****

**HO CHI MINH CITY UNIVERSITY OF TECHNOLOGY AND EDUCATION**

**FACULTY OF INTERNATIONAL EDUCATION**

**🕯✡🕮🕮✡🕯**

**A logo of hands holding a book

Description automatically generated**

**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 2**](#_heading=h.30j0zll)

[1.1. Problem Statement: 2](#_heading=h.1fob9te)

[1.2. Problem Resolution: 3](#_heading=h.3znysh7)

[**PART B: MAIN CONTENT 4**](#_heading=h.2et92p0)

[2. RAM 6](#_heading=h.tyjcwt)

[2.1. BLOCK DIAGRAM OF RAM 6](#_heading=h.3dy6vkm)

[2.2. CODING 8](#_heading=h.1t3h5sf)

[2.3. SCHEMATIC OF RAM 9](#_heading=h.4d34og8)

[2.4. TESTCASES FOR RAM 10](#_heading=h.2s8eyo1)

[2.5. VERILOG TESTBENCH 10](#_heading=h.17dp8vu)

[3. ARBITER 16](#_heading=h.3rdcrjn)

[3.1. FSM 16](#_heading=h.26in1rg)

[3.2. ARBITER 26](#_heading=h.lnxbz9)

[4. RAM ARBITER 42](#_heading=h.35nkun2)

[4.1. PORT MAP IN BETWEEN RAM AND ARBITER 42](#_heading=h.1ksv4uv)

[4.2. CODING 42](#_heading=h.44sinio)

[4.3. SCHEMATIC OF RAM ARBITER 44](#_heading=h.2jxsxqh)

[4.4. TESTCASES FOR RAM ARBITER 44](#_heading=h.z337ya)

[4.5. VERILOG TESTBENCH 45](#_heading=h.3j2qqm3)

[4.6. POWER & TIMING 52](#_heading=h.1y810tw)

[**PART C: CONCLUSION 53**](#_heading=h.4i7ojhp)

[**PART D: REFERENCES 53**](#_heading=h.2xcytpi)

# **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.

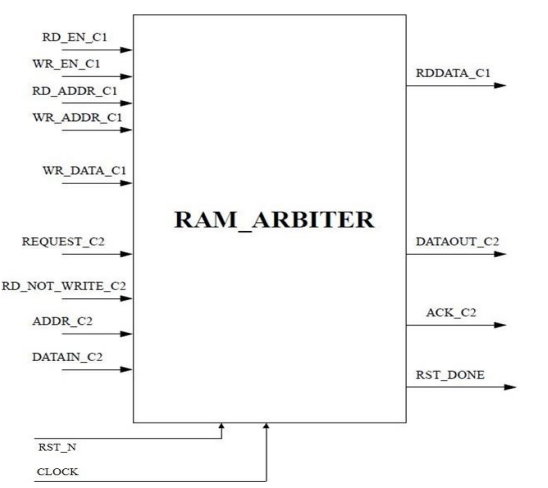
### 1.2. Problem Resolution:

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.

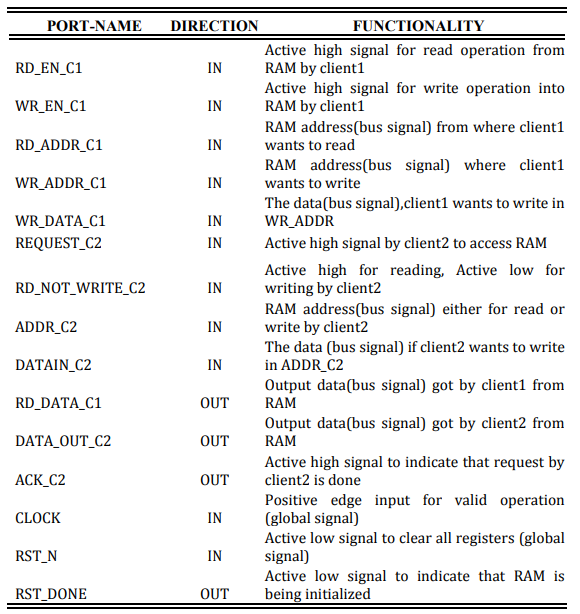
Through the research process, we will explore methods and algorithms to make informed decisions when multiple requests coincide. The goal is to build an Arbiter with flexible responsiveness, minimizing conflicts and ensuring that every system component has an opportunity for efficient memory access.

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 List and their Functionality**

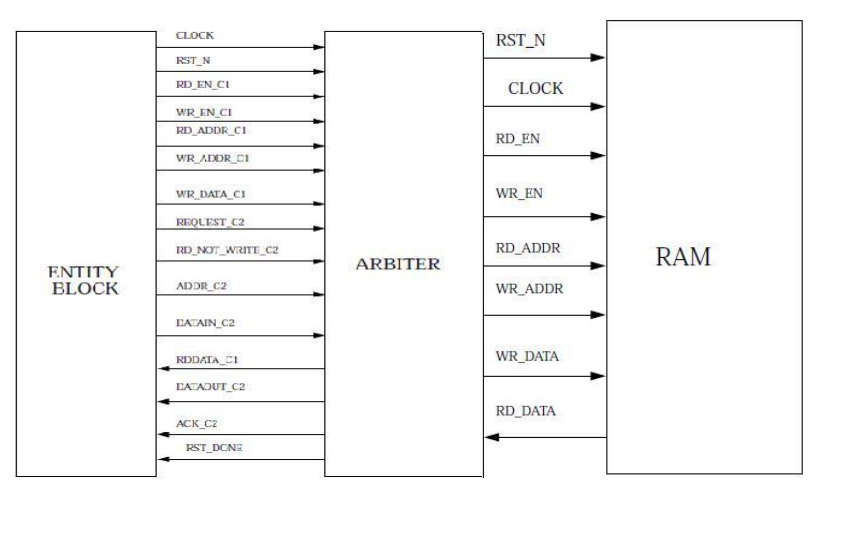
This is the top level block diagram which can interface with two clients (client1 and client2) with their respective input and output ports. But client1 has a higher priority than client2. So client1 can have the access of the RAM any time it wants. But if client2 wants to access the RAM then it must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

