iJRASET
International Journal For Research in Applied Science and Engineering Technology

INTERNATIONAL JOURNAL FOR RESEARCH

IN APPLIED SCIENCE & ENGINEERING TECHNOLOGY

Volume: 9 **Issue:** X **Month of publication:** October 2021

DOI: https://doi.org/10.22214/ijraset.2021.38451

www.ijraset.com

Call: 08813907089 | E-mail ID: ijraset@gmail.com

International Journal for Research in Applied Science & Engineering Technology (IJRASET)
ISSN: 2321-9653; IC Value: 45.98; SJ Impact Factor: 7.429
Volume 9 Issue X Oct 2021- Available at www.ijraset.com

Designing and verification of first in first out using Verilog

Kalyani Tekade
Ramaiah Institute of Technology

***Abstract:** This paper deals with the designing and verification of first in first out (fifo) using verilog. A FIFO is a memory queue which controls the data flow between two modules. It has the capacity to trigger different flags according to the status of the FIFO such has empty fifo status, half read fifo status, half written fifo status, full fifo status. Both the reading and writing operation can be performed simultaneously as it has dual port memory. After the completion of designing and simulating it on xilinx vivado this report also covers the verification carried out in modelsim. A detailed discussion on the architecture of each module that is writing module, reading module and memory array along with the various test benches and waveforms of simulation and verification is included in the paper.*

List of Figures and Tables

I. INTRODUCTION TO FIRST IN FIRST OUT (FIFO)

A. FIFO stands for First in First Out which describes to us how data is managed with respect to priority that is the data that arrives first will also be the data to leave first.

B. Thus, we would require a read/write memory array to keep the track of order in which the data is entered into FIFO or is read out of FIFO

C. In this project we will be dealing with synchronous FIFO.

D. However, we might be required to exchange data between systems operating at different clock frequencies known as clock domain crossing.

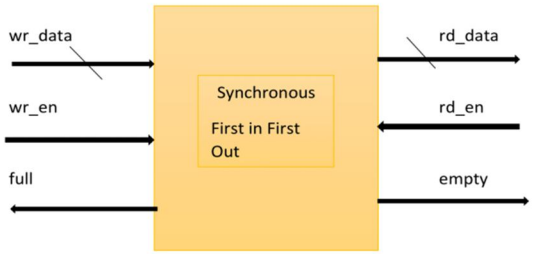
E. In such cases usually internally synchronized circuits (the systems that are synchronized with one another) is preferred to attain synchronization.

F. Flip Flops would serve as a synchronizer however in flip flops one has to make sure that the setup time (it is the minimum time for which the inputs have to be given before the clock pulse) and hold time (it is the minimum time for which the inputs have to remain constant after giving the clock input) is being satisfied.

G. Maintaining the setup and hold time Is very crucial as violating it would make the flip flop enter into metastable state which is also the unpredictable state of flip flop.

H. First in First out module can be used to effectively synchronise between different clock domain solving the synchronization problem.

Below is the block diagram of basic Synchronous FIFO

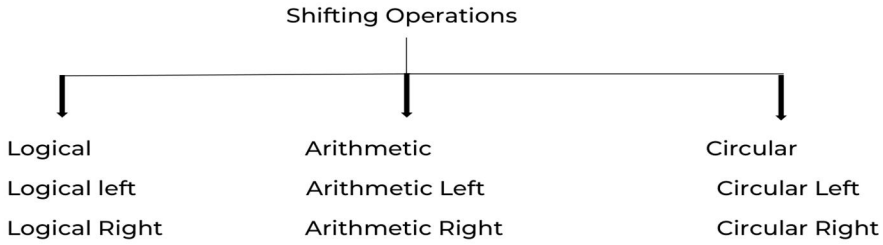


Basic FIFO Block Diagram

II. DIFFERENT TYPES OF SHIFT OPERATIONS

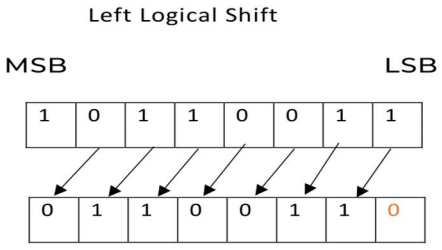
Shifting can be done in various ways

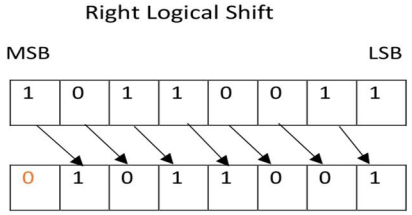
1) Logical operation
2) Arithmetic operation
3) Circular operation



A. Logical Left And Right Shift

1) A Left Logical Shift of one position moves each bit to the left by one. The vacant least significant bit (LSB) is filled with zero and the most significant bit (MSB) is discarded.
2) A Right Logical Shift of one position moves each bit to the right by one. The least significant bit is discarded and the vacant MSB is filled with zero.



