--->RESET===>if RST_N=0 holds or RST_N=0 for once and RST_DONE=0
19. RESET--->IDLE===>RST_N=1 and RST_DONE=1.
20. IDLE--->RESET==>RST_N=0.

3.1.3. CODING

```
module FSM #(

    parameter G_ADDR_WIDTH = 4,

    parameter G_DATA_WIDTH = 8,

    parameter G_REGISTERED_DATA = 0)

(

    input RST_N, CLOCK,

    output RST_DONE,

    input RD_EN_C1, WR_EN_C1,    // Read enable, write enable

    input [G_ADDR_WIDTH-1 : 0] RDADDR_C1, WRADDR_C1,    // Read address, write address

    input [G_DATA_WIDTH-1 : 0] WRDATA_C1,

    input [G_DATA_WIDTH-1 : 0] DATAIN_C2,

    input REQUEST_C2, RD_NOT_WRITE_C2,    // Request for memory access and read-write bar

    input [G_ADDR_WIDTH-1 : 0] ADDR_C2,    // This address is for both read and write one at a time

    output [G_DATA_WIDTH-1 : 0] RDDATA_C1,    // Data out for client1/user1

    output [G_DATA_WIDTH-1 : 0] DATAOUT_C2,    // Data out for client2/user2

    output ACK_C2,        // Acknowledgement in the case of client2,

                    // since it will not be given access immediately

                    // since priority is more of client1

    output RD_EN, WR_EN,    // Read and write enable signals for the arbiter
```

```
        output [G_ADDR_WIDTH-1 : 0] WR_ADDR, RD_ADDR,    // Write and read address signals

        output [G_DATA_WIDTH-1 : 0] WR_DATA,    // Write data signal

        input [G_DATA_WIDTH-1 : 0] RD_DATA    // Read data signal

    );

    /// Temporary registers to store data during state transitions

    reg [G_DATA_WIDTH-1 : 0] TEMP_RD_DATA, TEMP_RD_DATA1, TEMP_RD_DATA2;

    reg TEMP_RD_EN, TEMP_WR_EN;

    reg [G_ADDR_WIDTH-1 : 0] TEMP_WR_ADDR, TEMP_RD_ADDR;

    reg [G_DATA_WIDTH-1 : 0] TEMP_WR_DATA;

    // Enumerated values for FSM states

    localparam [2:0] reset=3'b000, idle=3'b001, client1_read=3'b010, client2_read=3'b011, client1_write=3'b100, client2_write=3'b101;

    // Registers to store current and next states for clients

    reg [2:0] pr_client_read, pr_client_write, nx_client_read, nx_client_write;

    // Acknowledgment signals for clients

    reg TEMP_ACK = 0, TEMP_ACK1, TEMP_ACK2, TEMP_WR=0;

    wire TEMP_WR1;

    // Signal for registered data

    wire REGISTERED_DATA = 0;

    reg RESET_DONE_REG;

    // Counter for reset completion

    integer COUNT = 0;

    // Signals for address clash detection

    reg ADDR_CLASHI=0, ADDR_CLASH=0;
```

```
    // FSM
    always @(posedge CLOCK, negedge RST_N) begin
        if (!RST_N) begin
            // Initialization during reset
            pr_client_read <= reset; //+ Initialize the client read state during the reset state
            pr_client_write <= reset; //+ Initialize the client write state during the reset state
        end
        else begin
            // Update current states
            pr_client_read <= nx_client_read; //+ Update the client read state
            pr_client_write <= nx_client_write; //+ Update the client write state
        end
    end
    // Generate block for handling registered data
    generate
        if (G_REGISTERED_DATA) begin : g1
            assign REGISTERED_DATA = G_REGISTERED_DATA; //+ Assign the value of the REGISTERED_DATA parameter
        end
    endgenerate
    // FSM state transition logic
    always @(pr_client_read, pr_client_write, CLOCK) begin
        if (RST_N & CLOCK) begin
            // Reset state initialization
            if (nx_client_read==reset && nx_client_write==reset) begin
```

```
            if (COUNT < (2**G_ADDR_WIDTH)) begin
                // Reset not done yet
                RESET_DONE_REG <= 1'b0;
                COUNT <= COUNT + 1;
            end
            else begin
                // Reset done
                nx_client_read = idle;
                nx_client_write = idle;
                RESET_DONE_REG = 1'b1;
                COUNT = 0;
            end
        end
    end
    else if(!RST_N) begin
        // Reset condition
        nx_client_read = reset;
        nx_client_write = reset;
    end
    // State transitions based on current state
    // Handling idle state
    if (pr_client_read == idle) begin
        if (!RD_EN_C1) begin
            if (!REQUEST_C2)
```

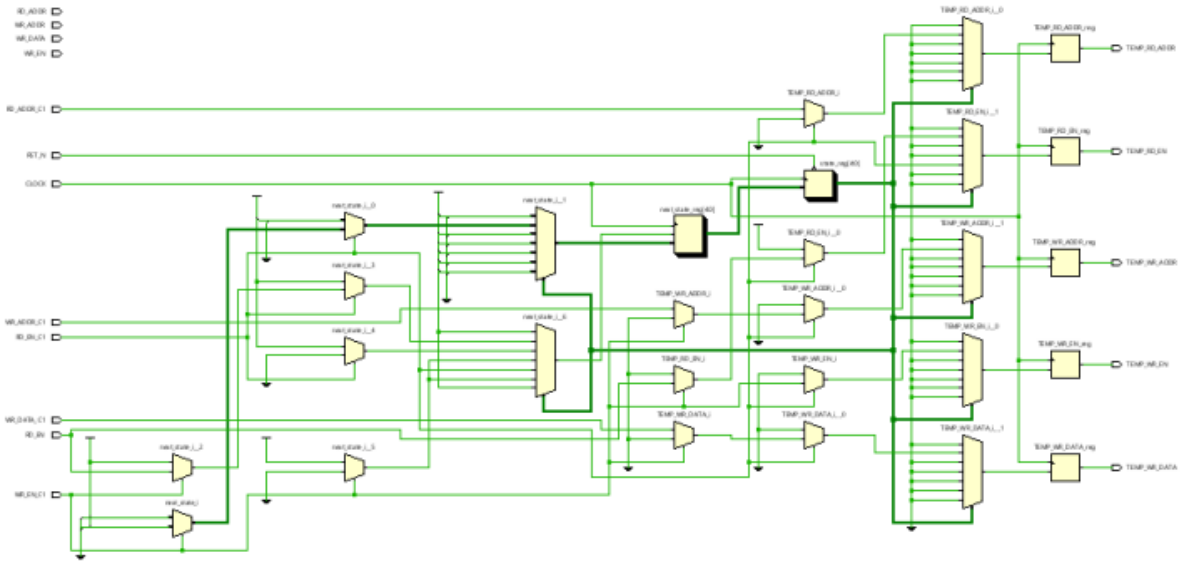
```
                nx_client_read = idle; //+ In idle state and no read requests from client 1 or 2, stay in idle state

            else if (RD_NOT_WRITE_C2)

                nx_client_read = client2_read; //+ If there is a request from client 2 and it's a read request, transition to client 2 read state

            else if (!RD_NOT_WRITE_C2)

                nx_client_write = client2_write; //+ If there is a request from client 2 and it's a write request, transition to client 2 write state

        end

        else

            nx_client_read = client1_read; //+ If there is a read request from client 1, transition to client 1 read state

    end

    if (pr_client_write == idle) begin

        if (!WR_EN_C1) begin

            if (!REQUEST_C2)

                nx_client_write = idle; //+ In idle state and no write requests from client 1 or 2, stay in idle state

            else if (!RD_NOT_WRITE_C2)

                nx_client_write = client2_write; //+ If there is a request from client 2 and it's a write request, transition to client 2 write state

            else if (RD_NOT_WRITE_C2)

                nx_client_read = client2_read; //+ If there is a request from client 2 and it's a read request, transition to client 2 read state

        end

        else

            nx_client_write = client1_write; //+ If there is a write request from client 1, transition to client 1 write state
```

```
        end

        // Handling client1 read state

        if (pr_client_read == client1_read) begin

            if (RD_EN_C1)

                nx_client_read = client1_read; //+ In client 1 read state and there is a read request from client 1, stay in client 1 read state

            else begin

                if (!REQUEST_C2)

                    nx_client_read = idle; //+ In client 1 read state and no requests from client 2, transition to idle state

                else if (RD_NOT_WRITE_C2)

                    nx_client_read = client2_read; //+ In client 1 read state and there is a read request from client 2, transition to client 2 read state

                else if (!RD_NOT_WRITE_C2)

                    nx_client_read = idle; //+ In client 1 read state and there is a write request from client 2, transition to idle state

            end

        end

        // Handling client1 write state

        if (pr_client_write == client1_write) begin

            if (WR_EN_C1)

                nx_client_write = client1_write; //+ In client 1 write state and there is a write request from client 1, stay in client 1 write state

            else begin

                if (!REQUEST_C2)

                    nx_client_write = idle; //+ In client 1 write state and no requests from client 2, transition to idle state

                else if (!RD_NOT_WRITE_C2)
```

```
                nx_client_write = client2_write; //+ In client 1 write state and there is a write request from client 2, transition to client 2 write state

            else if (RD_NOT_WRITE_C2)

                nx_client_write = idle; //+ In client 1 write state and there is a read request from client 2, transition to idle state

            end

        end

        // Handling client2 read state

        if (pr_client_read == client2_read) begin

            if (!RD_EN_C1) begin

                if (REQUEST_C2) begin

                    if (RD_NOT_WRITE_C2)

                        nx_client_read = client2_read; //+ In client 2 read state and there is a read request from client 2, stay in client 2 read state

                    else begin

                        nx_client_read = idle; //+ In client 2 read state and there is a write request from client 2, transition to idle state

                        nx_client_write = client2_write; //+ In client 2 read state and there is a write request from client 2, transition to client 2 write state

                    end

                end

                else

                    nx_client_read = idle; //+ In client 2 read state and no requests from client 2, transition to idle state

            end

            else

                nx_client_read = client1_read; //+ In client 2 read state and there is a read request from client 1, transition to client 1 read state
```

```
        end

        // Handling client2 write state

        if (pr_client_write == client2_write) begin

            if (!WR_EN_C1) begin

                if (REQUEST_C2) begin

                    if (!RD_NOT_WRITE_C2)

                        nx_client_write = client2_write; //+ In client 2 write state and there is a write request from client 2, stay in client 2 write state

                    else begin

                        nx_client_write = idle; //+ In client 2 write state and there is a read request from client 2, transition to idle state

                        nx_client_read = client2_read; //+ In client 2 write state and there is a read request from client 2, transition to client 2 read state

                    end

                end

                else

                    nx_client_write = idle; //+ In client 2 write state and no requests from client 2, transition to idle state

            end

            else

                nx_client_write = client1_write; //+ In client 2 write state and there is a write request from client 1, transition to client 1 write state

        end

    end

    endmodule
```