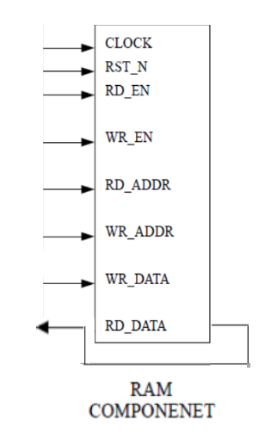
## 2. RAM

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:



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.**

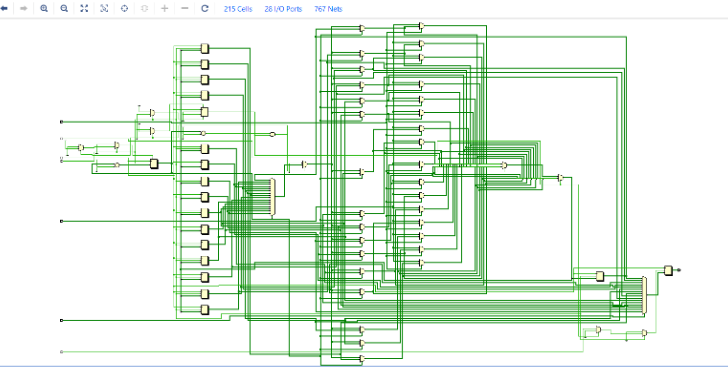
**end**

**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**

**`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;**

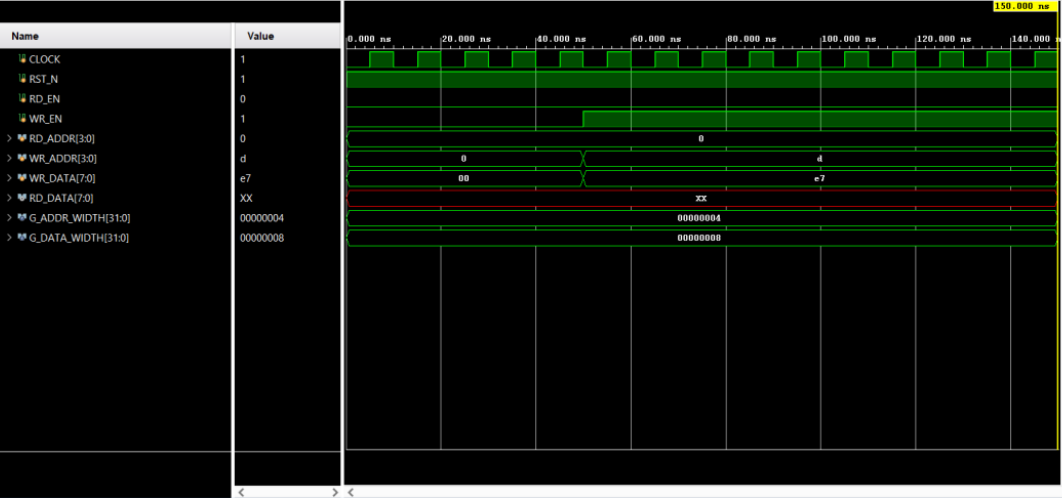
**// Finish simulation**

**$finish;**

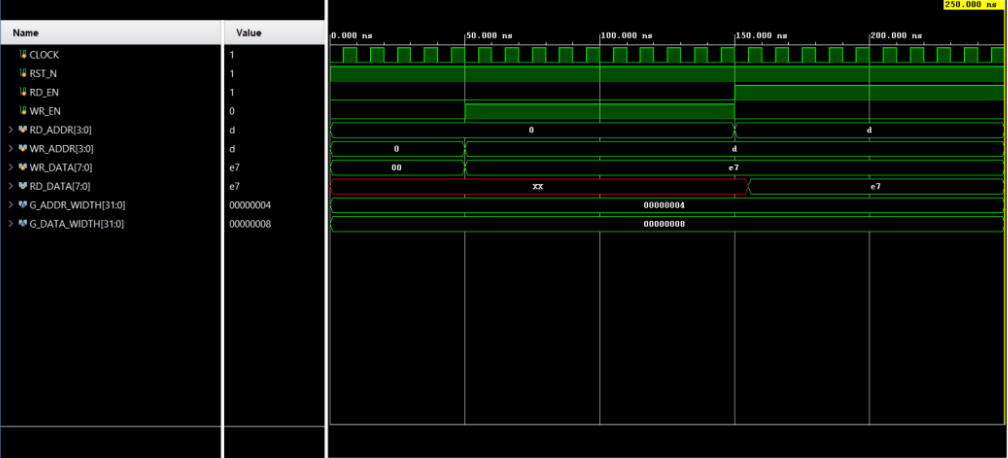
**end**

**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**

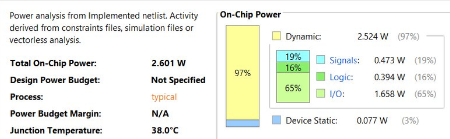
Similar to the previous condition, when the RST\_N pin is enabled within the first RAM cycle, data is read from the address location “1101 ( binary ) = d ( hex )”.

* **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

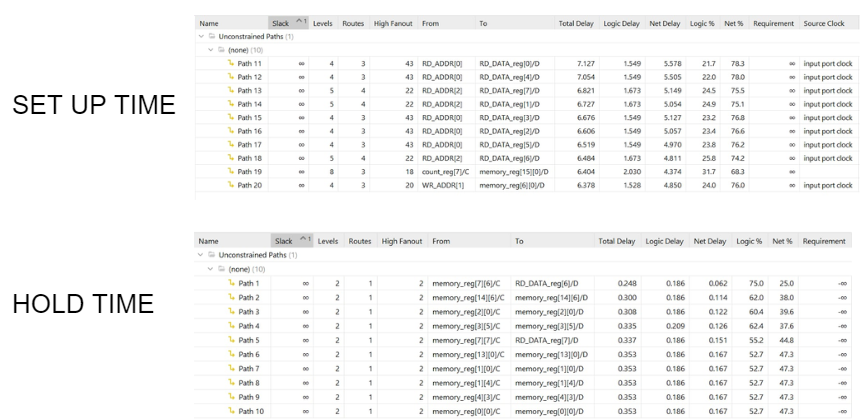
**Power:**

* 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.