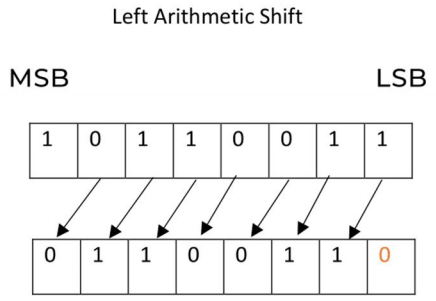


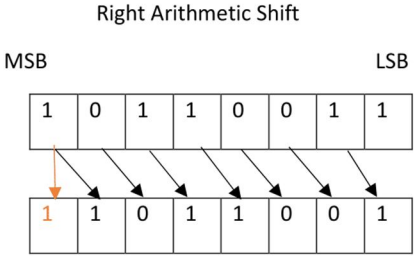
ISSN: 2321-9653; IC Value: 45.98; SJ Impact Factor: 7.429
Volume 9 Issue X Oct 2021- Available at www.ijraset.com

B. Arithmetic Left And Right Shift

1) A Left Arithmetic Shift of one position moves each bit to the left by one. The vacant least significant bit (LSB) is filled with zero and the most significant bit (MSB) is discarded. It is identical to Left Logical Shift.

2) A Right Arithmetic Shift of one position moves each bit to the right by one. The least significant bit is discarded and the vacant MSB is filled with the value of the previous (now shifted one position to the right) MSB.

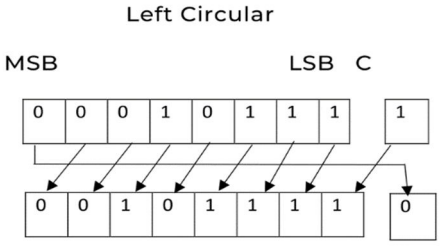


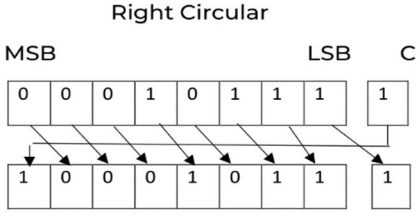


C. Circular Left And Right Shift

1) Left circular shift – moving the final bit to the first position while shifting all other bits to the next position.

2) Right circular shift -moving the first bit to the last position while shifting all other bits to the previous position.

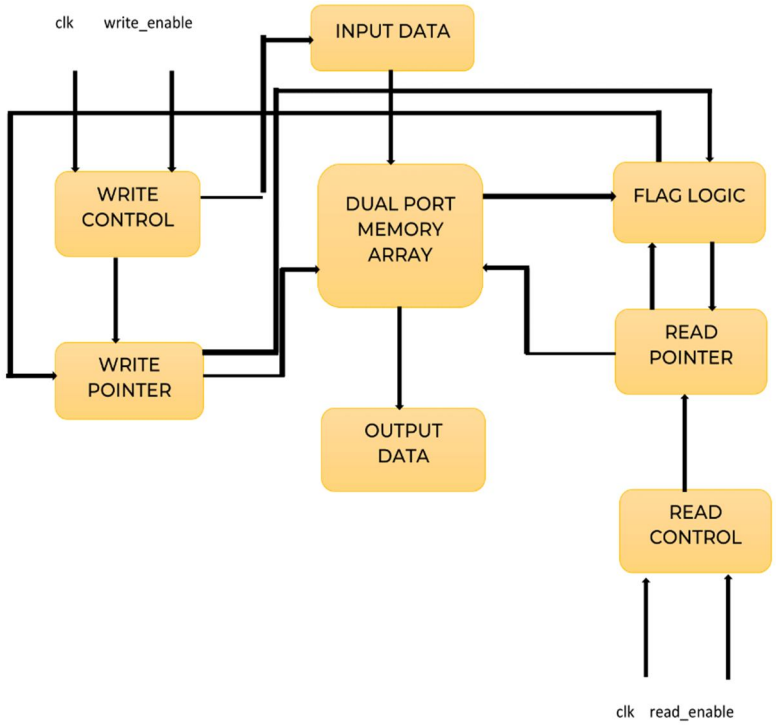




ISSN: 2321-9653; IC Value: 45.98; SJ Impact Factor: 7.429
Volume 9 Issue X Oct 2021- Available at www.ijraset.com

FIFO ARCHITECTURE

FLOW CHART OF FIFO



1) The synchronous FIFO is a first in first out memory queue with control logic that manages the read and write pointer and generates status flag.
2) The basic building block of a synchronous FIFO are memory array and flag logic.
3) The memory array has been implemented as dual port memory. Dual port implementation ensures simultaneous read and write access of the memory.
4) There are no timing or phase restrictions between accesses of the two port.
5) Data is steered into and out of the memory array by two pointers, a read address pointer and write address pointer. After each operation, the respective pointer is incremented to allow access to the next address sequentially in the array.
6) The flag logic compares the value in each of the two address pointer. If the difference between the two pointers is zero, the FIFO is empty and empty flag is asserted.
7) If the difference between the two values is equal to the depth of the port, the FIFO is full and full flag is asserted.