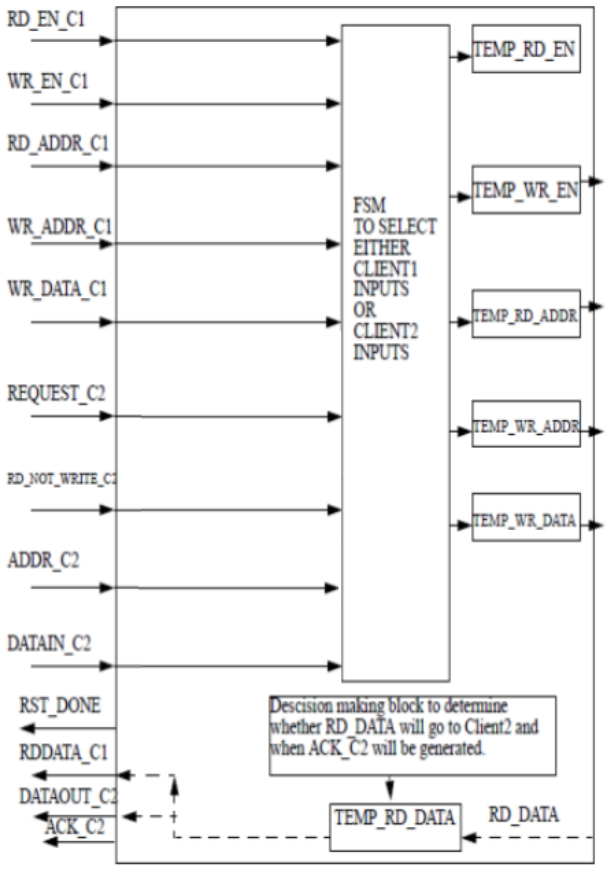
3.1.4. SCHEMATIC



Schematic of FSM

3.2. ARBITER

3.2.1. BLOCK DIAGRAM



Block Diagram Of Arbiter

3.2.2. CODING

```
module arbiter #(
    parameter G_ADDR_WIDTH = 4,
    parameter G_DATA_WIDTH = 8,
    parameter G_REGISTERED_DATA = 0)
(
    input RST_N, CLOCK,
    output RST_DONE,
    input RD_EN_C1, WR_EN_C1,       // Read enable, write enable
    input [G_ADDR_WIDTH-1 : 0] RDADDR_C1, WRADDR_C1,   // Read address, write address
    input [G_DATA_WIDTH-1 : 0] WRDATA_C1,
    input [G_DATA_WIDTH-1 : 0] DATAIN_C2,
    input REQUEST_C2, RD_NOT_WRITE_C2,         // Request for memory access and read-write_bar
    input [G_ADDR_WIDTH-1 : 0] ADDR_C2,          // This address is for both read and write one at a time
    output [G_DATA_WIDTH-1 : 0] RDDATA_C1,     // Data out for client1/user1
    output [G_DATA_WIDTH-1 : 0] DATAOUT_C2,  // Data out for client2/user2
    output ACK_C2,                  // Acknowledgement in case of client2, since it will not be given access immediately since priority is more of client1
    output RD_EN, WR_EN,
    output [G_ADDR_WIDTH-1 : 0] WR_ADDR, RD_ADDR,
    output [G_DATA_WIDTH-1 : 0] WR_DATA,
    input [G_DATA_WIDTH-1 : 0] RD_DATA
);
// Registers to hold temporary data
```

```
    reg [G_DATA_WIDTH-1 : 0] TEMP_RD_DATA, TEMP_RD_DATA1, TEMP_RD_DATA2;

    reg TEMP_RD_EN, TEMP_WR_EN;

    reg [G_ADDR_WIDTH-1 : 0] TEMP_WR_ADDR, TEMP_RD_ADDR;

    reg [G_DATA_WIDTH-1 : 0] TEMP_WR_DATA;




    // FSM states

    localparam [2:0] reset = 3'b000, idle = 3'b001, client1_read = 3'b010, client2_read = 3'b011, client1_write = 3'b100, client2_write = 3'b101;

    // FSM state registers

    reg [2:0] pr_client_read, pr_client_write, nx_client_read, nx_client_write;

    // Acknowledgement registers

    reg TEMP_ACK = 0, TEMP_ACK1, TEMP_ACK2, TEMP_WR=0;

    wire TEMP_WR1;

    // Wire to determine if data should be registered

    wire REGISTERED_DATA = 0;

    // Reset-related variables

    reg RESET_DONE_REG;

    integer COUNT = 0;

    // Address clash detection signals

    reg ADDR_CLASHI=0, ADDR_CLASH=0;

    // FSM

    always @(posedge CLOCK, negedge RST_N) begin

        if (!RST_N) begin

                pr_client_read <= reset;
```

```
                pr_client_write <= reset;
        end
        else begin
                pr_client_read <= nx_client_read;
                pr_client_write <= nx_client_write;
        end
    end
    // Conditional assignment of REGISTERED_DATA based on the parameter G_REGISTERED_DATA
    generate
        if (G_REGISTERED_DATA) begin : g1
                assign REGISTERED_DATA = G_REGISTERED_DATA;
        end
    endgenerate
    // FSM logic
    always @(pr_client_read, pr_client_write, CLOCK) begin
        if (RST_N & CLOCK) begin
                // Reset logic
                if (nx_client_read == reset && nx_client_write == reset) begin
                        if (COUNT < (2**G_ADDR_WIDTH)) begin
                                RESET_DONE_REG <= 1'b0;
                                COUNT <= COUNT + 1;
                        end
                        else begin
                                nx_client_read = idle;
```

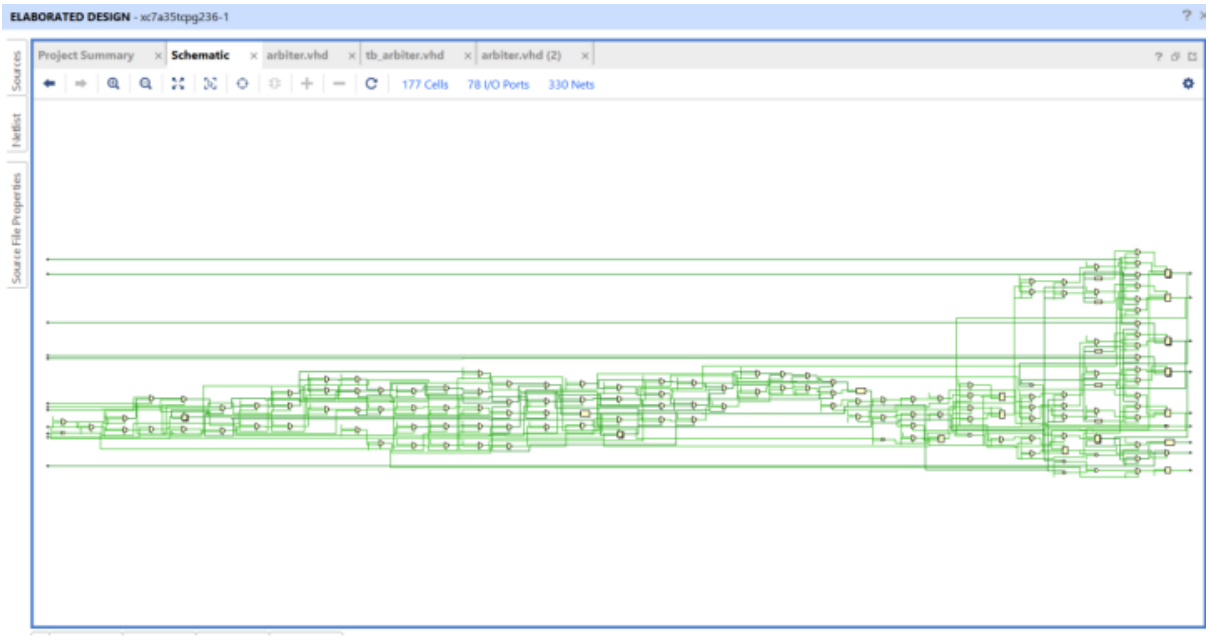
```
                                nx_client_write = idle;
                                RESET_DONE_REG = 1'b1;
                                COUNT = 0;
                        end
                end
        end
        else if(!RST_N) begin
                nx_client_read = reset;
                nx_client_write = reset;
        end
        // FSM state transitions
        // ... (state transition logic)
end
// Register data assignment logic
always @(posedge CLOCK) begin
        if (!RST_N) begin
                TEMP_RD_DATA <= 0;
                TEMP_RD_DATA1 <= 0;
                TEMP_RD_DATA2 = 0;
        end
        else begin
                // Read data assignment
                if (nx_client_read == idle) begin
                        TEMP_RD_EN <= 1'b0;
                        TEMP_RD_ADDR <= 0;
```

```
        end
        else if (nx_client_read == client1_read) begin
                TEMP_RD_EN <= RD_EN_C1;
                TEMP_RD_ADDR <= RDADDR_C1;
        end
        else if (nx_client_read == client2_read) begin
                if (!TEMP_ACK) begin
                        TEMP_RD_EN <= 1'b1;
                        TEMP_RD_ADDR <= ADDR_C2;
                        TEMP_ACK <= 1'b1;
                end
        end
        // Write data assignment
        if (nx_client_write == idle) begin
                TEMP_WR_EN <= 1'b0;
                TEMP_WR_DATA <= 0;
                TEMP_WR_ADDR <= 0;
        end
        else if (nx_client_write == client1_write) begin
                TEMP_WR_EN <= WR_EN_C1;
                TEMP_WR_DATA <= WRDATA_C1;
                TEMP_WR_ADDR <= WRADDR_C1;
        end
        else if (nx_client_write == client2_write) begin
                if (!TEMP_WR) begin
```

```
                                TEMP_WR_EN <= 1'b1;
                                TEMP_WR_ADDR <= ADDR_C2;
                                TEMP_WR_DATA <= DATAIN_C2;
                                TEMP_WR <= 1'b1;
                            end
                        end
                        // Additional logic
                        ADDR_CLASHI <= ADDR_CLASH;
                        TEMP_RD_DATA1 <= TEMP_RD_DATA;
                        TEMP_RD_DATA2 <= RD_DATA;
                    end
                end
                // Output assignments
                assign RD_EN = TEMP_RD_EN;
                assign WR_EN = TEMP_WR_EN;
                assign WR_DATA = TEMP_WR_DATA;
                assign WR_ADDR = TEMP_WR_ADDR;
                assign RD_ADDR = TEMP_RD_ADDR;
                assign TEMP_WR1 = TEMP_WR;
                assign ACK_C2 = (TEMP_ACK1 | TEMP_WR1) ? 1'b1 : 1'b0;
                assign RST_DONE = RESET_DONE_REG;
                assign DATAOUT_C2 = (!ADDR_CLASH) ? RD_DATA : TEMP_RD_DATA;
                assign RDDATA_C1 = (REGISTERED_DATA == 0 && ADDR_CLASH == 1'b0 ) ?
RD_DATA :
                                    (REGISTERED_DATA == 0 && ADDR_CLASH == 1'b1 ) ?
TEMP_RD_DATA :
```

```
                                (REGISTERED_DATA == 1 && ADDR_CLASHI == 1'b0 )
? TEMP_RD_DATA2 :

                                (REGISTERED_DATA == 1 && ADDR_CLASHI == 1'b1 )
? TEMP_RD_DATA1 : RDDATA_C1;

    End
```