**Setup Time and Hold Time:**

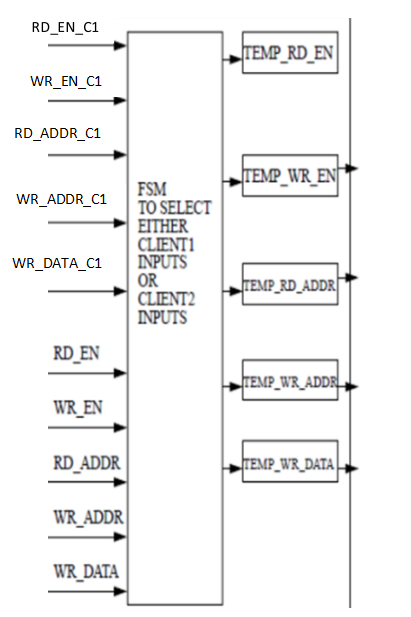
* Setup time is the time before a synchronous signal changes that the input data needs to be stable.
* Hold time is the time after a synchronous signal changes that the input data needs to be held stable.

****

## 3. ARBITER

### 3.1. FSM

#### 3.1.1. BLOCK DIAGRAM



#### 3.1.2. FSM DESIGN BLOCK

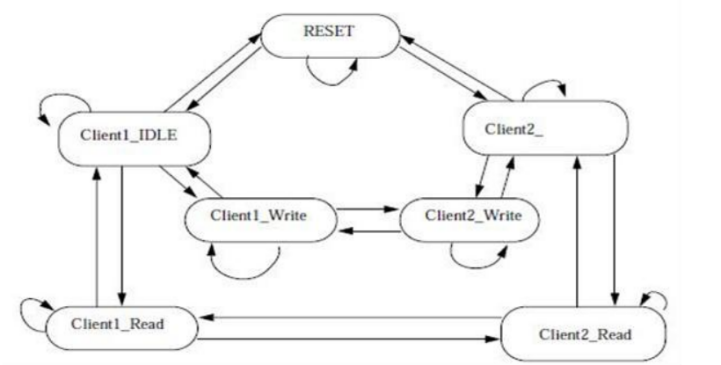
This FSM determines which client gets access to the specified RAM depending on the following conditions:

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

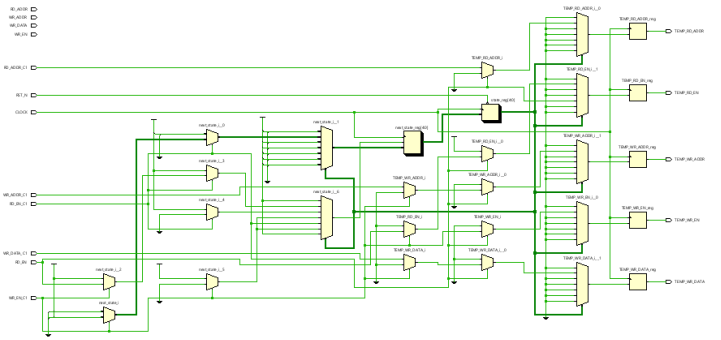
nx\_client\_write = client1\_write; //+ In client 2 write state and there is a write request from client 1, transition to client 1 write state

end

end

endmodule

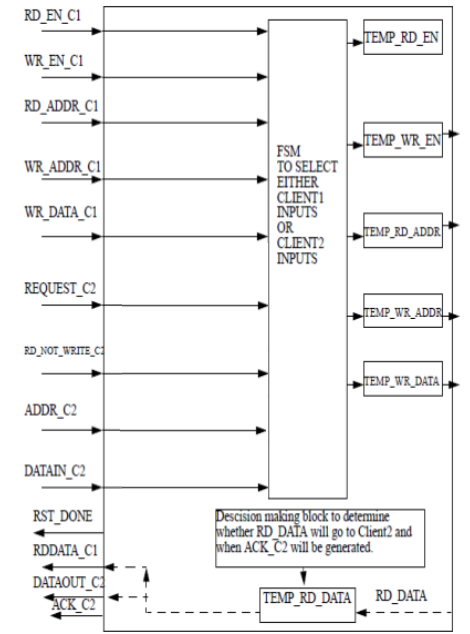
#### 3.1.4. SCHEMATIC



**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 :

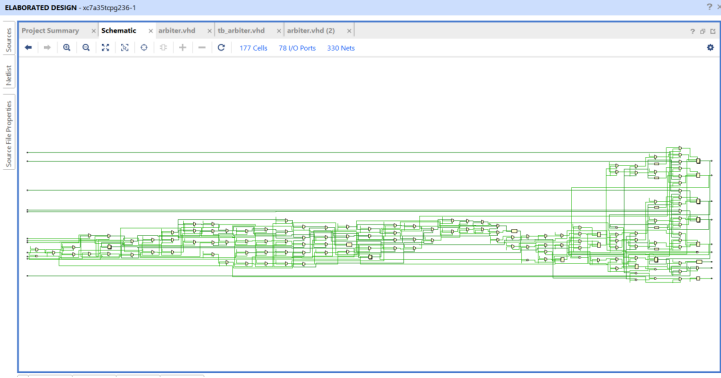
(REGISTERED\_DATA == 0 && ADDR\_CLASH == 1'b1 ) ? TEMP\_RD\_DATA :

(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;