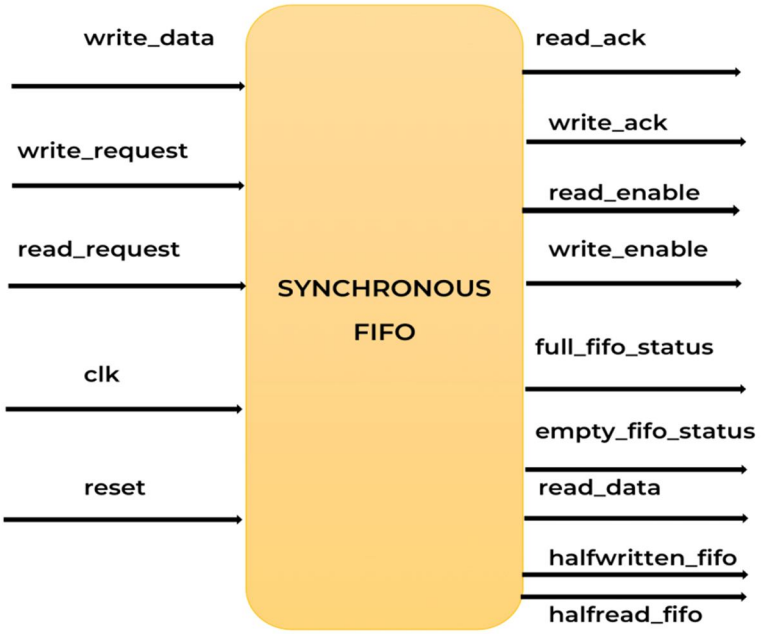
III. SYNCHRONOUS FIFO INPUT/OUTPUT SIGNALS

A. Input Signals To The Synchronous Fifo Design Include

1) Clk: All the read and write operation related signals are sampled at the rising edge of this clock input.
2) Read_request: This signal is sampled at the rising edge of clk. It decides the data read operation. When it is asserted, data read will be initiated. Data read from the memory array will be sent out through read_data output signal.
3) Write_request: This signal is sampled at the rising edge of clk. Data write happens when this signal is asserted.
4) Write_data: This is a 8 bit wide data signal written into the memory array, which is also a part of synchronous FIFO design.
5) Reset: Resetting the part sets the read and write address pointer to zero, clears the output data register and sets the status flags to represent empty device.

B. Output Signals From The Synchronous Fifo Design Include

1) *Read_data:* This is a 8 bit wide data signal read from the memory array, only when the read_enable is asserted.
2) *Read_enable:* This signal is enabled when a read is requested through a read_request signal and the status of the FIFO not being empty.
3) *Write_enable:* This signal is enabled when a write is requested through a write_request signal and the status of the FIFO not being full.
4) *Read_ack:* This is an acknowledgment given by the module notifying a successful read from the memory array by asserting the read_ack signal.
5) *Write_ack:* This is an acknowledgment given by the module notifying a successful write into the memory array by asserting the write_ack signal.
6) *Full_fifo_status:* This is a status signal that tries to notify the full status of the memory array. It is checked along with the write_request signal before asserting the write_enable signal.
7) *Empty_fifo_status:* This is a status signal that tries to notify the empty status of the memory array. It is checked along with the read_request signal before asserting the read_enable signal.
8) *Halfwritten_fifo_status:* This is a status signal that tries to notify if the half of the memory array is written.
9) *Halfread_fifo_status:* This status signal is asserted when half the memory array is read.