3.2.3. SCHEMATIC



Arbiter's schematic

3.2.4. TESTCASES

Case 1 : Only Client1 wants to write.
Case 2 : Only Client1 wants to read.
Case 3 : Only Client2 wants to write.
Case 4: Only Client2 wants to read.

3.2.5. VERILOG TESTBENCH

3.2.5.1.TESTBENCH CODE

```
module arbiter;

  // Component Declaration for the Unit Under Test (UUT)

  ARBITER_NEW uut (

    .RST_N(RST_N),          // Reset input to the UUT
```

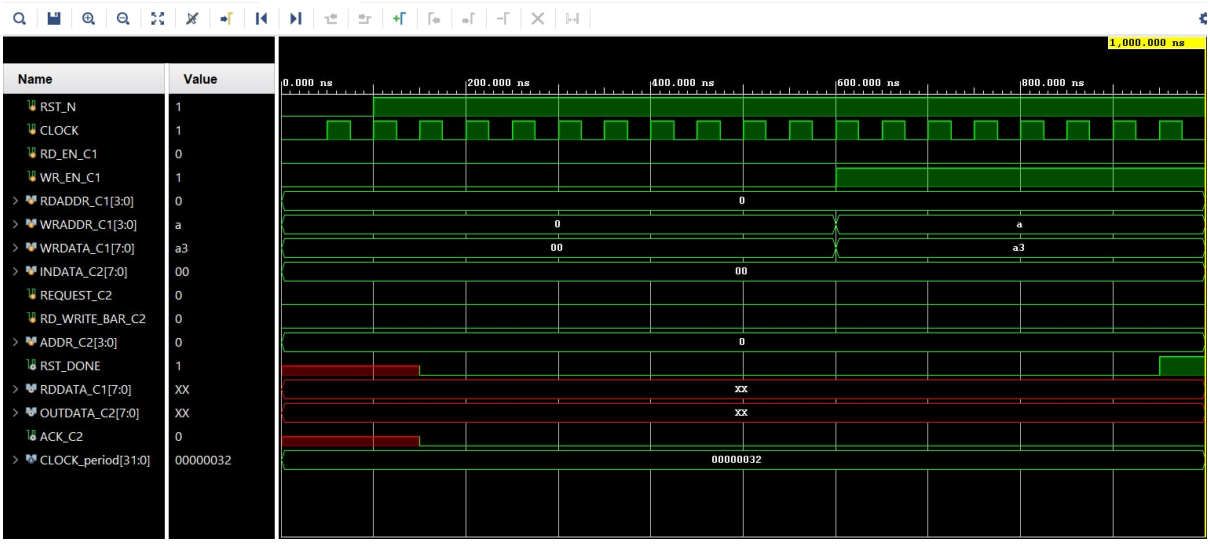
```
    .CLOCK(CLOCK),          // Clock input to the UUT
    .RST_DONE(RST_DONE),    // Output signal indicating reset is done
    .RD_EN_C1(RD_EN_C1),    // Read enable for client1
    .WR_EN_C1(WR_EN_C1),    // Write enable for client1
    .RDADDR_C1(RDADDR_C1),  // Read address for client1
    .WRADDR_C1(WRADDR_C1),  // Write address for client1
    .WRDATA_C1(WRDATA_C1),  // Write data for client1
    .DATAIN_C2(DATAIN_C2),  // Data input for client2
    .REQUEST_C2(REQUEST_C2), // Request signal for client2
    .RD_NOT_WRITE_C2(RD_NOT_WRITE_C2), // Read not write signal for client2
    .ADDR_C2(ADDR_C2),      // Address signal for client2
    .RDDATA_C1(RDDATA_C1),  // Read data for client1
    .DATAOUT_C2(DATAOUT_C2), // Data output for client2
    .ACK_C2(ACK_C2),        // Acknowledge signal for client2
    .RD_EN(RD_EN),          // Read enable signal
    .WR_EN(WR_EN),          // Write enable signal
    .WR_ADDR(WR_ADDR),      // Write address signal
    .RD_ADDR(RD_ADDR),      // Read address signal
    .WR_DATA(WR_DATA),      // Write data signal
    .RD_DATA(RD_DATA)       // Read data signal
);
// Inputs
reg RST_N = 0;           // Reset signal (active low)
reg CLOCK = 0;           // Clock signal
reg RD_EN_C1 = 0;        // Read enable for client1
```

```
    reg WR_EN_C1 = 0;          // Write enable for client1
    reg [3:0] RDADDR_C1 = 4'b0000; // Read address for client1
    reg [3:0] WRADDR_C1 = 4'b0000; // Write address for client1
    reg [7:0] WRDATA_C1 = 8'b00000000; // Write data for client1
    reg [7:0] DATAIN_C2 = 8'b00000000; // Data input for client2
    reg REQUEST_C2 = 0;         // Request signal for client2
    reg RD_NOT_WRITE_C2 = 0;    // Read not write signal for client2
    reg [3:0] ADDR_C2 = 4'b0000; // Address signal for client2
    reg [7:0] RD_DATA = 8'b00000000; // Read data for client1

    // Outputs
    wire RST_DONE;             // Output signal indicating reset is done
    wire [7:0] RDDATA_C1;      // Read data for client1
    wire [7:0] DATAOUT_C2;     // Data output for client2
    wire ACK_C2;               // Acknowledge signal for client2
    wire RD_EN;                // Read enable signal
    wire WR_EN;                // Write enable signal
    wire [3:0] WR_ADDR;        // Write address signal
    wire [3:0] RD_ADDR;        // Read address signal
    wire [7:0] WR_DATA;        // Write data signal
    // Clock period definitions
    parameter CLOCK_period = 100; // Clock period set to 100 time units
    // Clock process definitions
    always #CLOCK_period/2 CLOCK = ~CLOCK; // Clock toggles every half the specified
period
```

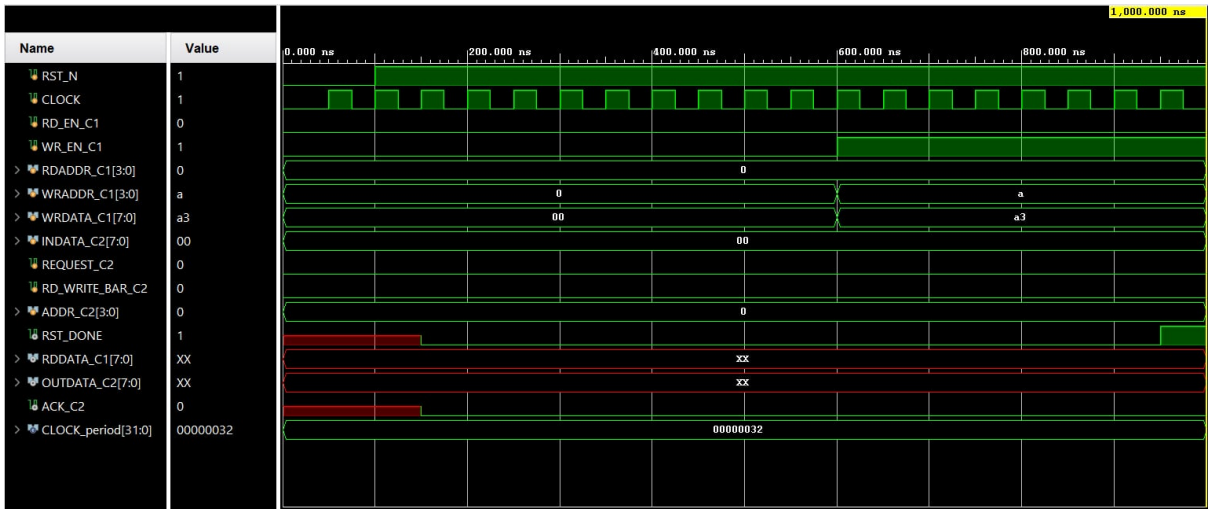
```
  // Stimulus process
  initial begin
    // Test case
    // Hold reset state for 100 ns.
    #100;
    // Add more test cases here…
#Case 1: Only Client1 wants to write
RST_N <= 1;
#500;
WR_EN_C1 <= 1;
WRADDR_C1 <= 4'b1010; WRDATA_C1 <= 8'b10100011;
#Case 2: Only Client1 wants to read
RST_N <= 1; #500;
WR_EN_C1 <= 1;
WRADDR_C1 <= 4'b1010;
WRDATA_C1 <= 8'b10100011;
#1700;
WR_EN_C1 <= 0;
RD_EN_C1 <= 1;
RDADDR_C1 <= 4'b1010;
#Case 3: Only Client2 wants to write
RST_N <= 1;
WR_EN_C1 <= 0;
REQUEST_C2 <= 1;
RD_NOT_WRITE_C2 <= 0;
```

```
ADDR_C2 <= 4'b1110;

DATAIN_C2 <= 8'b11100011;

#Case 4: Only Client2 wants to read

RST_N <= 1;

WR_EN_C1 <= 0;

REQUEST_C2 <= 1; RD_NOT_WRITE_C2 <= 0;

ADDR_C2 <= 4'b1110;

DATAIN_C2 <= 8'b11100011; #1700;  WR_EN_C1 <= 0; RD_NOT_WRITE_C2 <= 1;

ADDR_C2 <= 4'b1110;

    // End simulation after running test cases

    $stop;

  end

endmodule
```

