DEPARTMENT OF ELECTRICAL ENGINEERING,
IIT BOMBAY

A REPORT

ON

An Arbiter for RAM module

July'21

AUTHOR'S NAME

Kanak Vjay (203070050)

Mohd. Faizaan Qureshi (203070062)

Introduction

Memory modules are standardised to be used with one system accessing it (i.e. store or access data). More complex memory modules would be accessed through a memory controller and these are also designed for one system. If we have multiple systems and all of them want to access a single memory module then there must be something that allows them access the memory without overriding or corrupting the access from the others. So we use arbiter for that which controls the flow of traffic into the memory controller. The arbiter is a decision making system that choose which system to grant the access to the shared resource on the other side of arbiter. Here that shared resource is memory. In this project, regular RAM module is designed to be used as a standard memory i.e. to work with a single system. Also, an arbiter is also designed that allows more than one system (in this project we took 2 devices) to use a single RAM module without any conflicts, in a synchronized manner. In this project, arbiter works on basis of priority. Also, there could be problems of having same address which is solved.

RAM (Random Access Memory)

It is a form of computer data storage. As the name suggests, we can access randomly any location of memory just by giving it an address. A random-access memory device allows data items to be read and written in roughly the same amount of time regardless of the order in which data items are accessed. In contrast, with other direct-access data storage media such as hard disks, CD-RWs, etc, the time required to read and write data items varies significantly depend on their physical locations on the recording medium, due to mechanical limitations such as media rotation speeds and arm movement delays. Today RAM is widely used in computers and other electronics as a way to access and store data. However, RAM is volatile memory and will only retain data as long as power is on. Once the system loses power, it loses any data stored in memory.

Static Random Access Memory (SRAM)) is a variation of RAM. SRAM is designed to fill two needs, first is to provide a direct interface to CPUs at speeds unattainable by DRAMs and secondly, it replaces DRAMs in systems that require very low power consumption. SRAM performs very well in low power applications due to the nature of the device. SRAM cells are comprised of six MOSFETs.

Other is Dynamic Random Access Memory (DRAM). As with SRAM, DRAM fundamentally holds onto the information of individual bits, but unlike SRAM, it is designed with capacitors along with transistors. The number of transistors is reduced to one in DRAM making it fundamentally simpler than SRAM. However since capacitors lose their charge over time, a refresh is needed to maintain stored data, which increases power usage due to the voltage of the capacitors The inability to maintain information without a refresh is why DRAM is considered dynamic as opposed to its “static” counterpart.

Arbiter

There exists many systems which has lot of requesters accessing a common resource. In our case, that shared resource is memory. An arbiter is needed to control the flow of traffic between the requestors and shared resource.

It is decision taking system which decides how the resource is allocated amongst the requesters. Internal logic of arbiter determines whom to give access and when. There are many arbitration schemes exist. These include round robin, FIFO, priority, dynamic priority. In this project, we have made arbiter based on priority.

The priority method grant access to whichever system has the highest priority. Let's say we have device1, which has highest priority. So, when it requests access to memory it is given it. Device2 will only receive resources if and when Device1 is no longer requesting.

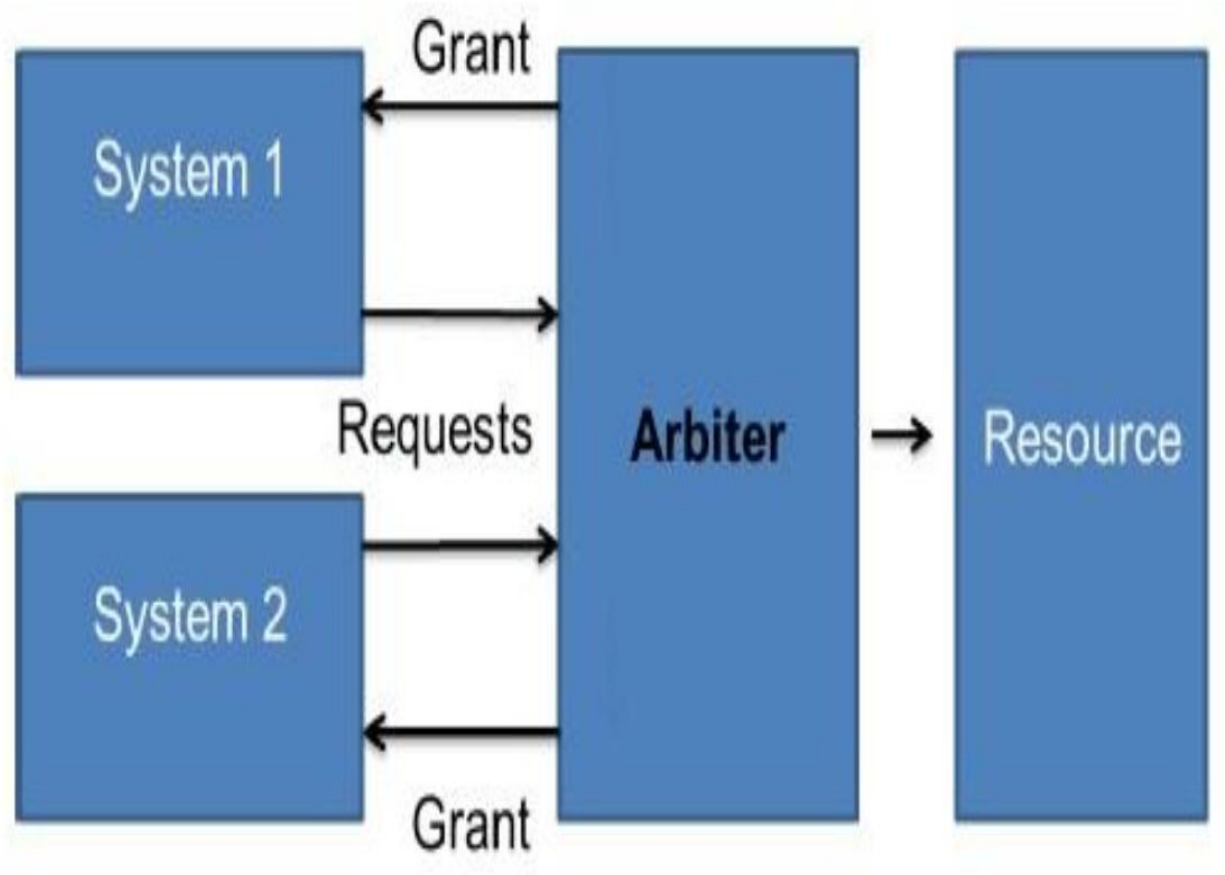


Fig. Block diagram of and two requesters accessing shared resource

RAM with Arbiter

A RAM arbiter maintain requests in order to ensure memory is holding accurate data.