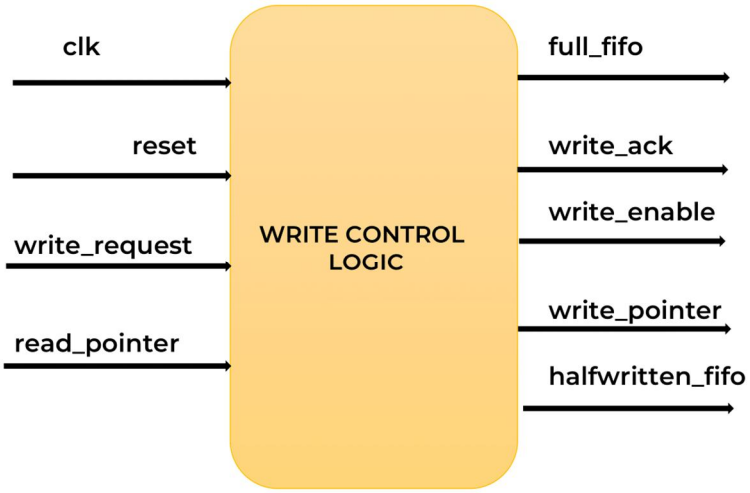


Block view of Synchronous FIFO

C. Write Control Logic

1) Write control logic is responsible for controlling the write of data into the memory array. This logic depends on the rising edge of the clk.
2) It increments a write pointer to point to the next empty location after every successful write.
3) The MSB of the write pointer will be used to indicate the memory array's full status.
4) The remaining 6 will be considered for the counter that increments after every successful write to point to the next empty location.
5) There is also a logic to notify the full status of the memory array.
6) Though a write is request comes, if the memory array is full this control logic waits till the full_fifo_status signal gets de-asserted for doing a successful write.

ISSN: 2321-9653; IC Value: 45.98; SJ Impact Factor: 7.429
Volume 9 Issue X Oct 2021- Available at www.ijraset.com



D. Read Control Logic

1) Read control logic is responsible for controlling the read of data from the memory array. All the signals in this logic act at the rising edge of clk. A 7 bit read pointer is incremented after a successful read to point to the next read location.

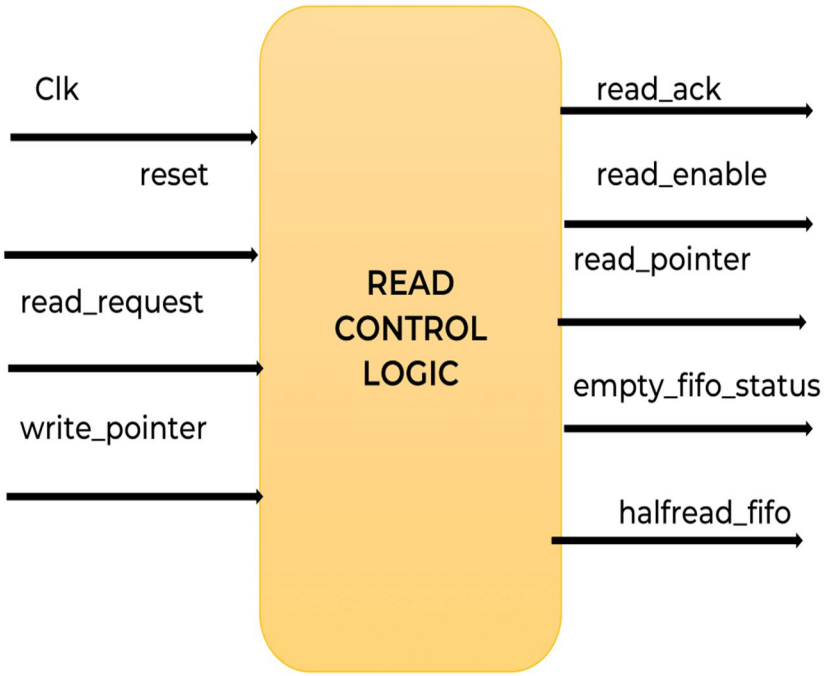
2) The MSB of the 7 bit will be used to indicate the FIFO (memory array) empty status.

3) The remaining 6 bit will be considered for the counter that increments after every successful read to point to the next available memory address.

4) This logic also asserts or de-asserts a flag meant for telling the FIFO's almost empty status.

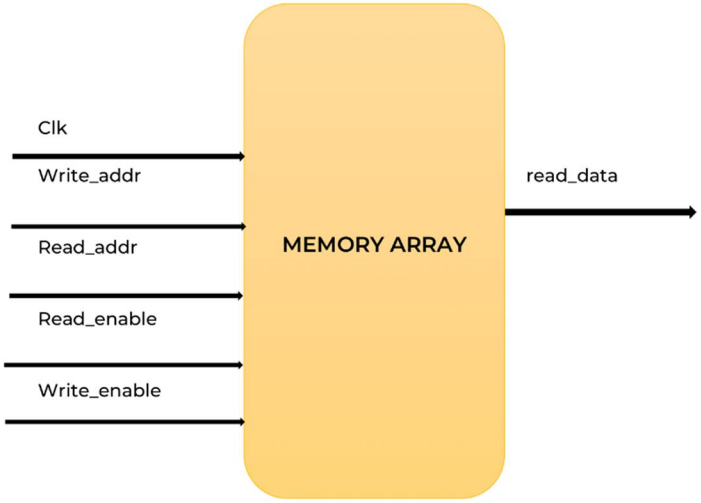
5) If the memory array is empty, read operation is not allowed even when there is a read request through a read_request signal.

6) Acknowledgment is given after a successful read through a read_ack signal.