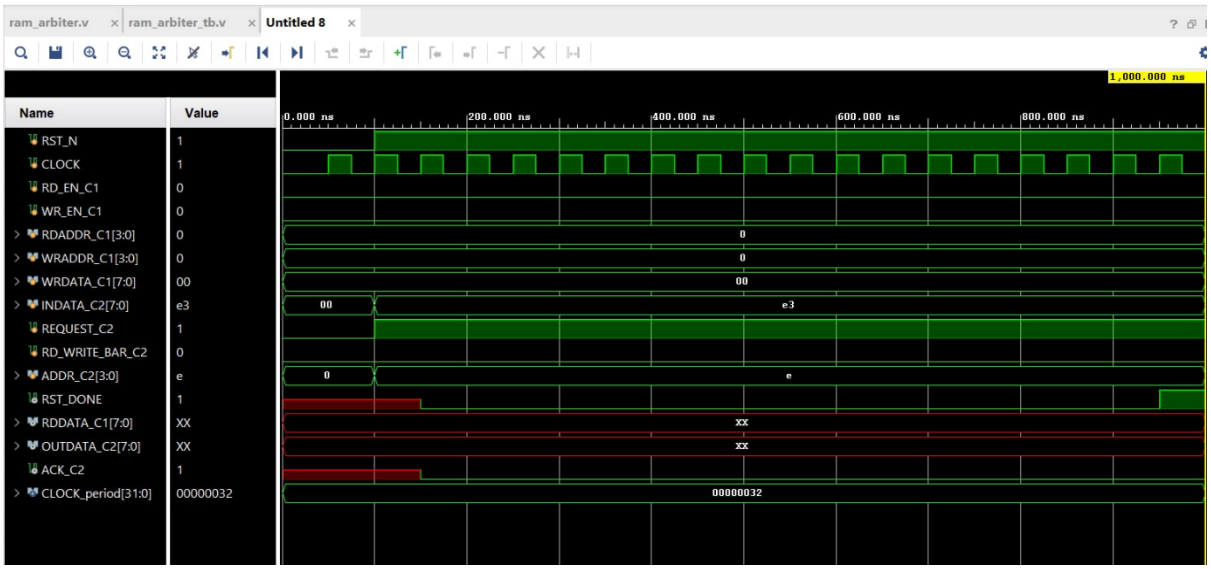
WAVEFORMS



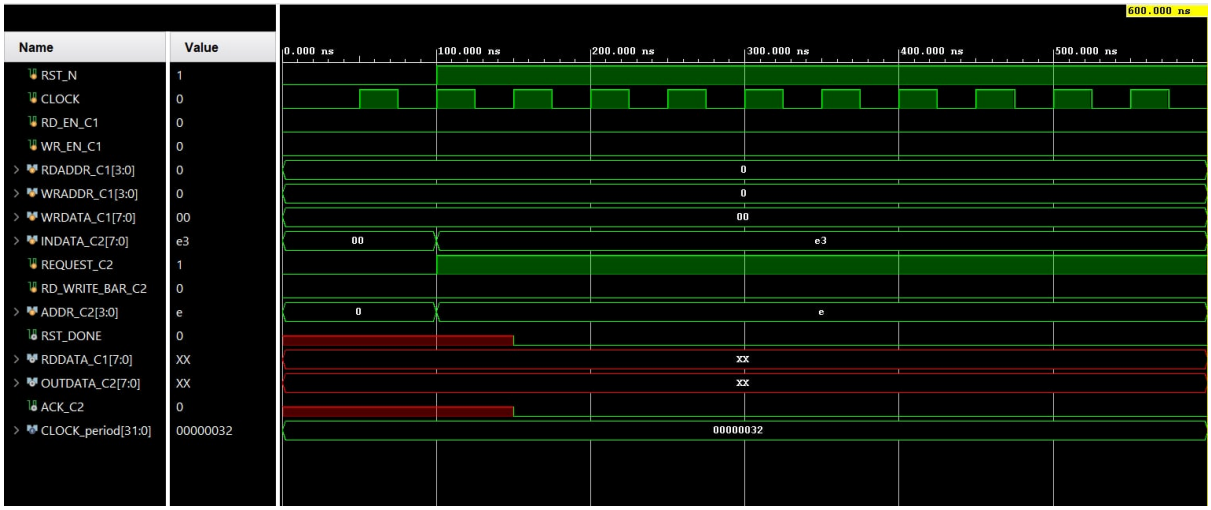
Case 1: Only Client1 wants to write



Case 2: Only Client1 wants to read (Fail simulation)



Case 3: Only Client2 wants to write



Case 4: Only Client2 wants to read (Fail simulation)

3.2.5.2. ANALYSIS OF THE TESTCASES

Case1: Only Client1 wants to write.

In this case, when the RST_N pin is enabled, the arbiter starts its operation. Within the first RAM_DEPTH cycle, Client1 enables its WR_EN_C1 pin and writes the data " a3 " at the address location " a ". The arbiter places the data it received, on the input pins of the RAM i.e. the WR_ADDR receives the address location and the WR_DATA receives the data.

Case 2: Only Client1 wants to read.

The Arbiter starts working as soon as the RST_N pin is enabled. The clock period is set to 50ns. Within the first RAM_DEPTH cycle, a data " a3 " is written at the location " a ". The R_EN_C1 pin is enabled to facilitate the operation. During the next RAM_DEPTH cycle, Client1 wishes to read at the RAM location " a " by enabling the RD_EN_C1 pin.

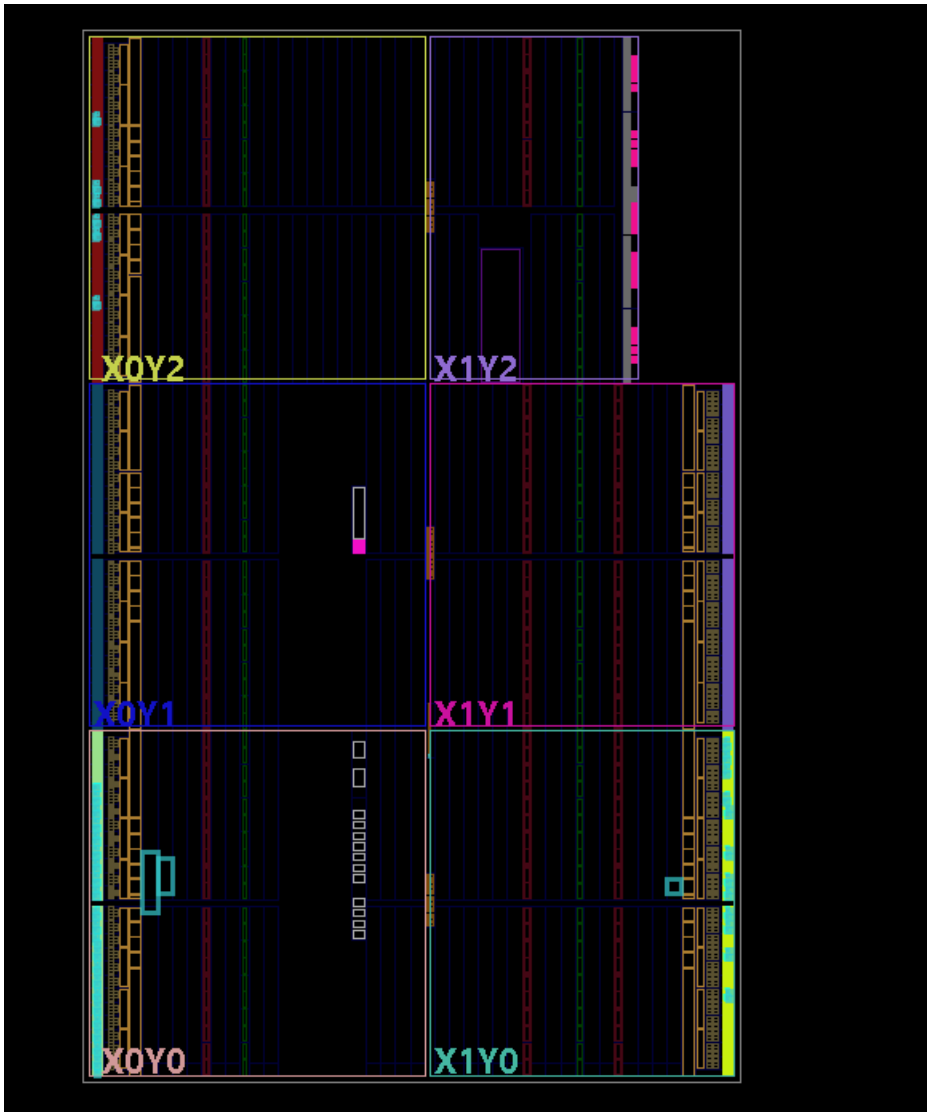
Case3: Only Client2 wants to write.

Here in this case, the Arbiter starts working as soon as the RST_N pin is set high. The clock period is set to 50 ns. As the arbitration scheme used is a priority based one, so Client2 needs to issue a signal on the REQUEST_C2 pin in order to access the RAM. Here the priority is given to Client1.Within the first RAM_DEPTH cycle, REQUEST_C2 pin is enabled to allow Client2 to access the RAM. Client2 uses two states of a single pin RD_NOT_WRITE to perform both of its read and write operation. This pin only works if REQUEST_C2 pin is enabled. Upon enabling REQUEST_C2, RD_NOT_WRITE is set to low to allow for the write operation. Data " e3 " is written at the location " e ". The arbiter acknowledges the write operation by setting up periodic pulses at the ACK_C2 output pin.

Case 4: Only Client2 wants to read.