Top Level Block Diagram

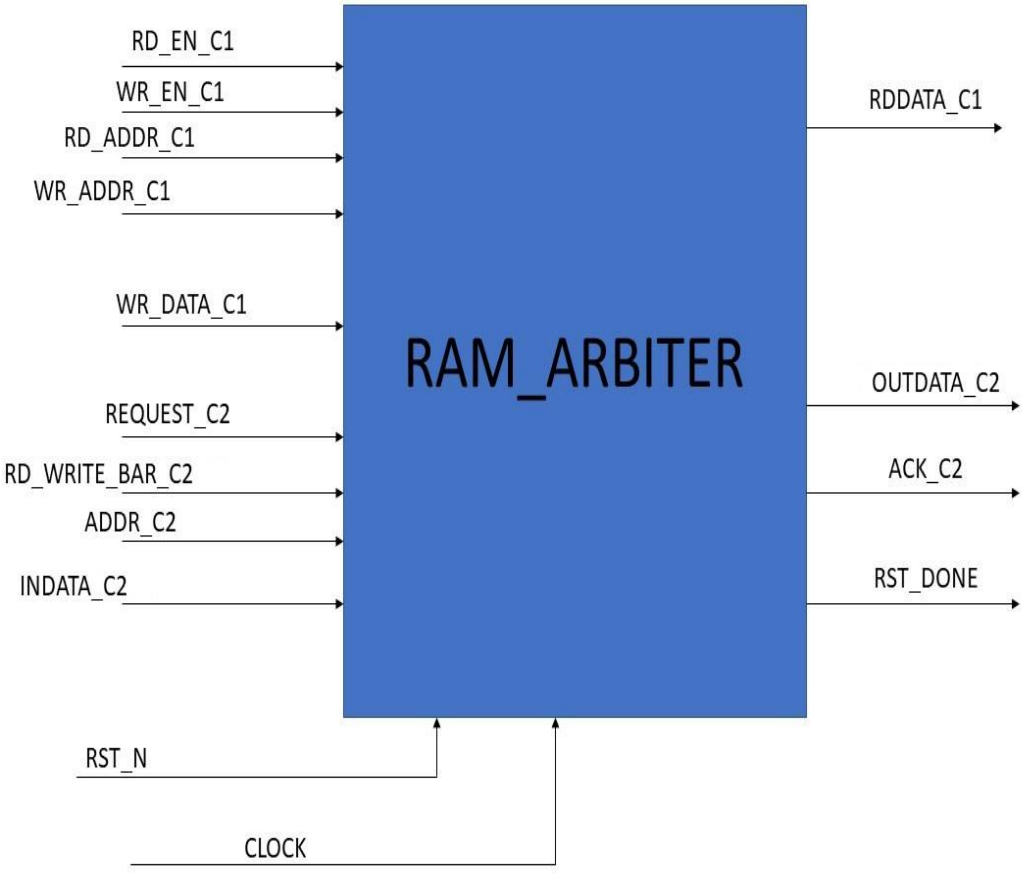


Fig. Block Diagram of RAM Arbiter

This is top level module which can interface with 2 devices (say clients) client1 and client2 with their respective input and output ports. Here we are making the priority of client1 high. So, client1 can have the access of the RAM any time it wants. But if client2 wants to access the RAM then it first give request then following things will occur :

1) If client1 is writing only not reading then client2 can access the RAM for reading.
2) If client1 is reading only not writing then client2 can access the RAM for writing.

3)) If client1 is reading and writing both, then client2 can't get access of the RAM.

4)) If client1 is sitting idle, doing nothing, then client2 can get access either for reading or writing.