E. Memory Array

1) Memory array used in this module is a dual port write/read RAM, which operates using clk.
2) The depth of the memory array is 8, which has been provided to a constant name FIFO_DEPTH in this code. At every rising edge of the clk data is read through read_data line from the memory location pointed by the read_addr(read address, only if read_enable signal is high)
3) At every rising edge of clk, data is written into the memory array at the location pointed by write_addr
4) Both these read and write operation happens at their corresponding rates simultaneously.



F. Status Flag

1) Status flag such as empty flag, half read flag, full flag and half written full flag are used to determine the FIFO status.
2) These flags are generated by comparing the values in the read and write address pointer.
3) External control logic should use these flag to determine if read or write operations can be performed on the FIFO.

G. Flag Update Cycle

1) As the empty and full flags are synchronous they require a rising edge of the clock to update them to their most current value.
2) Under boundary condition(full or empty) there is a dead cycle known as “flag update cycle”.
3) For example, we assume we have an empty FIFO, a write is performed to the port on the rising edge of clk.
4) The memory is no longer empty, because the empty_flag is synchronized to the clock, the flag will not be de-asserted until it receives a clock rising edge.
5) Read operation to the device are prevented until the empty flag is de-asserted.
6) The first clock rising edge updates the flags and second clock rising edge reads out the first word.
7) The initial clock cycle is referred to as dead cycle or flag update cycle.

H. Generation Of Fifo Full Status And Fifo Empty Status

1) Write control logic includes a code that asserts or de-asserts full_fifo_status and halfwritten_fifo_status.
2) Read control logic includes a code that asserts or de-asserts the empty-status and halfread_fifo_status signal.
3) The LSB of a 6 bits are extracted from two pointer, read pointer and write pointer and are stored in separate register read_addr and write_addr.
4) If the values of read_addr and write_addr match only the MSB of the read pointer and write pointer decide if the memory array is full or empty.
5) If the MSB’s are different it means the memory array is full.
6) If the MSB’s are same it means the memory array is empty.
7) Accordingly, the corresponding flags are set.

ISSN: 2321-9653; IC Value: 45.98; SJ Impact Factor: 7.429
Volume 9 Issue X Oct 2021- Available at www.ijraset.com

I. Design Code

```
module fifo( reset, clk, write_request, read_request, write_enable, read_enable, write_ack, write_count, read_count, read_ack, full_fifo_status, empty_fifo_status, halfwritten_fifo_status, halfread_fifo_status, read_data, write_data );

    parameter mem_depth=8 ;
    parameter mem_addr_width=8;
    parameter mem_data_width=8;
    parameter mem_width=8;
    parameter fifo_halffull=4;
    parameter MAX_COUNT=8'b00001000;
    input reset;
    input clk;
    input write_request;
    input read_request;
    input  [mem_data_width-1:0] write_data;
    output write_enable;
    output read_enable;
    output full_fifo_status;
    output empty_fifo_status;

    output reg read_ack;
    output reg write_ack;
    output reg halfread_fifo_status;
    output reg halfwritten_fifo_status;
    output reg [mem_data_width-1:0] read_data;

    reg[mem_addr_width: 0] write_pointer =0;
    reg[mem_addr_width: 0] read_pointer = 0;
    output reg [(mem_depth -1) : 0]write_count = 8'b00000000 ;
    output reg [(mem_depth -1) : 0]read_count = 8'b00001000;

    reg[mem_data_width -1:0] memory [mem_depth -1:0];

    wire[mem_addr_width-1:0] write_addr;
    wire[mem_addr_width-1:0] read_addr;

    assign read_addr= read_pointer[mem_addr_width-1:0];
    assign write_addr= write_pointer[mem_addr_width-1:0];

    assign full_fifo_status=((write_addr == mem_depth));
    assign empty_fifo_status=((write_addr == 0)&&(read_addr==0));
    assign write_enable= (write_request && (full_fifo_status==0)) ?1'b1:1'b0;
    assign read_enable= (read_request && (empty_fifo_status==0)) ?1'b1:1'b0;
    always@(posedge clk or posedge reset)
    begin
    if(reset)
    begin
    write_pointer <= 0;
    write_ack <= 0;
```

```
end
else
begin
halfwritten_fifo_status=(((mem_depth)-write_addr)<=(fifo_halffull))?1'b1:1'b0;
if(write_enable)
begin
write_ack<=1;
write_pointer<=write_pointer+1'b1;
end
end
end

always@(posedge clk or posedge reset)
begin
if (write_request && (write_pointer ==4'b1000))
write_pointer = 1'b0;
end

always@(posedge clk or posedge reset)
begin
if(reset)
begin
read_pointer <= 0;
read_ack <= 0;
end
else
begin
halfread_fifo_status=((fifo_halffull)<=(read_addr))?1'b1:1'b0; if(read_enable)
begin
memory[read_pointer]<=0;
read_ack<=1;
read_pointer<=read_pointer+1'b1;
if(read_pointer == 4'b1000)
begin
read_pointer<=1'b0;
end
end
end
end

always@(posedge clk or posedge reset)
begin
if (read_request && read_pointer ==4'b1000)
read_pointer=1'b1;
end

always @(posedge clk)
begin
if(reset==1) begin
write_count <= 8'b00000000;
```