In this case, within the first RAM_DEPTH cycle, data " e3 " is written in the address location " e " by enabling the REQUEST_C2 pin and setting RD_NOT_WRITE pin to an active low state. However in the second RAM_DEPTH cycle, Client2 activates the high signal on RD_NOT_WRITE pin and issue a signal to the arbiter to read from location " e ". The read operation is acknowledged by the Arbiter by issuing clock pulses at the ACK_C2 pin whose clock period is two times that of the former one.

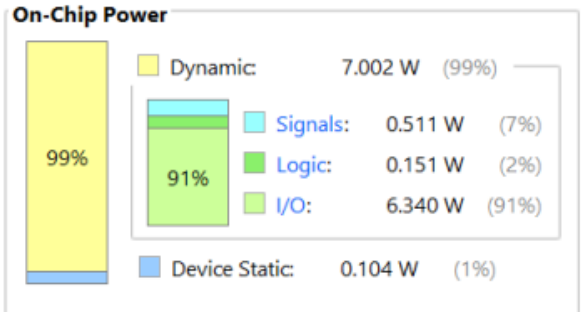
3.2.6. FPGA



3.2.7. Power & Timing

Power analysis from Implemented netlist. Activity derived from constraints files, simulation files or vectorless analysis.

Total On-Chip Power:	**7.105 W**
Design Power Budget:	**Not Specified**
Process:	typical
Power Budget Margin:	**N/A**
Junction Temperature:	**60.5°C**



POWER

Name	Slack	Levels	Routes	High Fanout	From	To	Total Delay	Logic Delay	Net Delay	Logic %	Net %	Requirement	Source Clock
Unconstrained Paths (1)													
(none) (10)													
Path 11	∞	3	2	2	RD_DATA[5]	DATAOUT_C2[5]	12.397	3.910	8.487	31.5	68.5	∞	input port clock
Path 12	∞	3	2	2	RD_DATA[7]	DATAOUT_C2[7]	12.369	3.906	8.463	31.6	68.4	∞	input port clock
Path 13	∞	3	2	2	RD_DATA[6]	DATAOUT_C2[6]	12.289	3.668	8.621	29.8	70.2	∞	input port clock
Path 14	∞	3	2	2	RD_DATA[4]	DATAOUT_C2[4]	12.072	3.657	8.415	30.3	69.7	∞	input port clock
Path 15	∞	3	2	2	RD_DATA[0]	DATAOUT_C2[0]	12.014	3.884	8.130	32.3	67.7	∞	input port clock
Path 16	∞	3	2	2	RD_DATA[3]	DATAOUT_C2[3]	11.882	3.886	7.996	32.7	67.3	∞	input port clock
Path 17	∞	3	2	2	RD_DATA[5]	RDDATA_C1[5]	11.134	3.953	7.181	35.5	64.5	∞	input port clock
Path 18	∞	3	2	2	RD_DATA[0]	RDDATA_C1[0]	10.871	3.864	7.007	35.5	64.5	∞	input port clock
Path 19	∞	3	2	2	RD_DATA[2]	DATAOUT_C2[2]	10.838	3.662	7.176	33.8	66.2	∞	input port clock
Path 20	∞	3	2	2	RD_DATA[2]	RDDATA_C1[2]	10.837	3.659	7.177	33.8	66.2	∞	input port clock

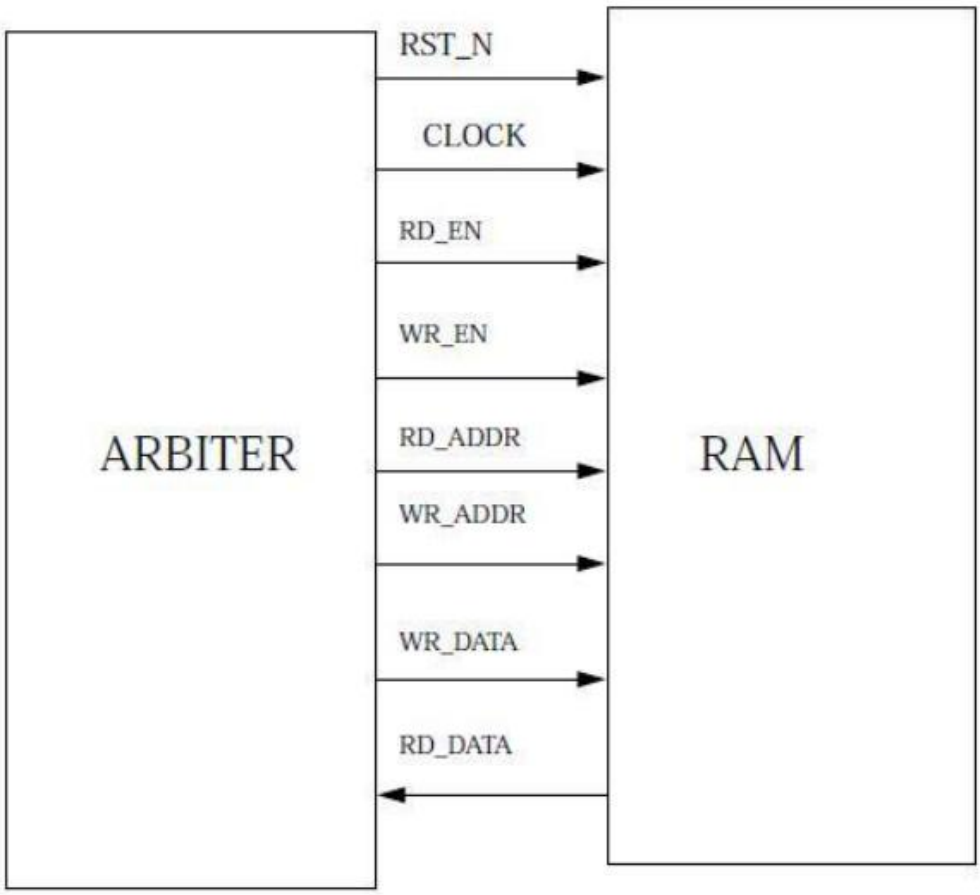
SET UP TIME

Name	Slack	Levels	Routes	High Fanout	From	To	Total Delay	Logic Delay	Net Delay	Logic %	Net %	Requirement
Unconstrained Paths (1)												
(none) (10)												
Path 1	∞	1	1	1	TEMP_WR_DATA_reg[6]/C	TEMP_RD_DATA_reg[6]/D	0.286	0.128	0.158	44.8	55.2	-∞
Path 2	∞	1	1	12	nx_client_read_reg[0]/G	pr_client_read_reg[0]/D	0.302	0.178	0.124	58.9	41.1	-∞
Path 3	∞	2	1	7	pr_client_write_reg[1]/C	nx_client_write_reg[1]/D	0.326	0.186	0.140	57.1	42.9	-∞
Path 4	∞	1	1	1	TEMP_WR_DATA_reg[3]/C	TEMP_RD_DATA_reg[3]/D	0.328	0.141	0.187	43.0	57.0	-∞
Path 5	∞	1	1	1	TEMP_WR_DATA_reg[1]/C	TEMP_RD_DATA_reg[1]/D	0.339	0.141	0.198	41.6	58.4	-∞
Path 6	∞	1	1	1	TEMP_WR_DATA_reg[5]/C	TEMP_RD_DATA_reg[5]/D	0.343	0.128	0.215	37.3	62.7	-∞
Path 7	∞	1	1	44	nx_client_read_reg[1]/G	pr_client_read_reg[1]/D	0.344	0.178	0.166	51.8	48.2	-∞
Path 8	∞	1	1	1	TEMP_WR_DATA_reg[7]/C	TEMP_RD_DATA_reg[7]/D	0.348	0.128	0.220	36.7	63.3	-∞
Path 9	∞	1	1	1	TEMP_WR_DATA_reg[0]/C	TEMP_RD_DATA_reg[0]/D	0.356	0.141	0.215	39.6	60.4	-∞
Path 10	∞	1	1	20	nx_client_write_reg[0]/G	pr_client_write_reg[0]/D	0.365	0.158	0.207	43.3	56.7	-∞