FSM of the Arbiter

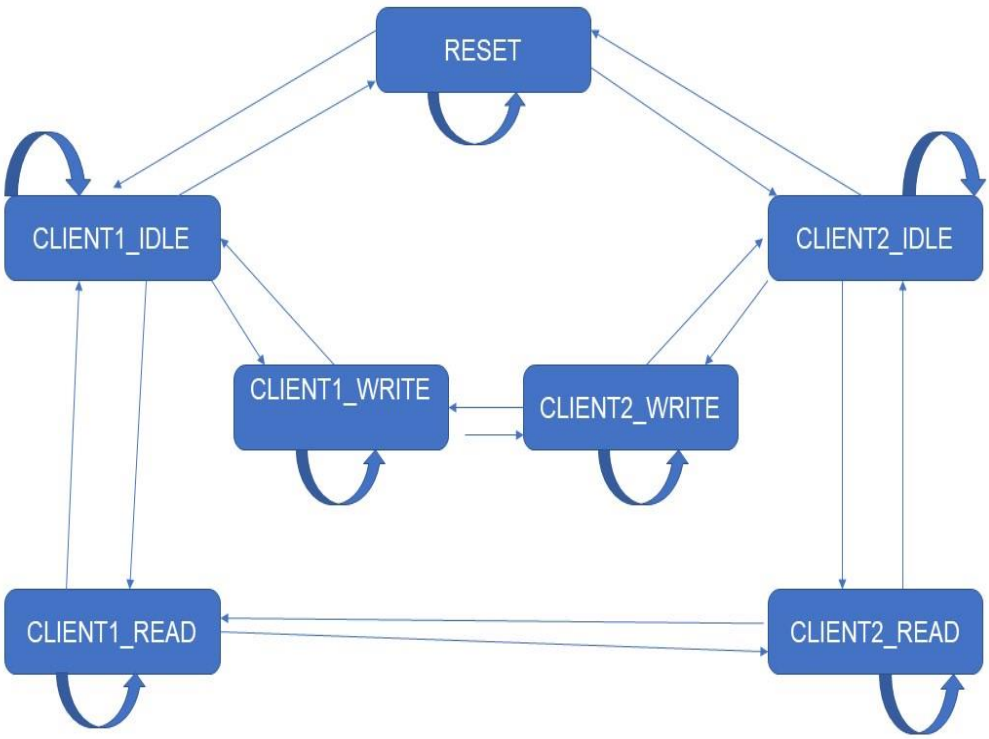
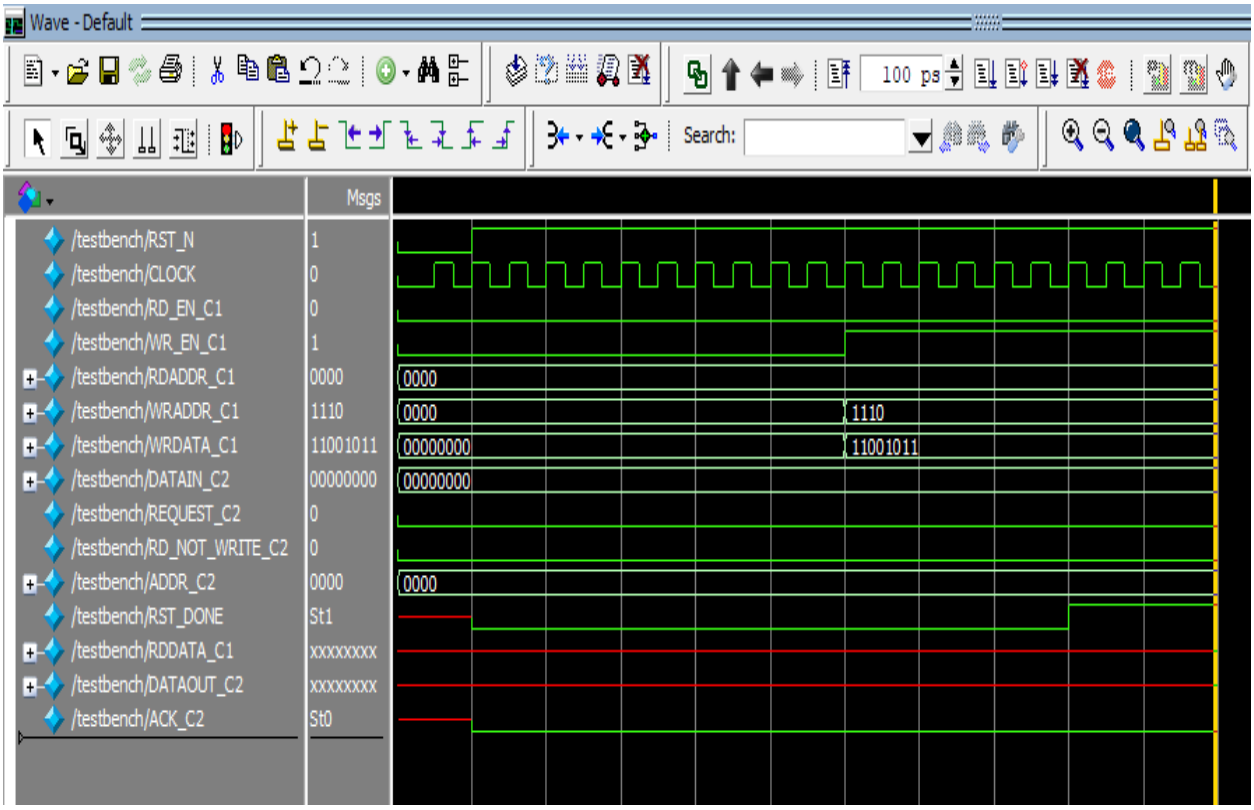


Fig. State diagram of Arbiter

Test Cases for RAM Arbiter

Case1 = Only Client1 wants to write

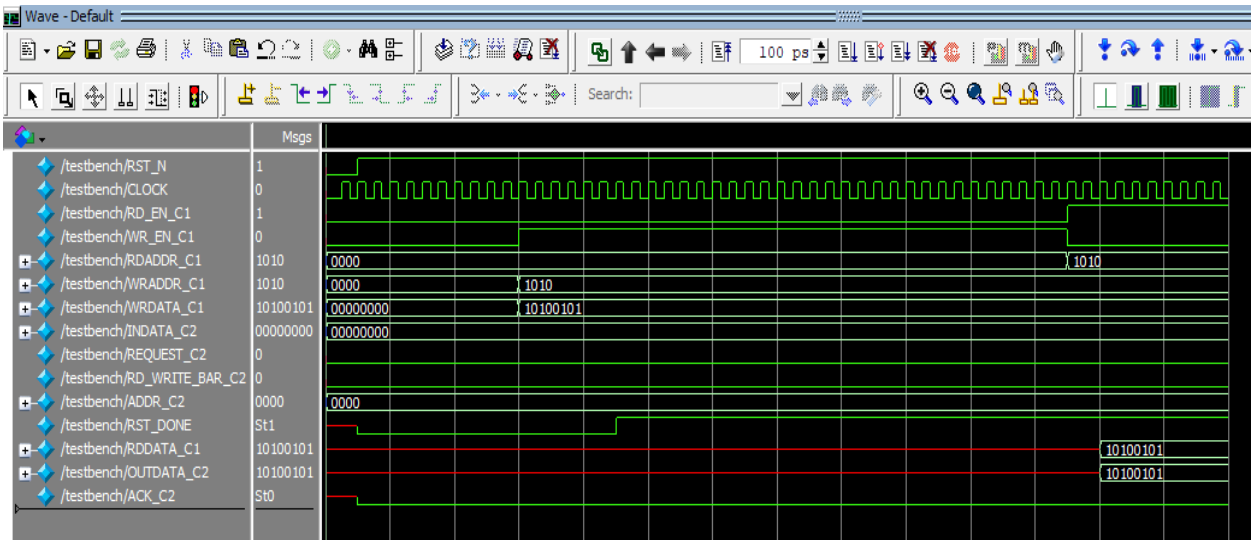
```
RST_N = 1;
#500
WR_EN_C1 = 1;
WRADDR_C1 <=4'b1110;
WRDATA_C1 <=8'b111001011;
```