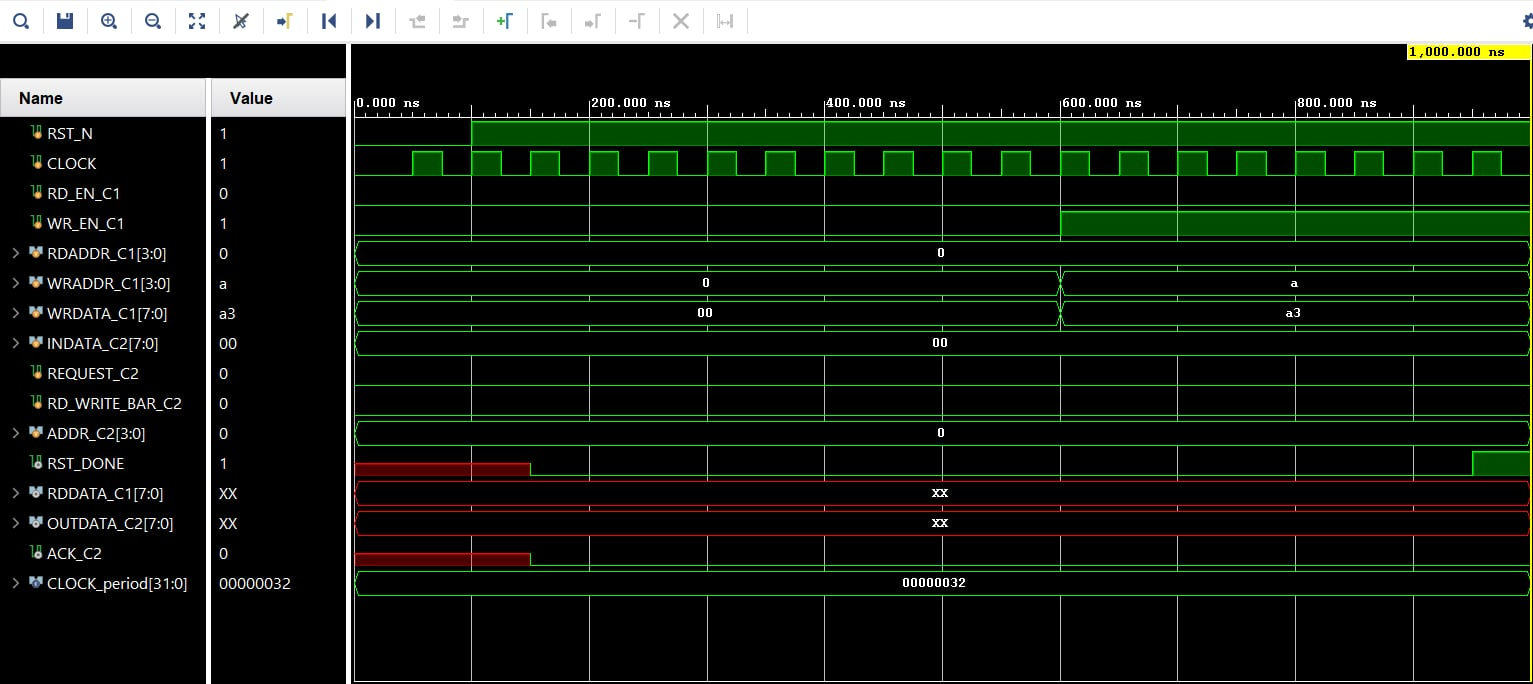
// End simulation after running test cases

$stop;

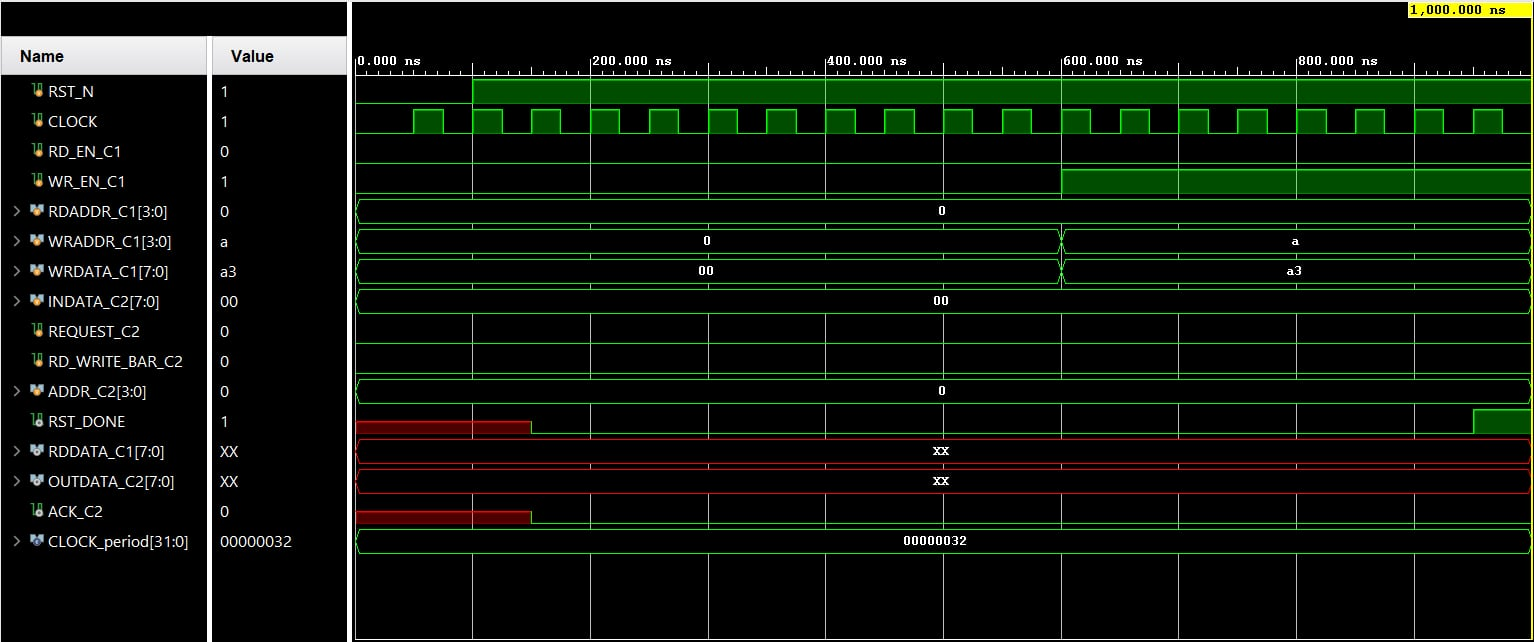
end

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.