HOLD TIME

4. RAM ARBITER

4.1. PORT MAP IN BETWEEN RAM AND ARBITER



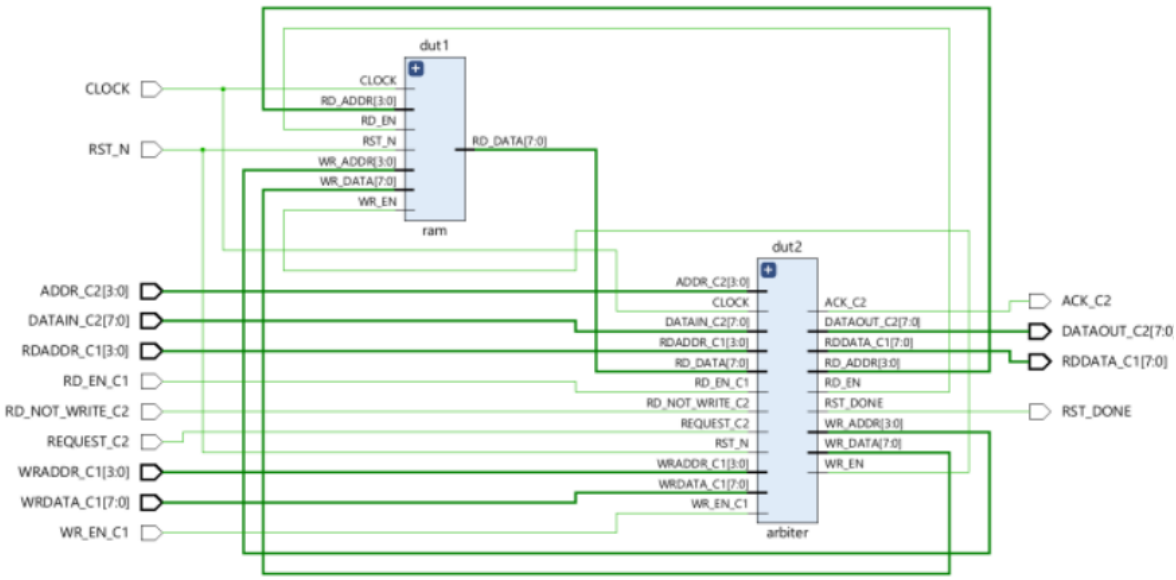
4.2. CODING

```
module ram_arbiter #(
    parameter G_ADDR_WIDTH = 4,     // Number of bits required to address the RAM
    parameter G_DATA_WIDTH = 8,     // Number of bits in a data
    parameter G_REGISTERED_DATA = 0  // 1 if output data is registered, 0 if not
)
(
    input RST_N, CLOCK,             // Reset and Clock signals
    output RST_DONE,                // Reset done signal

    input RD_EN_C1, WR_EN_C1,       // Read Enable and Write Enable signals from Client 1
    input [G_ADDR_WIDTH-1 : 0] RDADDR_C1, WRADDR_C1,  // Read and Write addresses from Client 1
    input [G_DATA_WIDTH-1 : 0] WRDATA_C1,          // Write data from Client 1
```

```
        input [G_DATA_WIDTH-1 : 0] DATAIN_C2,          // Input data from Client 2
        input REQUEST_C2, RD_NOT_WRITE_C2,             // Memory access request
and             read/write bar signal from Client 2
        input [G_ADDR_WIDTH-1 : 0] ADDR_C2,            // Address for both read and
write           operations from Client 2

        output [G_DATA_WIDTH-1 : 0] RDDATA_C1,         // Data out for Client 1
        output [G_DATA_WIDTH-1 : 0] DATAOUT_C2,        // Data out for Client 2
        output ACK_C2                                  // Acknowledgement signal for Client 2
    );

    // Wires to connect signals to the RAM module
    wire [G_DATA_WIDTH-1 : 0] WR_DATA, RD_DATA1;
    wire [G_ADDR_WIDTH-1 : 0] WR_ADDR, RD_ADDR;
    wire RD_EN, WR_EN;

    // Instantiate RAM module with parameters
    ram #(G_ADDR_WIDTH, G_DATA_WIDTH) dut1 (
        .CLOCK(CLOCK),
        .RST_N(RST_N),
        .RD_EN(RD_EN),
        .WR_EN(WR_EN),
        .RD_ADDR(RD_ADDR),
        .WR_ADDR(WR_ADDR),
        .WR_DATA(WR_DATA),
        .RD_DATA(RD_DATA1)
    );

    // Instantiate Arbiter module with parameters
    arbiter #(G_ADDR_WIDTH,G_DATA_WIDTH,G_REGISTERED_DATA) dut2 (
        .RST_N(RST_N),
        .CLOCK(CLOCK),
        .RST_DONE(RST_DONE),
        .RD_EN_C1(RD_EN_C1),
        .WR_EN_C1(WR_EN_C1),
        .RDADDR_C1(RDADDR_C1),
        .WRADDR_C1(WRADDR_C1),
        .WRDATA_C1(WRDATA_C1),
        .REQUEST_C2(REQUEST_C2),
        .RD_NOT_WRITE_C2(RD_NOT_WRITE_C2),
        .ADDR_C2(ADDR_C2),
        .DATAIN_C2(DATAIN_C2),
        .RD_EN(RD_EN),
        .WR_EN(WR_EN),
```

```
    .RD_ADDR(RD_ADDR),
    .WR_ADDR(WR_ADDR),
    .WR_DATA(WR_DATA),
    .RD_DATA(RD_DATA1),
    .DATAOUT_C2(DATAOUT_C2),
    .ACK_C2(ACK_C2),
    .RDDATA_C1(RDDATA_C1)
);

endmodule
```

4.3. SCHEMATIC OF RAM ARBITER



4.4. TESTCASES FOR RAM ARBITER

1. Client1 wants to read and Client2 wants to read in same RAM location at different time.
2. Client1 wants to read and Client2 wants to read in same RAM location at same time.
3. Client1 wants to read and write in the same RAM location and Client2 also wants to read in the RAM location where Client1 has written at different time.
4. Client1 wants to read and write in the same RAM location and Client2 also wants to read in the RAM location where Client1 has written at same time.
5. Client2 wants to read and write in the same RAM location and Client1 also wants to write in the RAM location where Client2 has written at same time.
6. Client2 wants to read and write in the same RAM location and Client1 also wants to read in the RAM location where Client2 has written at same time.

4.5. VERILOG TESTBENCH

4.5.1. TESTBENCH CODE

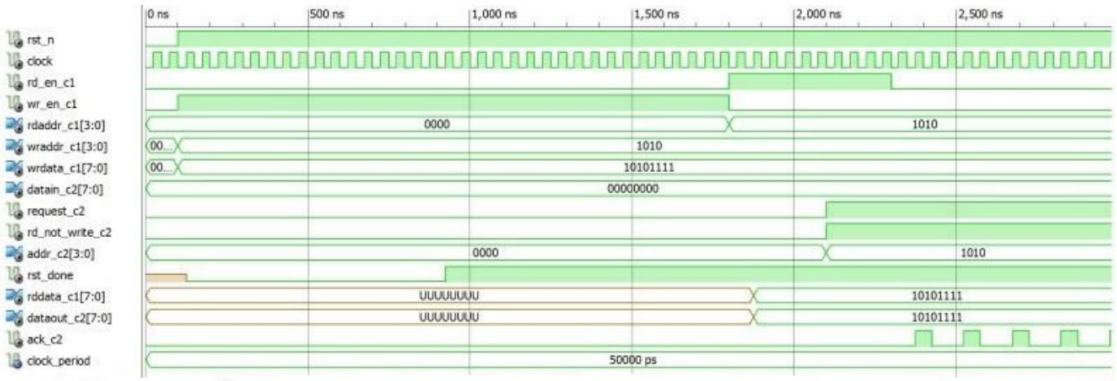
```
`timescale 1ns / 1ps

module RAM_ARBITER_TEST;

 // Inputs
 reg RST_N;
 reg CLOCK;
 reg RD_EN_C1;
 reg WR_EN_C1;
 reg [3:0] RDADDR_C1;
 reg [3:0] WRADDR_C1;
 reg [7:0] WRDATA_C1;
 reg [7:0] DATAIN_C2;
 reg REQUEST_C2;
 reg RD_NOT_WRITE_C2;
 reg [3:0] ADDR_C2;

 // Outputs
 wire RST_DONE;
 wire [7:0] RDDATA_C1;
 wire [7:0] DATAOUT_C2;
 wire ACK_C2;

 // Instantiate the Unit Under Test (UUT)
 RAM_ARBITER_NEW uut (
  .RST_N(RST_N),
  .CLOCK(CLOCK),
  .RST_DONE(RST_DONE),
  .RD_EN_C1(RD_EN_C1),
  .WR_EN_C1(WR_EN_C1),
  .RDADDR_C1(RDADDR_C1),
  .WRADDR_C1(WRADDR_C1),
  .WRDATA_C1(WRDATA_C1),
  .DATAIN_C2(DATAIN_C2),
  .REQUEST_C2(REQUEST_C2),
  .RD_NOT_WRITE_C2(RD_NOT_WRITE_C2),
  .ADDR_C2(ADDR_C2),
  .RDDATA_C1(RDDATA_C1),
  .DATAOUT_C2(DATAOUT_C2),
  .ACK_C2(ACK_C2)
```

```
    );

    // Clock process definitions
    always begin
      #25 CLOCK = ~CLOCK;
    End

    // Stimulus process
    initial begin

      // Test Case-1: Client1 wants to read and Client2 wants to read in same RAM location at
    different time.

     #100 RST_N = 1;
     #200 WR_EN_C1 = 1;
    #100 RD_EN_C1 = 0;
    #200 WRADDR_C1 = 4'b1010;
    #200 WRDATA_C1 = 8'b10101111;
    #1700 RD_EN_C1 = 1;
    #300 WR_EN_C1 = 0;
    #200 RDADDR_C1 = 4'b1010;

      // Test Case-2: Client1 wants to read and Client2 wants to read in same RAM location at same time.

    #100 RST_N = 1;
    #200 WR_EN_C1 = 1;
    #100 RD_EN_C1 = 0;
    #200 WRADDR_C1 = 4'b1010;
    #200 WRDATA_C1 = 8'b10101111;
    #1700 RD_EN_C1 = 1;
    #200 WR_EN_C1 = 0;
    #200 RDADDR_C1 = 4'b1010;
    #100 REQUEST_C2 = 1;
    #100 RD_NOT_WRITE_C2 = 1;
    #200 ADDR_C2 = 4'b1010;

      // Test Case-3: Client1 wants to read and write in the same RAM location and Client2 also
    wants to read in the RAM location where Client1 has written at different time.

     #100 RST_N = 1;
    #200 WR_EN_C1 = 1;
```