Case2 = Only Client1 wants to read

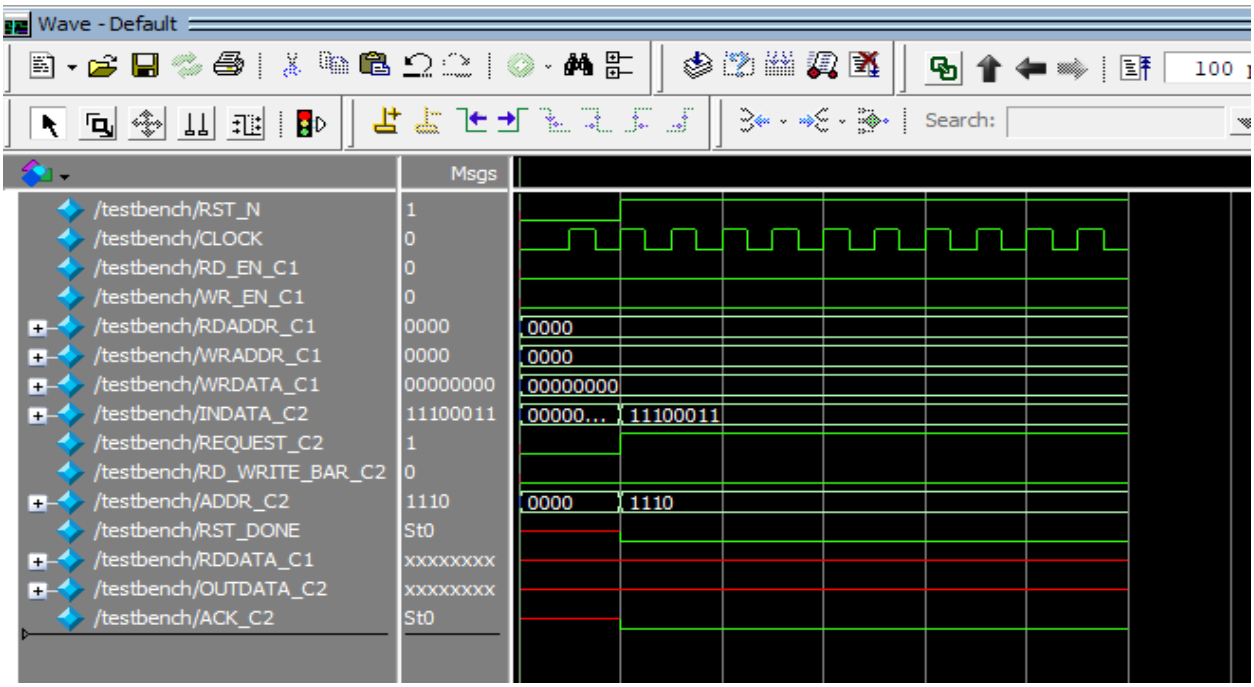
```
RST_N = 1;
#500
WR_EN_C1 = 1'b1;
WRADDR_C1 = 4'b1010;
WRDATA_C1 = 8'b10100101;
#1700
WR_EN_C1 = 1'b0;
RD_EN_C1 = 1'b1;
RDADDR_C1 = 4'b1010;
```

Since memory has not any data at the starting, so we are writing data long time before (equal to no of cycles needed to traversing the whole RAM) at that location from which we want to read



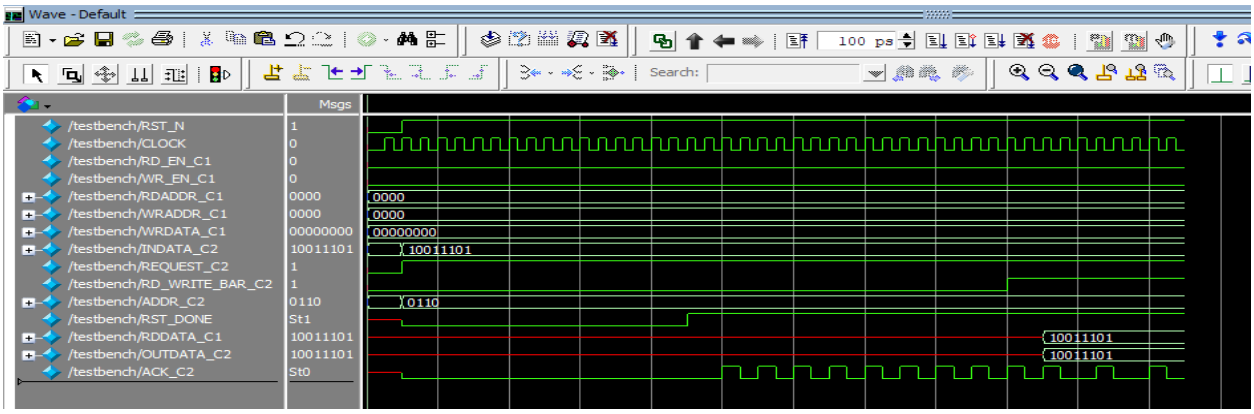
Case 3 = Only Client2 wants to write

```
RST_N =1'b1;
WR_EN_C1 = 1'b0;
REQUEST_C2 = 1'b1;
RD_WRITE_BAR_C2 = 1'b0;
ADDR_C2 =4'b1110;
INDATA_C2 =8'b11100011;
```