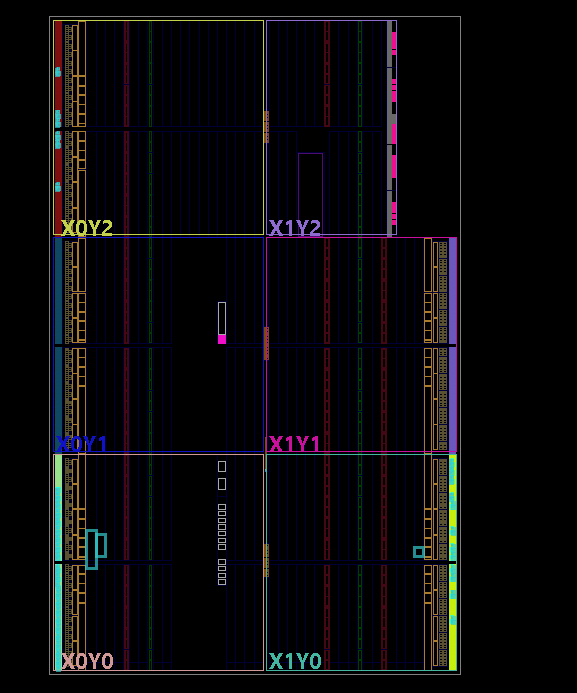
**Case3: Only Client2 wants to write.**

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.