ISSN: 2321-9653; IC Value: 45.98; SJ Impact Factor: 7.429
Volume 9 Issue X Oct 2021- Available at www.ijraset.com

```
read_count <= 8'b00000000;
end
else begin
case({read_request,write_request})
2'b00: begin
 read_count = read_count;
write_count = write_count;
end
2'b01:          begin
if(write_count != MAX_COUNT)        begin
write_count = write_count+1;
read_count = read_count;
end
else
begin
write_count = write_count;
end
end
2'b10: begin
if(read_count!=8'b00000000)
read_count = read_count-1;
write_count = write_count;
end
2'b11: begin
if(empty_fifo_status == 1)
write_count = write_count+1;
read_count = read_count;
end
endcase

end

end

always@(posedge clk)
begin
if(write_request)
memory[write_addr] <= write_data;
end

always@(posedge clk)
begin
if(read_request)
read_data<= memory[read_addr];
end

endmodule
```

ISSN: 2321-9653; IC Value: 45.98; SJ Impact Factor: 7.429
Volume 9 Issue X Oct 2021- Available at www.ijraset.com

J. Test Bench

```
module fifo_tb();
parameter mem_depth=8 ;
parameter mem_addr_width=8;
parameter mem_data_width=8;
parameter mem_width=8;
parameter fifo_halffull=4;
parameter MAX_COUNT=8'b00001000;

reg clk;
reg reset;
reg write_request;
reg read_request;
reg[mem_data_width - 1 : 0]write_data;

wire[mem_data_width - 1 : 0]read_data;
wire full_fifo_status;
wire empty_fifo_status;
wire halfwritten_fifo_status;
wire halfread_fifo_status;
wire [(mem_depth -1) : 0]write_count ;
wire [(mem_depth -1) : 0]read_count ;
wire[7:0] memory0, memory1, memory2, memory3, memory4, memory5, memory6, memory7;

assign memory0=f1.memory[0];
assign memory1=f1.memory[1];
assign memory2=f1.memory[2];
assign memory3=f1.memory[3];
assign memory4=f1.memory[4];
assign memory5=f1.memory[5];
assign memory6=f1.memory[6];
assign memory7=f1.memory[7];


fifo f1(reset, clk, write_request, read_request, write_enable, read_enable, write_ack,write_count, read_count,
read_ack, full_fifo_status, empty_fifo_status, halfwritten_fifo_status, halfread_fifo_status, read_data, write_data);
always
begin
clk=1'b0;
forever #10 clk=~clk;
end
//writing and reading operation
//Write 8 data and read 8 data:

initial
begin
#10 read_request=1'b0;write_request=1'b1;reset=1'b0;write_data= 8'b0001_0010;
#20 write_data= 8'b0010_0011;
#20 write_data= 8'b0011_0100;
#20 write_data= 8'b0100_0101;
```

```
    #20 write_data= 8'b0101_0110;
    #20 write_data= 8'b0110_0111;
    #20 write_data= 8'b0111_1000;
    #20 write_data= 8'b1000_1001;
    #20 write_request=1'b0;read_request=1'b0;
    #20;
    #20 read_request=1'b1;write_request=1'b0;
    #20 read_request=1'b1;write_request=1'b0;
    #20 read_request=1'b1;write_request=1'b0;
    #20 read_request=1'b1;write_request=1'b0;
    #20 read_request=1'b1;write_request=1'b0;
    #20 read_request=1'b1;write_request=1'b0;
    #20 read_request=1'b1;write_request=1'b0;
    #20 read_request=1'b1;write_request=1'b0;
    #20 $stop;
    End
```

```
//Write 2 data and read 2 data:
    initial
    begin
    #10 read_request=1'b0;write_request=1'b1;reset=1'b0;write_data= 8'b0001_0010;
    #20 write_data= 8'b0010_0011;
    #20 write_request=1'b0;read_request=1'b0;
    #20;
    #20 read_request=1'b1;write_request=1'b0;
    #20 read_request=1'b1;write_request=1'b0;
    #20 $stop;
    end
```

```
//Write 4 Data and read only 2 data:
    initial
    begin
    #10 read_request=1'b0;write_request=1'b1;reset=1'b0;write_data= 8'b0001_0010;
    #20 write_data= 8'b0010_0011;
    #20 write_data= 8'b0011_0100;
    #20 write_data= 8'b0100_0101;
    #20 write_request=1'b0;read_request=1'b0;
    #20;
    #20 read_request=1'b1;write_request=1'b0;
    #20 read_request=1'b1;write_request=1'b0;
    #20 $stop;
    end
```