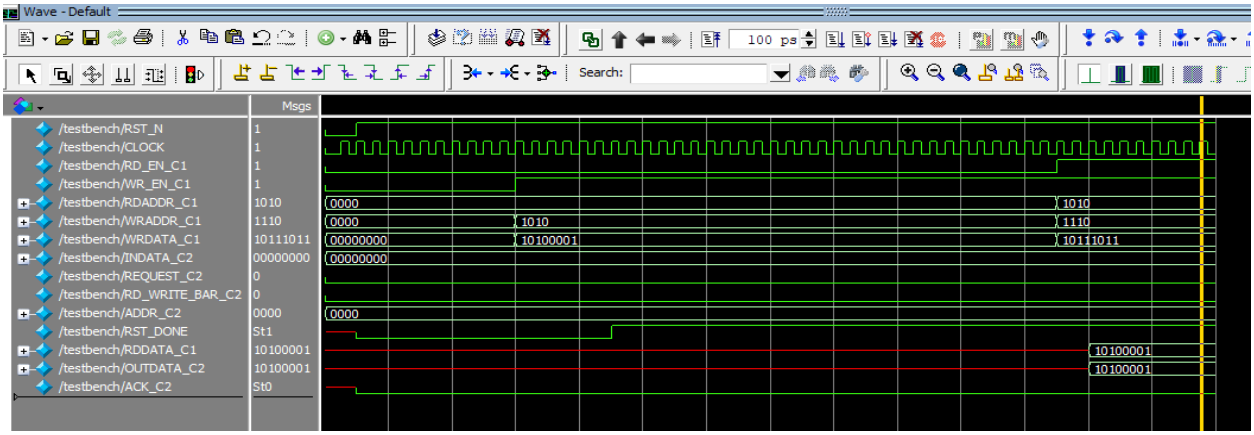
Case 4 = Only Client2 wants to read

```
RST_N = 1;
WR_EN_C1 = 0;
REQUEST_C2 = 1;
RD_WRITE_BAR_C2 = 0;
ADDR_C2  = 4'b0110;
INDATA_C2 = 8'b10011101;
#1700
WR_EN_C1 = 0;
RD_WRITE_BAR_C2 = 1;
ADDR_C2 = 4'b0110;
```



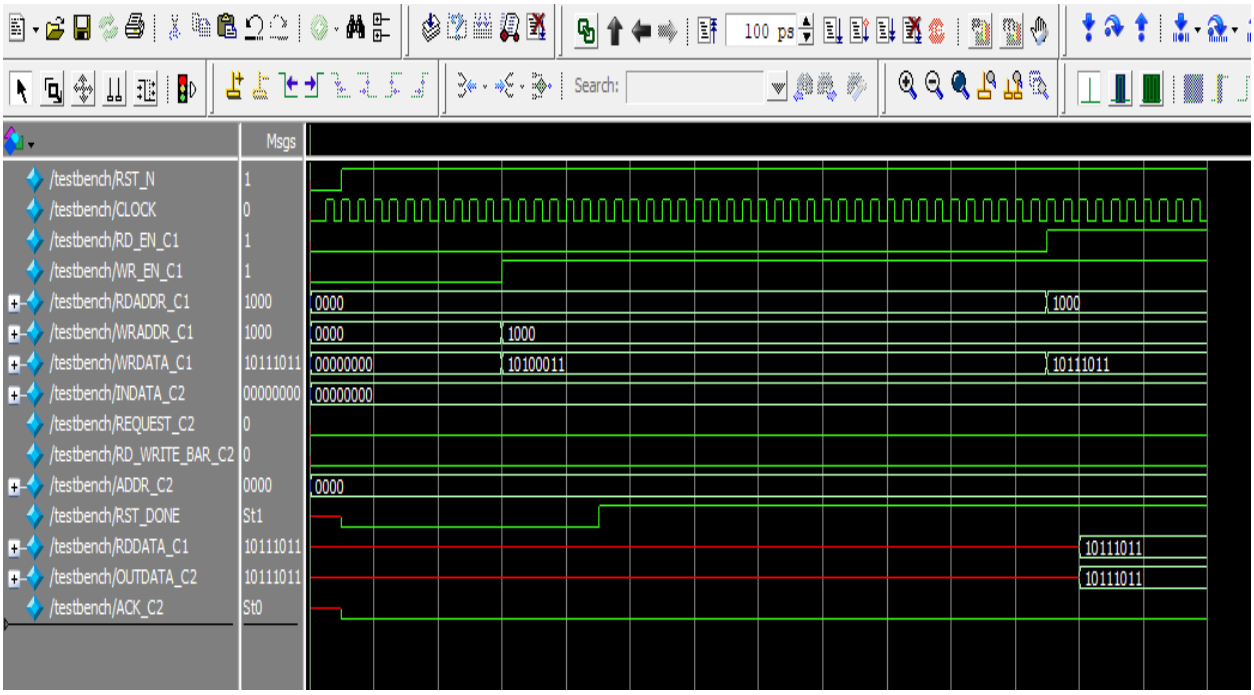
Case 5 = Client1 wants to read and write in different RAM location at same time

```
RST_N = 1;
#500
WR_EN_C1 = 1;
WRADDR_C1 = 4'b1010;
WRDATA_C1 = 8'b10100001;
#1700
RD_EN_C1 = 1;
RDADDR_C1 = 4'b1010;
WRADDR_C1 = 4'b1110;
WRDATA_C1 = 8'b10111011;
```