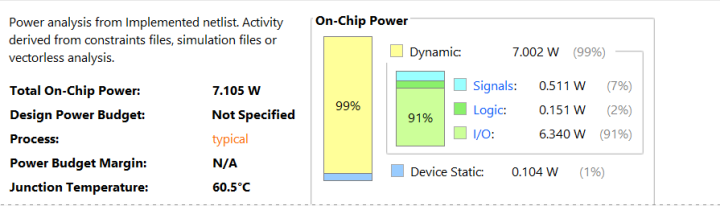
**Case 4: Only Client2 wants to read.**

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



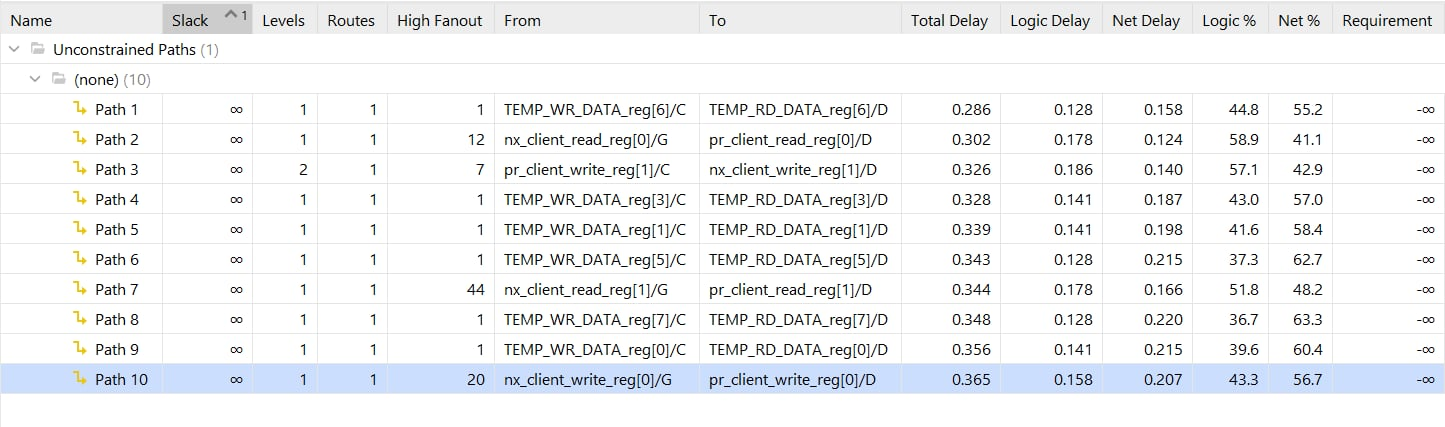
#### 3.2.7. Power & Timing



**POWER**



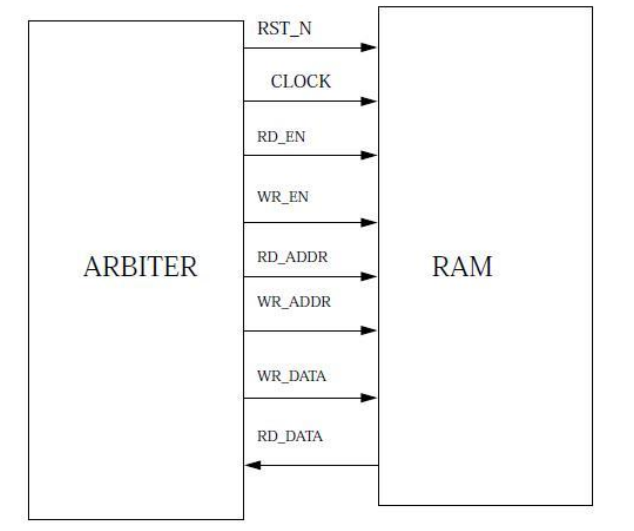
SET UP TIME



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),**

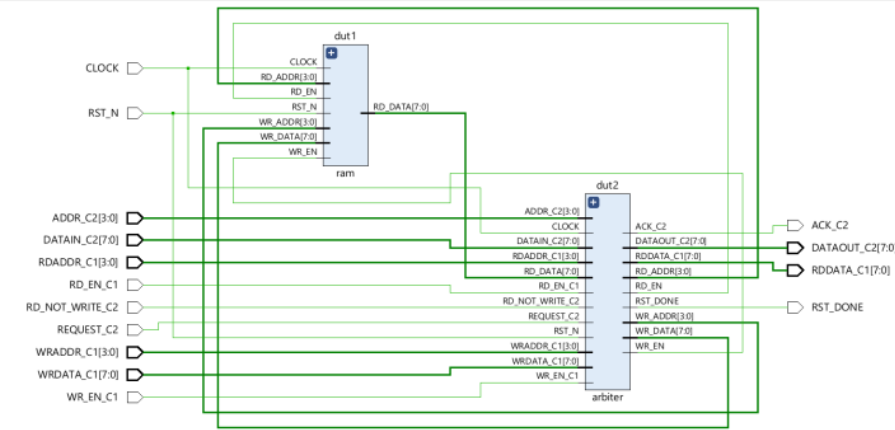