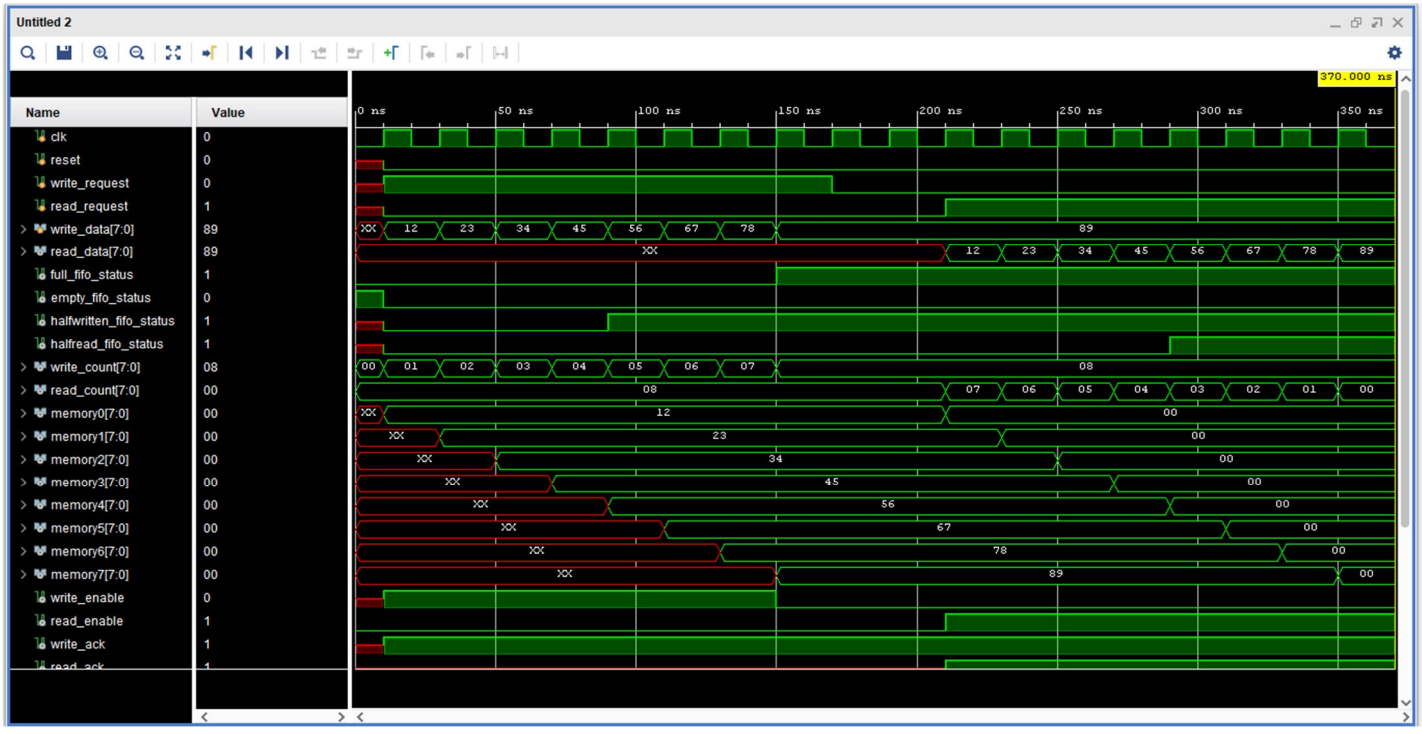
```
//Write 6 Data and check full Flag status:
    initial
    begin
    #10 read_request=1'b0;write_request=1'b1;reset=1'b0;write_data= 8'b0001_0010;
    #20 write_data= 8'b0010_0011;
    #20 write_data= 8'b0011_0100;
    #20 write_data= 8'b0100_0101;
    #20 write_data= 8'b0101_0110;
```

```
    #20 write_data= 8'b0110_0111;
    #20 write_request=1'b0;read_request=1'b0;
    #20;
    #20 $stop;
    end

//Read 4 Data and check half read flag:
    initial
    begin
    #10 read_request=1'b0;write_request=1'b1;reset=1'b0;write_data= 8'b0001_0010;
    #20 write_data= 8'b0010_0011;
    #20 write_data= 8'b0011_0100;
    #20 write_data= 8'b0100_0101;
    #20 write_data= 8'b0101_0110;
    #20 write_data= 8'b0110_0111;
    #20 write_data= 8'b0111_1000;
    #20 write_data= 8'b1000_1001;
    #20 write_request=1'b0;read_request=1'b0;
    #20;
    #20 read_request=1'b1;write_request=1'b0;
    #20 read_request=1'b1;write_request=1'b0;
    #20 read_request=1'b1;write_request=1'b0;
    #20 read_request=1'b1;write_request=1'b0;
    #20 read_request=1'b1;write_request=1'b0;
    #20 $stop;
    end

//Write 4 Data:
    initial
    begin
    #10 read_request=1'b0;write_request=1'b1;reset=1'b0;write_data= 8'b0001_0010;
    #20 write_data= 8'b0010_0011;
    #20 write_data= 8'b0011_0100;
    #20 write_data= 8'b0100_0101;
    #20 write_request=1'b0;read_request=1'b0;
    #20;
    #20 $stop;
    end

//Read all 8 Data and check empty fifo status:
    initial
    begin
    #10 read_request=1'b0;write_request=1'b1;reset=1'b0;write_data= 8'b0001_0010;
    #20 write_data= 8'b0010_0011;
    #20 write_data= 8'b0011_0100;
    #20 write_data= 8'b0100_0101;
    #20 write_data= 8'b0101_0110;
    #20 write_data= 8'b0110_0111;
    #20 write_data= 8'b0111_1000;
    #20 write_data= 8'b1000_1001;
    #20 write_request=1'b0;read_request=1'b0;
```