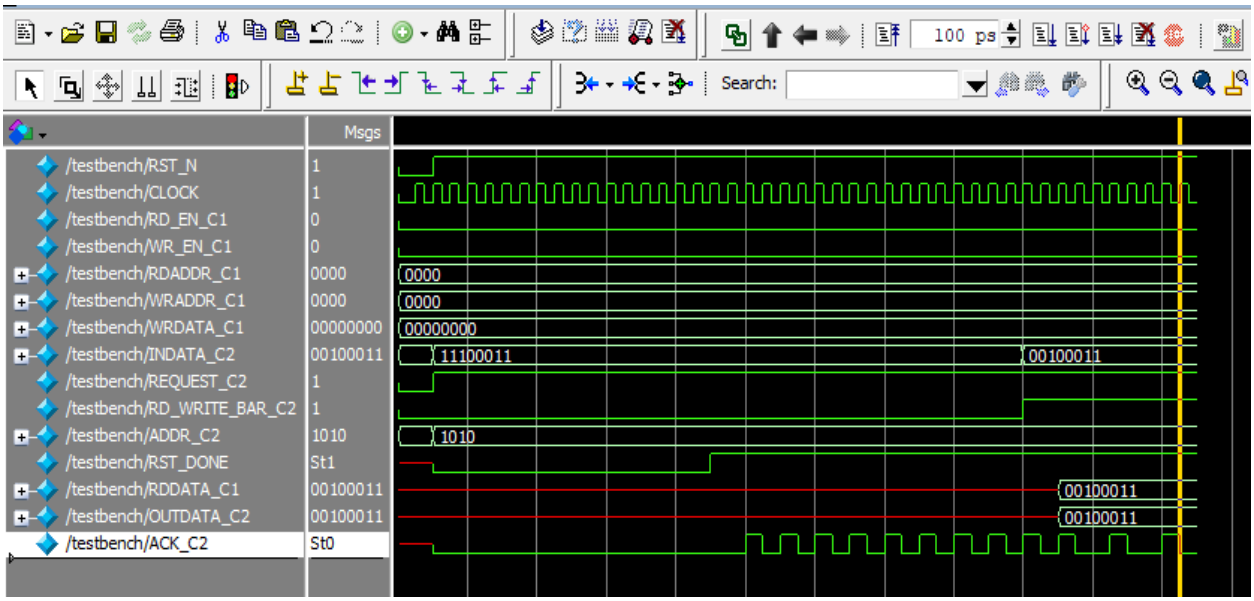
Case 6 = Client1 wants to read and write in same RAM location at same time

```
RST_N = 1;
#500
WR_EN_C1 = 1;
WRADDR_C1 = 4'b1000;
WRDATA_C1 = 8'b10100011;
#1700
RD_EN_C1 = 1;
RDADDR_C1 = 4'b1000;
WRADDR_C1 = 4'b1000;
WRDATA_C1 = 8'b10111011;
```



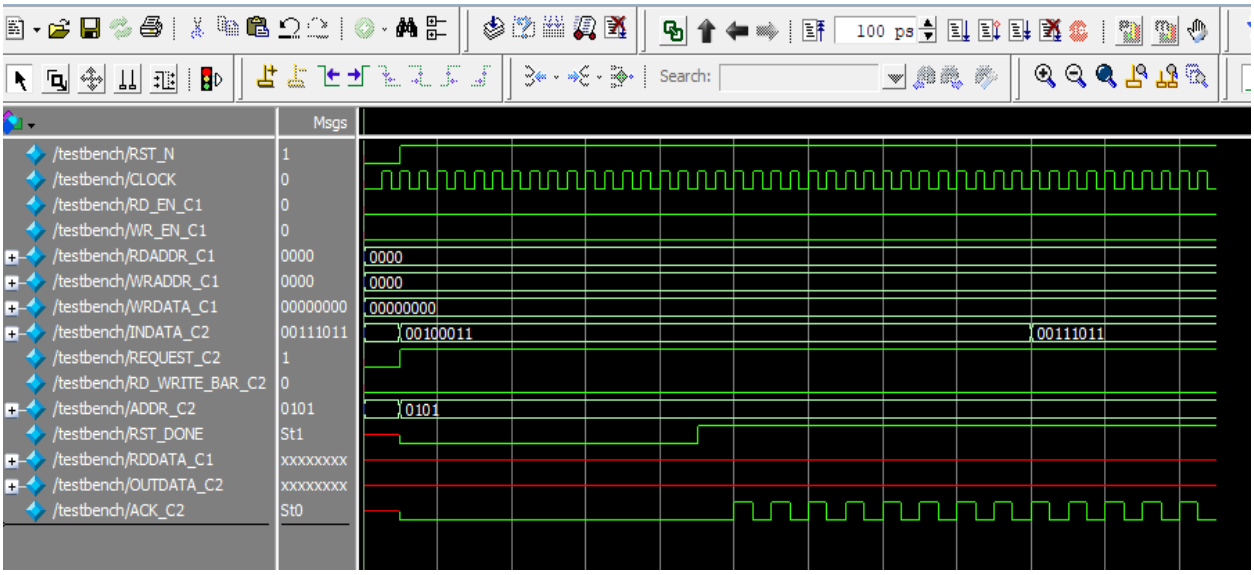
Case 7 = Client2 wants to read and write in different RAM location at same time

```
RST_N = 1;
WR_EN_C1 = 0;
RD_EN_C1 = 0;
REQUEST_C2 = 1;
RD_WRITE_BAR_C2 = 0;
ADDR_C2  = 4'b1010;
INDATA_C2 = 8'b11100011;
#1700
RD_WRITE_BAR_C2 = 0;
ADDR_C2 = 4'b1001;
INDATA_C2 = 8'b00100011;
RD_WRITE_BAR_C2 = 1;
ADDR_C2 = 4'b1010;
```