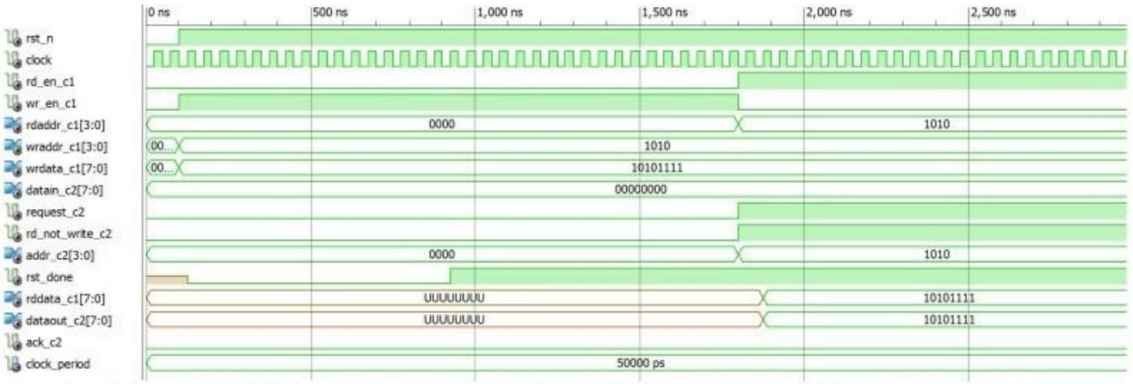
```
    #100 RD_EN_C1 = 0;
    #200 WRADDR_C1 = 4'b1001;
    #200 WRDATA_C1 = 8'b10101111;
    #1700 RD_EN_C1 = 1;
    #200 RDADDR_C1 = 4'b1001;
    #200 WRADDR_C1 = 4'b1001;
    #200 WRDATA_C1 = 8'b10100011;
    #300 REQUEST_C2 = 1;
    #100 RD_NOT_WRITE_C2 = 1;
    #200 ADDR_C2 = 4'b1001;
    #200 RD_EN_C1 = 0;

        // Test Case-4: Client1 wants to read and write in the same RAM location and Client2 also
    wants to read in the RAM location where Client1 has written at same time.

        #100 RST_N = 1;
        #200 WR_EN_C1 = 1;
        #100 RD_EN_C1 = 0;
        #200 WRADDR_C1 = 4'b1001;
        #200 WRDATA_C1 = 8'b10101111;
        #1700 RD_EN_C1 = 1;
        #200 RDADDR_C1 = 4'b1001;
        #200 WRADDR_C1 = 4'b1001;
        #200 WRDATA_C1 = 8'b10100011;
        #100 REQUEST_C2 = 1;
        #100 RD_NOT_WRITE_C2 = 1;
        #200 ADDR_C2 = 4'b1001;
        #200 RDADDR_C1 = 4'b1001;

        // Test Case-5: Client2 wants to read and write in the same RAM location and Client1 also
    wants to write in the RAM location where Client2 has written at same time.

        #100 RST_N = 1;
        #200 WR_EN_C1 = 0;
        #200 RD_EN_C1 = 0;
        #100 REQUEST_C2 = 1;
        #100 RD_NOT_WRITE_C2 = 0;
        #200 ADDR_C2 = 4'b1001;
        #200 DATAIN_C2 = 8'b11100011;
        #1700 WR_EN_C1 = 1;
        #200 RD_NOT_WRITE_C2 = 1;
        #200 ADDR_C2 = 4'b1001;
```

```
        #200 WRADDR_C1 = 4'b1001;
        #200 WRDATA_C1 = 8'b10101111;

        // Test Case-6: Client2 wants to read and write in the same RAM location and
Client1 also
    wants to read in RAM location where Client2 has written at same time.

        #100 RST_N = 1;
        #200 WR_EN_C1 = 0;
        #200 RD_EN_C1 = 0;
        #100 REQUEST_C2 = 1;
        #100 RD_NOT_WRITE_C2 = 0;
        #200 ADDR_C2 = 4'b1001;
        #200 DATAIN_C2 = 8'b11100011;
        #1700 RD_EN_C1 = 1;
        #200 RD_NOT_WRITE_C2 = 1;
        #200 ADDR_C2 = 4'b1001;
        #200 RDADDR_C1 = 4'b1001;
      end

    endmodule
```

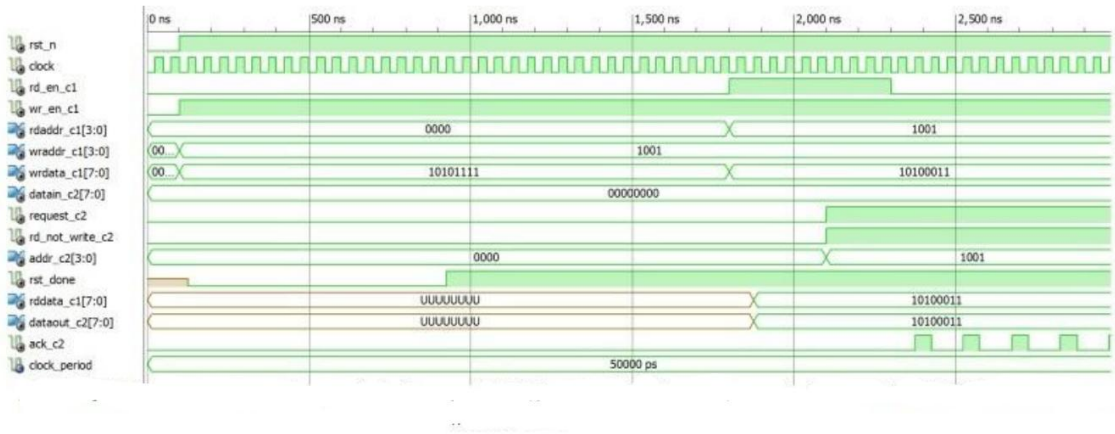
WAVEFORM



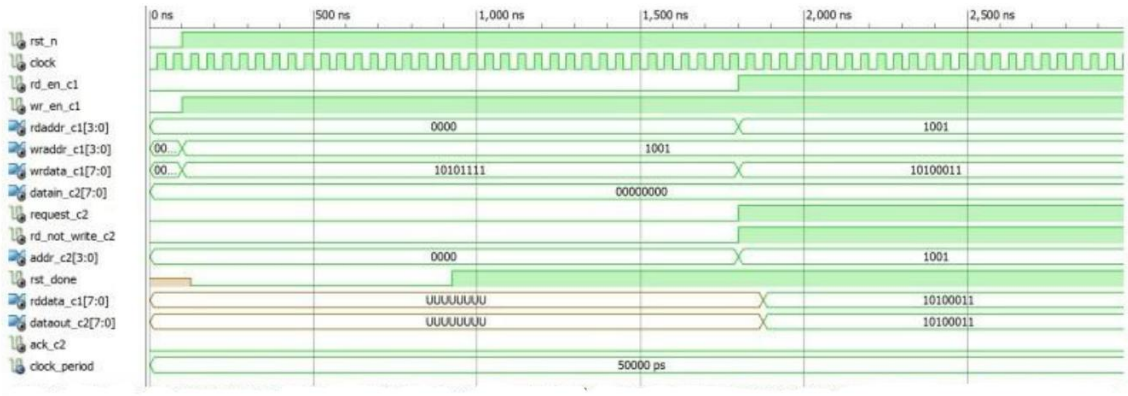
Case 1: Client1 wants to read and Client2 wants to read in same RAM location at different time.



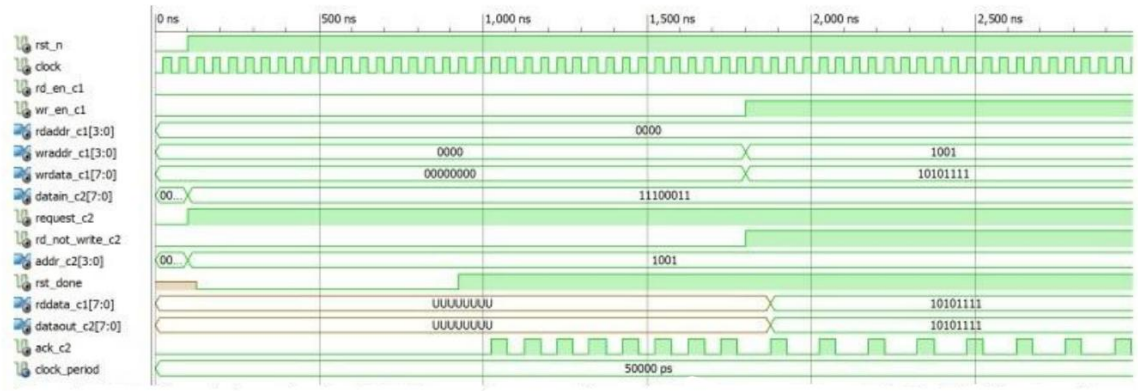
Case 2: Client1 wants to read and Client2 wants to read in same RAM location at same time



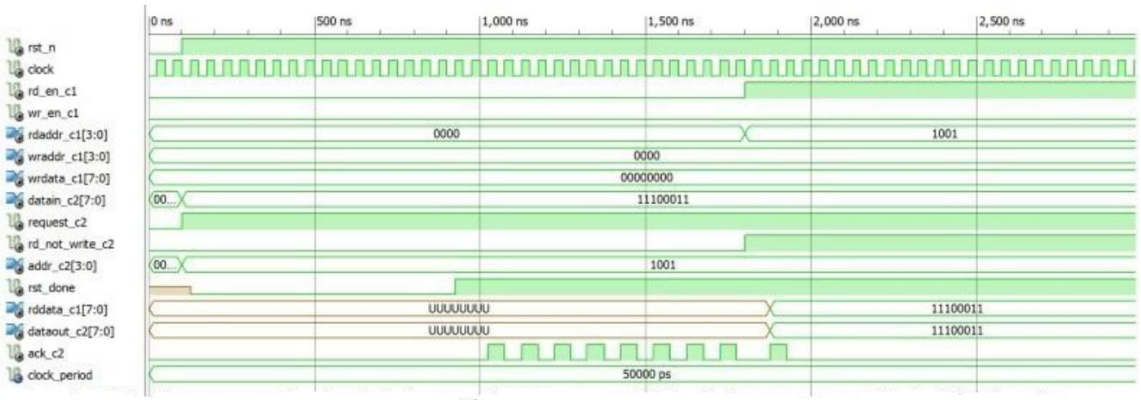
Case 3: Client1 wants to read and write in the same RAM location and Client2 also wants to read in the RAM location where Client1 has written at different time.



Case 4: Client1 wants to read and write in the same RAM location and Client2 also wants to read in the RAM location where Client1 has written at same time.



Case 5: Client2 wants to read and write in the same RAM location and Client1 also wants to write in the RAM location where Client2 has written at same time.



Case 6: Client2 wants to read and write in the same RAM location and Client1 also wants to read in the RAM location where Client2 has written at same time.

4.5.2. ANALYSIS OF THE TEST CASES

Case 1: Client1 wants to read and Client2 wants to read in same RAM location at different time.

During the first RAM_DEPTH cycle, Client1 perform write operation at the address location “1010”. Subsequently in the next RAM_DEPTH cycle, both Client1 and Client2 try to access the RAM to read the same RAM address location. Since it a priority based arbitration system, Client1 gets access to the RAM compared to Client2. It is only when Client1 does not require the read operation any longer that the system grants access to Client2 which reads from the same location. There is a time delay of 500 ns after which Client1 sets its RD_EN_C1 to active low signal. Since both access the same RAM location at different time, this does not affect the clients on the whole.

Case 2: Client1 wants to read and Client2 wants to read in same RAM location at same time.

The present case is similar to the previous test case differing only in the fact that here the scheme of priority based arbitration is clearly visible. During the second RAM_DEPTH cycle,

both the clients try to access the same RAM location at the same time but only Client1 gets access to the RAM. Since there is no output at ACK_C2 pin it bears testimony to our claims.

Case 3: Client1 wants to read and write in the same RAM location and Client2 also wants to read in the RAM location where Client1 has written at different time.