**.ACK\_C2(ACK\_C2),**

**.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;**

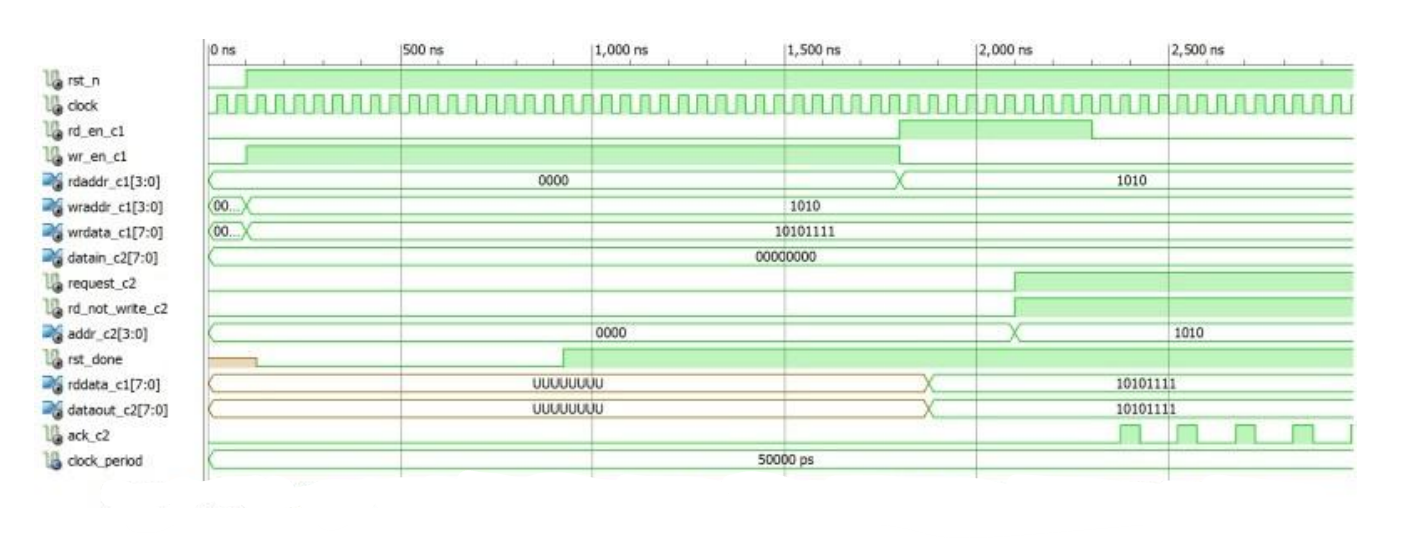
**#200 ADDR\_C2 = 4'b1001;**

**#200 RDADDR\_C1 = 4'b1001;**

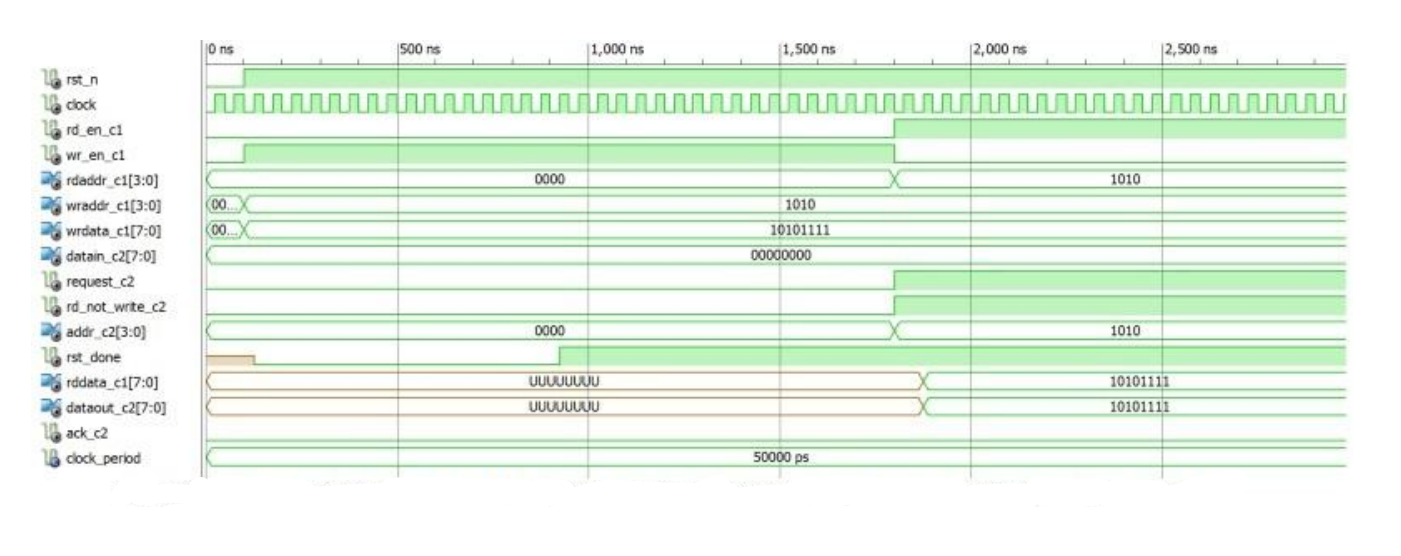
**end**

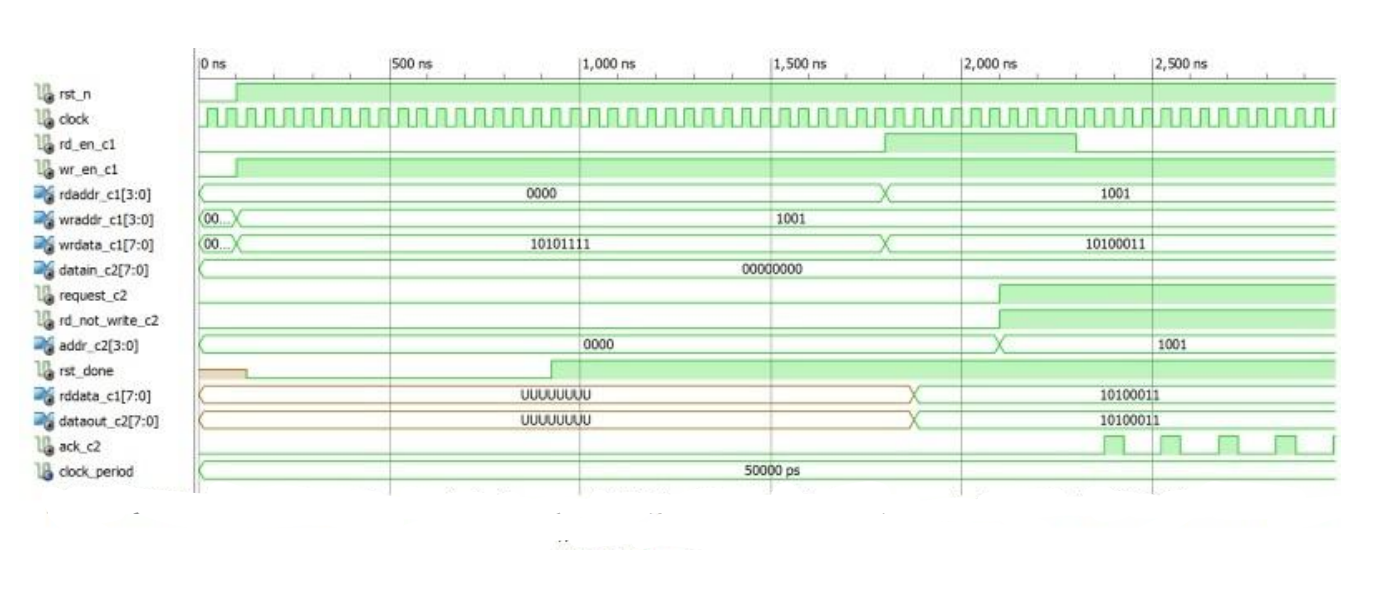
**endmodule**

**WAVEFORM**



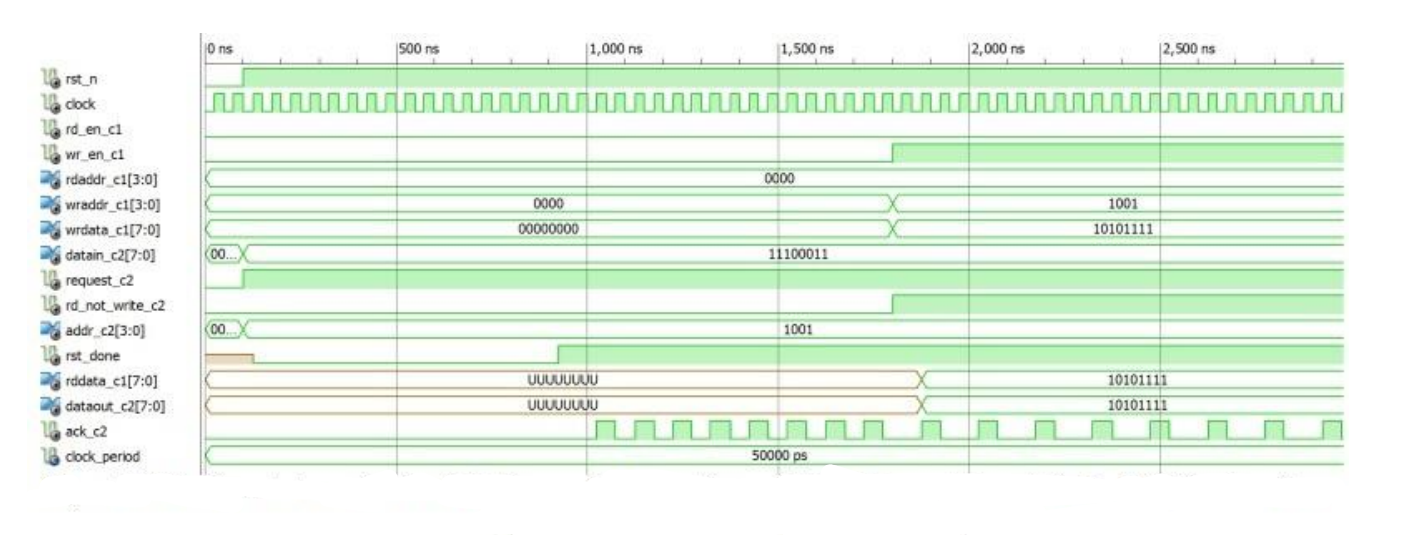
**Case 1: Client1 wants to read and Client2 wants to read in same RAM location at different time.**