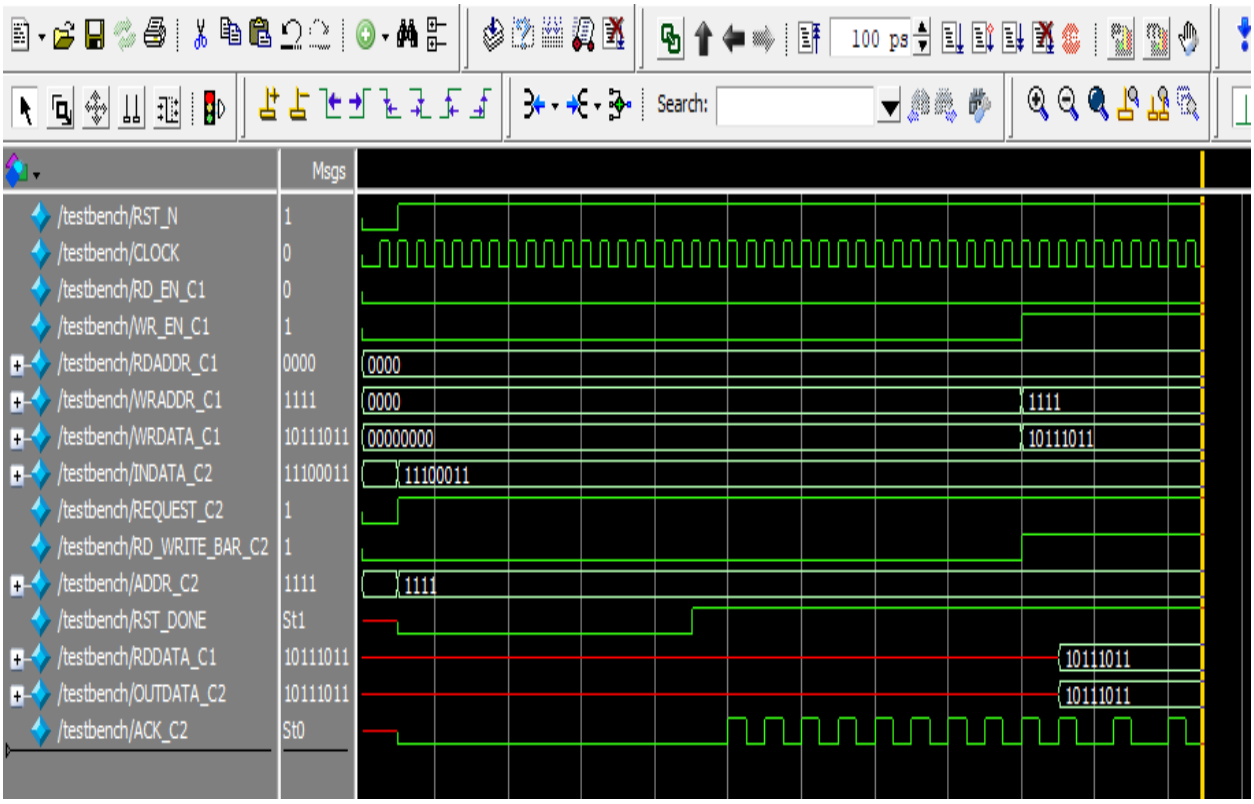
Case 8 = Client2 wants to read and write in same RAM location at same time

```
RST_N = 1;
WR_EN_C1 = 0;
RD_EN_C1 = 0;
REQUEST_C2 = 1;
RD_WRITE_BAR_C2 = 0;
ADDR_C2 = 4'b0101;
INDATA_C2 = 8'b00100011;
#1700
RD_WRITE_BAR_C2 = 1;
ADDR_C2 = 4'b0101;
RD_WRITE_BAR_C2 = 0;
ADDR_C2 = 4'b0101;
INDATA_C2 = 8'b00111011;
```



Case 9 = Client1 wants to write and client2 wants to read in same RAM location at same time

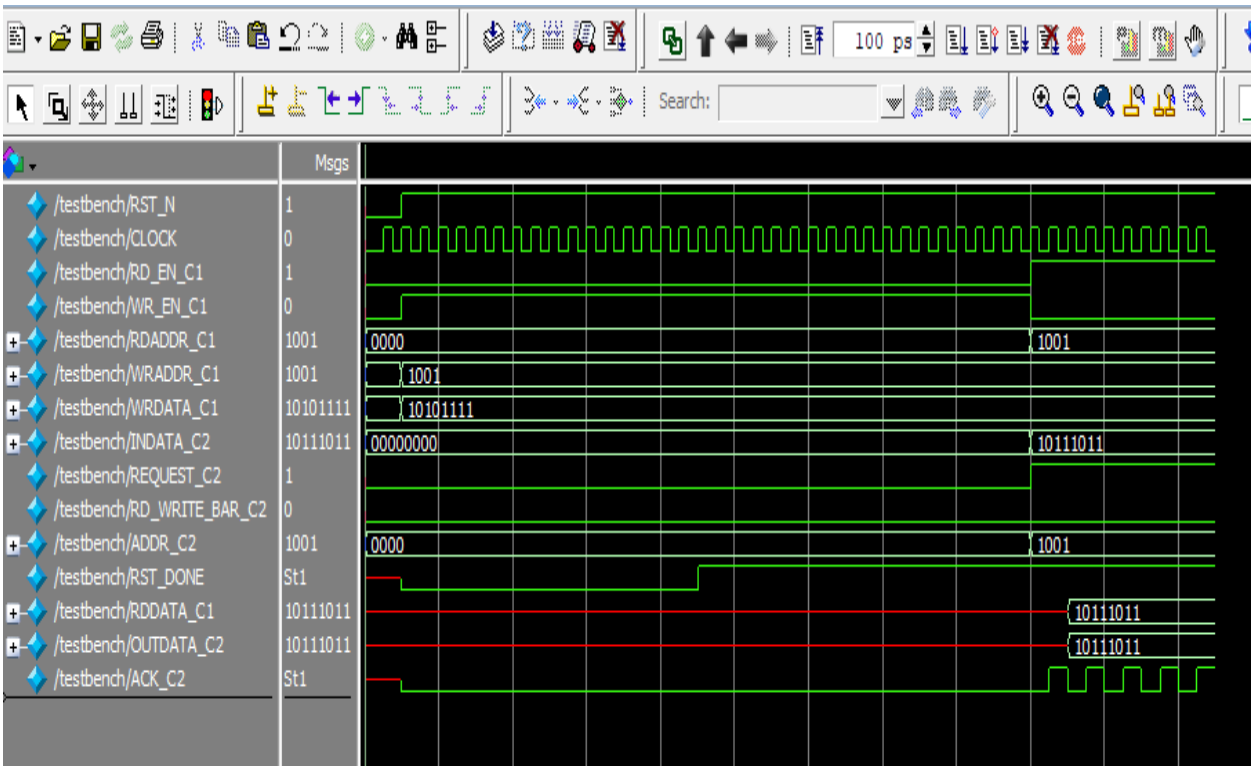
```
RST_N = 1;
WR_EN_C1 = 0;
REQUEST_C2 = 1;
RD_WRITE_BAR_C2 = 0;
ADDR_C2 = 4'b1111;
INDATA_C2 = 8'b11100011;
#1700
WR_EN_C1 = 1;
RD_EN_C1 = 0;
WRADDR_C1 = 4'b1111;
WRDATA_C1 = 8'b10111011;
REQUEST_C2 = 1;
RD_WRITE_BAR_C2 = 1;
ADDR_C2 = 4'b1111;
```



Case 10 = Client1 wants to read and Client2 wants to write in same RAM location at same time