In this test case Client1 performs the write operation at the address location "1001". In the second RAM_DEPTH cycle, Client1 as it enjoys the higher priority among the two, can simultaneously perform the read and write operation. It writes the data "10100011" at the previous address location However as previously also discussed, the output at the RDDATA_C1 pin is the apparent output which comes from the temporary register located in the Arbiter itself which provides this output when client1 performs the read operation. Since Client2 access the system at a different time (i.e. after 500 ns), the arbiter grants permission to Client2 to read from the same location as Client1 is not using the read operation. This can be observed by the setting low of the RD_EN_C1 signal near 2500 ns mark in the waveform graph.

Case 4: Client1 wants to read and write in the same RAM location and Client2 also wants to read in the RAM location where Client1 has written at same time.

Client1 performs the write operation at the address location "1001". In the second RAM_DEPTH cycle, Client1 as it enjoys the higher priority among the two, can simultaneously perform the read and write operation. It writes the data "10100011" at the previous address location. However as previously also discussed, the output at the RDDATA_C1 pin is the apparent output which comes from the temporary register located in the Arbiter itself which provides this output when client1 performs the read operation. Since Client2 access the system at the same time, hence Client2 is unable to access the RAM at all.

Case 5: Client2 wants to read and write in the same RAM location and Client1 also wants to write in the RAM location where Client2 has written at same time.

During the first RAM_DEPTH cycle, Client2 performs the write operation at the address location "1001". The data written is "11100011". However in the second RAM_DEPTH cycle, since Client2 has a lower priority in the system, it can only read or write at any given time. The Arbiter only allows Client2 to read from the address location "1001". This can verified from the pulses at the output pins of ACK_C2.Moreover, since Client2 reads, the arbiter at a later time allows Client1 to write to the RAM at the same address location. Almost instantaneously, the data given is sent to the input pins of the RAM and is also reflected at the output pins of DATAOUT_C2 and RDDATA_C1, the reason for which had already been discussed. We observe that the write operation of Client2 during the second RAM_DEPTH cycle is omitted.

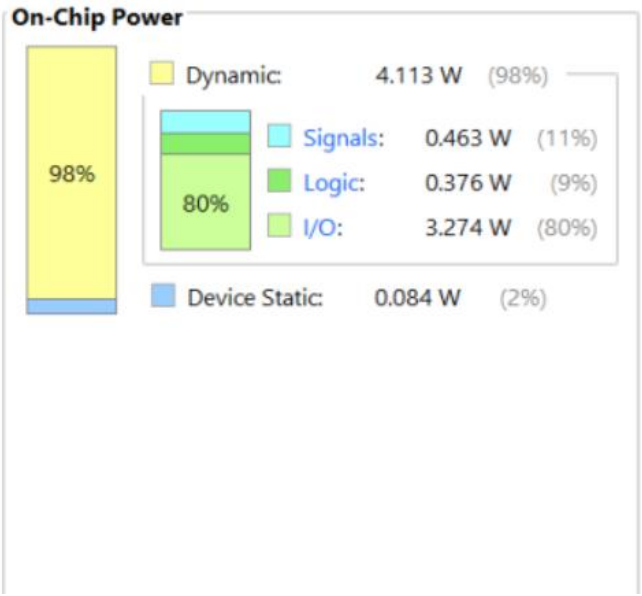
Case 6: Client2 wants to read and write in the same RAM location and Client1 also wants to read in the RAM location where Client2 has written at same time.

In this case, Client2 performs a write operation during the first RAM_DEPTH cycle. In the next cycle, one would observe that around 1700-1800 ns that Client2 performs the read operation on the address location "1010".This is more clear from the type of the output pulses at the output pin of ACK_C2. But almost within 25-50 ns, we see that the Arbiter grants the read operation to Client1 who enjoys a higher priority. The ACK_C2 signals stops giving any output. Since both Client1 and Client2 try to perform the read operation at the same time, it may be somewhat difficult to comprehend the change with the naked eye.

4.6. POWER & TIMING

Power estimation from Synthesized netlist. Activity derived from constraints files, simulation files or vectorless analysis. Note: these early estimates can change after implementation.

Total On-Chip Power:	**4.197 W**
Design Power Budget:	**Not Specified**
Process:	typical
Power Budget Margin:	**N/A**
Junction Temperature:	**46.0°C**
Thermal Margin:	39.0°C (7.8 W)
Ambient Temperature:	25.0 °C
Effective ϑJA:	5.0°C/W
Power supplied to off-chip devices:	0 W
Confidence level:	Low



POWER

Name	Slack	Levels	Routes	High Fanout	From	To	Total Delay	Logic Delay	Net Delay	Logic %	Net %	Requirement
Unconstrained Paths (1)												
(none) (10)												
Path 11	∞	3	3	2	dut2/TEMP_RD_DATA_reg[1]/C	DATAOUT_C2[1]	4.903	3.362	1.541	68.6	31.4	∞
Path 12	∞	3	3	2	dut2/TEMP_RD_DATA_reg[3]/C	DATAOUT_C2[3]	4.903	3.362	1.541	68.6	31.4	∞
Path 13	∞	3	3	2	dut2/TEMP_RD_DATA_reg[5]/C	DATAOUT_C2[5]	4.903	3.362	1.541	68.6	31.4	∞
Path 14	∞	3	3	2	dut2/TEMP_RD_DATA_reg[7]/C	DATAOUT_C2[7]	4.903	3.362	1.541	68.6	31.4	∞
Path 15	∞	3	3	2	dut2/TEMP_RD_DATA_reg[1]/C	RDDATA_C1[1]	4.903	3.362	1.541	68.6	31.4	∞
Path 16	∞	3	3	2	dut2/TEMP_RD_DATA_reg[3]/C	RDDATA_C1[3]	4.903	3.362	1.541	68.6	31.4	∞
Path 17	∞	3	3	2	dut2/TEMP_RD_DATA_reg[5]/C	RDDATA_C1[5]	4.903	3.362	1.541	68.6	31.4	∞
Path 18	∞	3	3	2	dut2/TEMP_RD_DATA_reg[7]/C	RDDATA_C1[7]	4.903	3.362	1.541	68.6	31.4	∞
Path 19	∞	3	3	2	dut2/TEMP_RD_DATA_reg[0]/C	DATAOUT_C2[0]	4.877	3.336	1.541	68.4	31.6	∞
Path 20	∞	3	3	2	dut2/TEMP_RD_DATA_reg[2]/C	DATAOUT_C2[2]	4.877	3.336	1.541	68.4	31.6	∞

SET UP TIME

Name	Slack ^1	Levels	Routes	High Fanout	From	To	Total Delay	Logic Delay	Net Delay	Logic %	Net %
Unconstrained Paths (1)											
(none) (10)											
Path 1	∞	1	1	17	dut2/TEMP_WR_DATA_reg[0]/C	dut2/TEMP_RD_DATA_reg[0]/D	0.305	0.141	0.164	46.2	53.8
Path 2	∞	1	1	17	dut2/TEMP_WR_DATA_reg[1]/C	dut2/TEMP_RD_DATA_reg[1]/D	0.305	0.141	0.164	46.2	53.8
Path 3	∞	1	1	17	dut2/TEMP_WR_DATA_reg[2]/C	dut2/TEMP_RD_DATA_reg[2]/D	0.305	0.141	0.164	46.2	53.8
Path 4	∞	1	1	17	dut2/TEMP_WR_DATA_reg[3]/C	dut2/TEMP_RD_DATA_reg[3]/D	0.305	0.141	0.164	46.2	53.8
Path 5	∞	1	1	17	dut2/TEMP_WR_DATA_reg[4]/C	dut2/TEMP_RD_DATA_reg[4]/D	0.305	0.141	0.164	46.2	53.8
Path 6	∞	1	1	17	dut2/TEMP_WR_DATA_reg[5]/C	dut2/TEMP_RD_DATA_reg[5]/D	0.305	0.141	0.164	46.2	53.8
Path 7	∞	1	1	17	dut2/TEMP_WR_DATA_reg[6]/C	dut2/TEMP_RD_DATA_reg[6]/D	0.305	0.141	0.164	46.2	53.8
Path 8	∞	1	1	17	dut2/TEMP_WR_DATA_reg[7]/C	dut2/TEMP_RD_DATA_reg[7]/D	0.305	0.141	0.164	46.2	53.8
Path 9	∞	1	1	8	dut2/nx_client_read_reg[2]/G	dut2/pr_client_read_reg[2]/D	0.331	0.175	0.156	52.8	47.2
Path 10	∞	1	1	8	dut2/nx_client_write_reg[1]/G	dut2/pr_client_write_reg[1]/D	0.331	0.175	0.156	52.8	47.2

HOLD TIME

PART C: CONCLUSION

General Comments on the Results:

The project has achieved success in designing and implementing an Arbiter for RAM, utilizing a priority system to manage access from various data sources. The primary function of the Arbiter is to allocate access privileges to RAM between Client 1 and Client 2, ensuring effective and equitable resource utilization.

Limitations and Challenges:

While there has been simulation and testing of some cases, the testbench's scope is still limited, unable to guarantee the Arbiter's completeness and comprehensiveness in all scenarios. There is a need to enhance the implementation of the testbench for the remaining blocks to verify and ensure the correctness of the entire system.

Development Suggestions:

- Expand the testbench to thoroughly assess the entire system under diverse conditions and different scenarios, particularly in boundary cases and error situations.
- Implement intricate test cases to ensure the Arbiter operates correctly and reliably in diverse environments.
- Integrate automatic testing mechanisms and performance evaluations to ensure the Arbiter meets speed and workload processing requirements.

These development suggestions aim to enhance the quality and reliability of the Arbiter, ensuring its stability and correctness in various usage conditions.

PART D: REFERENCES

1. Architectural Design of a RAM Arbiter_Sourangsu Banerji_Department of Electronics & Communication Engineering, RCC-Institute of Information Technology, Under West Bengal University of Technology, April, 2014
2. M. Weber, "Arbiters: Design Ideas and Coding Styles," Synopsys Users Group, Boston, 2001.
3. E. S. Shin, V. J. Mooney III, G. F. Riley, “Round-robin Arbiter Design and Generation,” Georgia Institute of Technology, Atlanta, GA, Technical Report GIT-CC-02-38,2002.
4. https://github.com/faizaan22/Arbiter-for-RAM-module/blob/main/Arbiter%20report.pdf
5. L. Fischer and Y. Pyatnychko, "FPGA Design for DDR3 Memory," WPI, Worcester, 2012.
6. Principles and practices of interconnection networks, William James Dally and Brian Towels, Morgan Kauffmann.