****

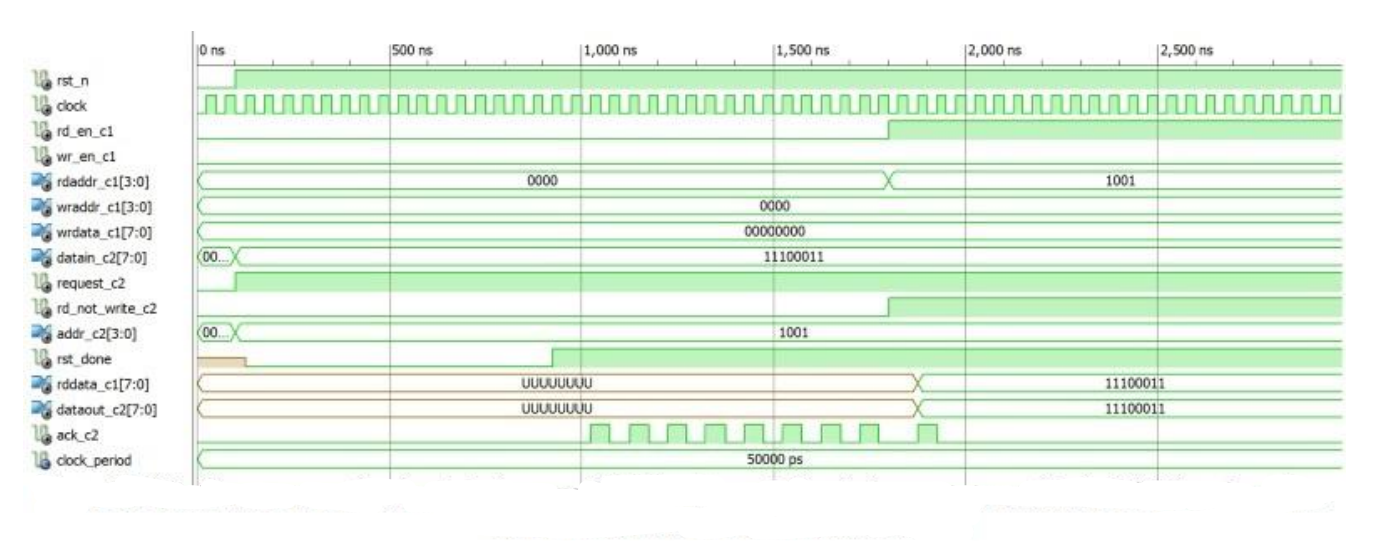
**Case 2: Client1 wants to read and Client2 wants to read in same RAM location at same time****Case 3: Client1 wants to read and write in the same RAM location and Client2 also wants to read in the RAM location where Client1 has written at different time.**



**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.

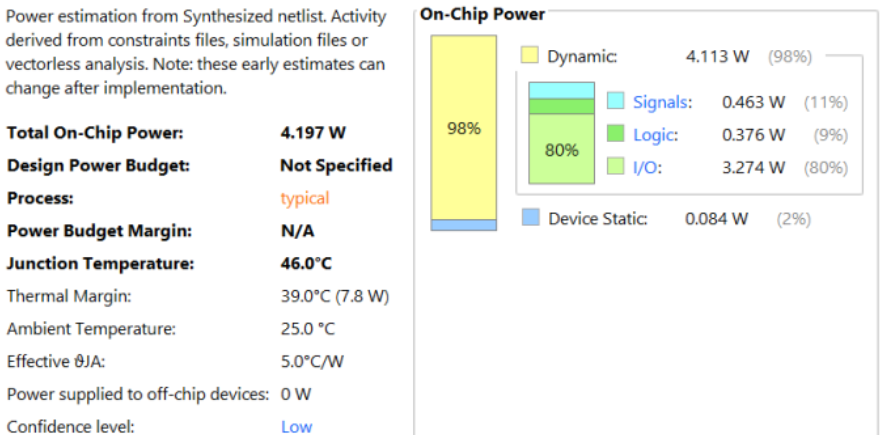
**Case 5: Client2 wants to read and write in the same RAM location and Client1 also wants to write in the RAM location where Client2 has written at same time.**

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.

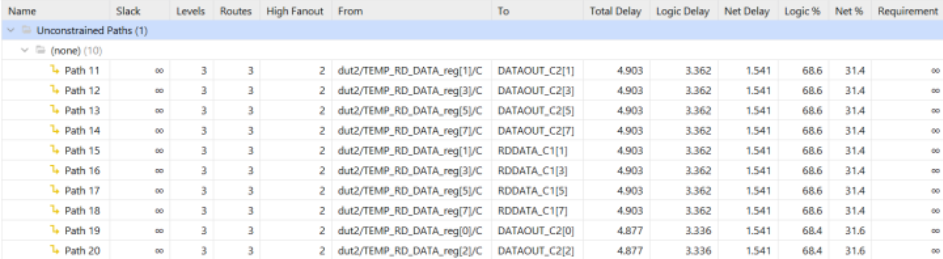
**Case 6: Client2 wants to read and write in the same RAM location and Client1 also wants to read in the RAM location where Client2 has written at same time.**

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.

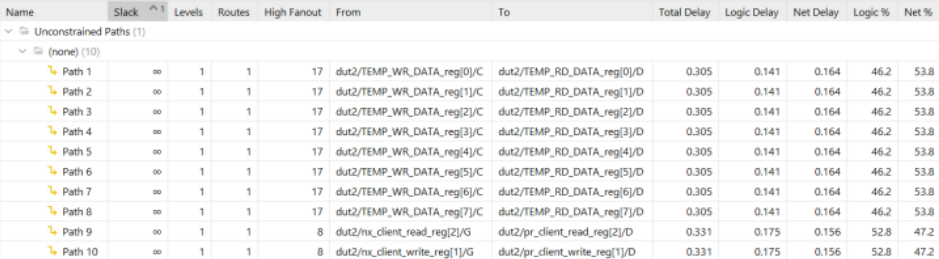
### 4.6. POWER & TIMING



**POWER**



**SET UP TIME**



**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.