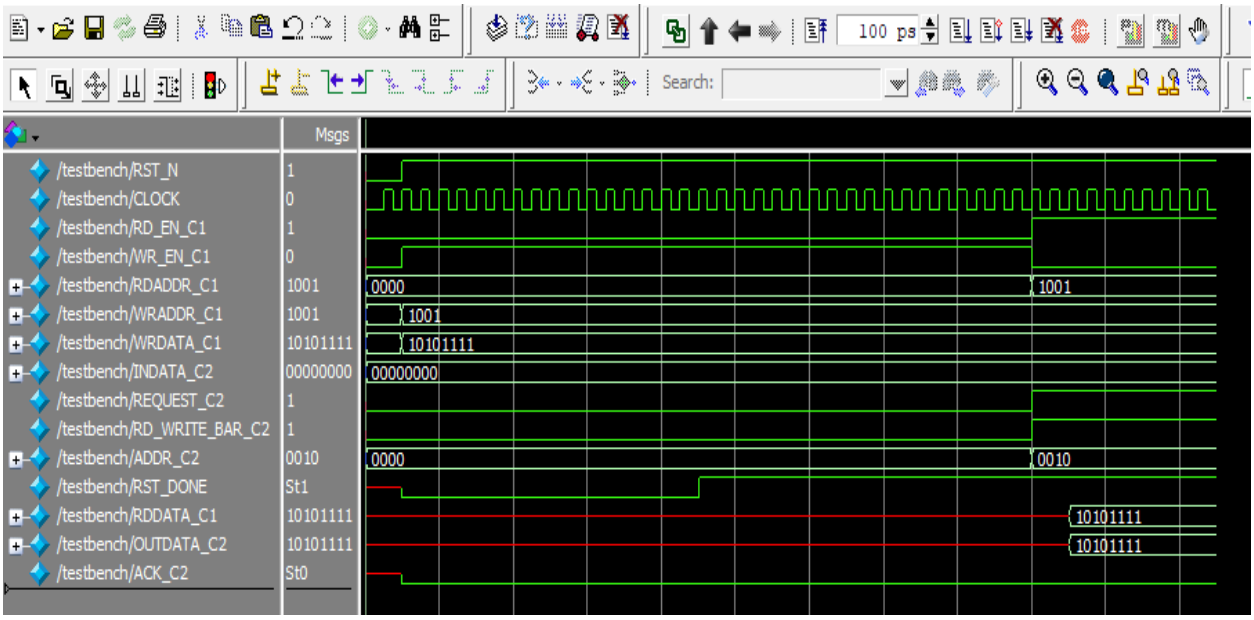
This is the case when both Client1 and Client2 want to access the same RAM location , this is called as Address Clashing Problem

```
RST_N = 1;
WR_EN_C1 = 1;
RD_EN_C1 = 0;
WRADDR_C1 = 4'b1001;
WRDATA_C1 = 8'b10101111;
#1700
RD_EN_C1 = 1;
WR_EN_C1 = 0;
RDADDR_C1 = 4'b1001;
REQUEST_C2 = 1;
RD_WRITE_BAR_C2 = 0;
ADDR_C2 = 4'b1001;
INDATA_C2 = 8'b10111011;
```



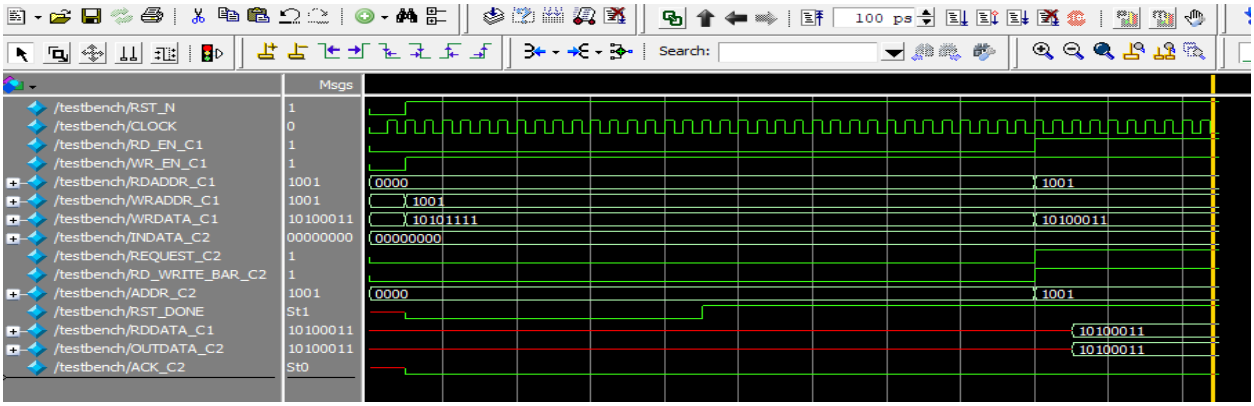
Case 11 = Client1 wants to read and Client2 wants to read in different RAM location at same time

```
RST_N = 1;
WR_EN_C1 = 1;
RD_EN_C1 = 0;
WRADDR_C1 = 4'b1001;
WRDATA_C1 = 8'b10101111;
#1700
RD_EN_C1 = 1;
WR_EN_C1 = 0;
RDADDR_C1 = 4'b1001;
REQUEST_C2 = 1;
RD_WRITE_BAR_C2 = 1;
ADDR_C2 = 4'b0010;
```



Case 12 = 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

```
RST_N = 1;
WR_EN_C1 = 1;
RD_EN_C1 = 0;
WRADDR_C1 = 4'b1001;
WRDATA_C1 = 8'b10101111;
#1700
RD_EN_C1 = 1;
RDADDR_C1 = 4'b1001;
WRADDR_C1 = 4'b1001;
WRDATA_C1 = 8'b10100011;
REQUEST_C2 = 1;
RD_WRITE_BAR_C2 = 1;
ADDR_C2 = 4'b1001;
```



Conclusion

In this project, we successfully designed and simulated architectural design of a RAM Arbiter. The single system interface with memory was essential in utilizing the memory and carrying out accesses to system then we use that singly accessed RAM module and designed a logic called Arbiter which basically serves a interface between multiple devices on one side and and only single resource at other side, and since we directly can't connect several systems with a singly accessable memory module, that's why Arbiter is very important and it is basic building block in nowadays memory controller.