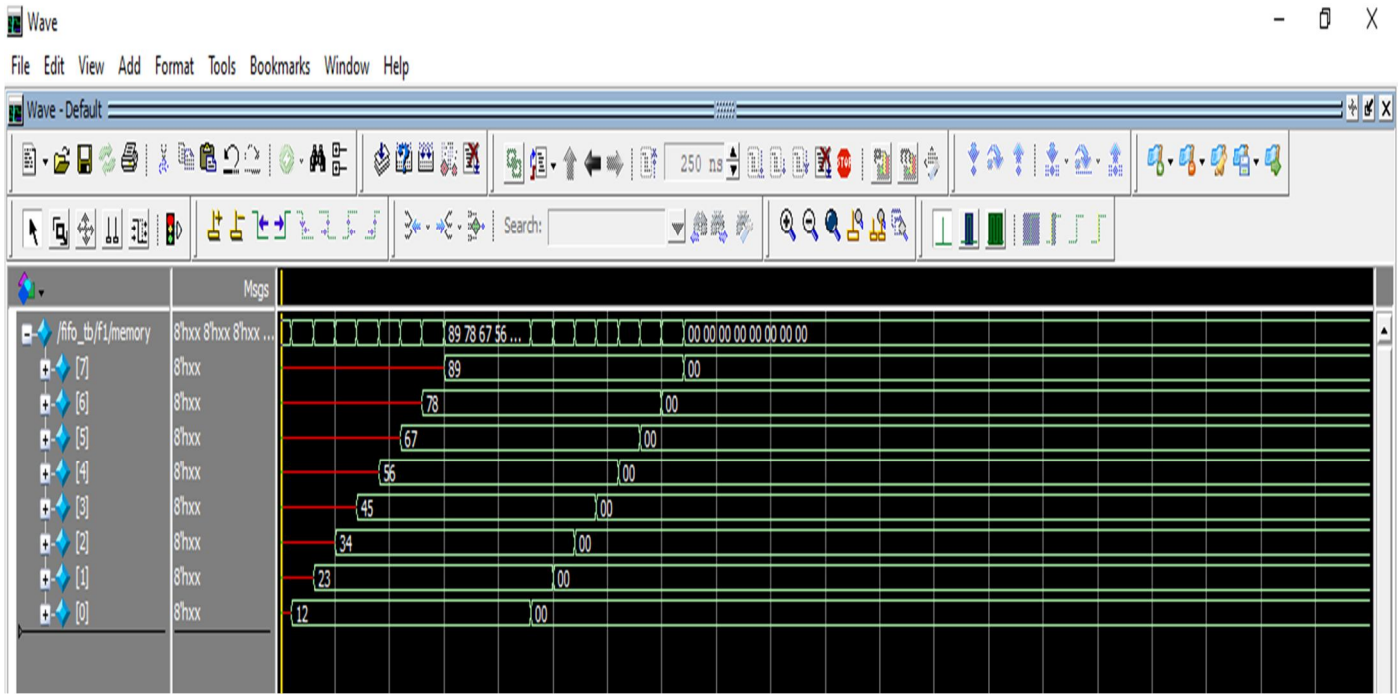
```
#20;
#20 read_request=1'b1;write_request=1'b0;
#20 read_request=1'b1;write_request=1'b0;
#20 read_request=1'b1;write_request=1'b0;
#20 read_request=1'b1;write_request=1'b0;
#20 read_request=1'b1;write_request=1'b0;
#20 read_request=1'b1;write_request=1'b0;
#20 read_request=1'b1;write_request=1'b0;
#20 read_request=1'b1;write_request=1'b0;
#20;
#20 write_request=1'b1;read_request=1'b0;
#20 $stop;
end
```

```
//Write 4 Data and check half write flag:
initial
begin
#10 read_request=1'b0;write_request=1'b1;reset=1'b0;write_data= 8'b0001_0010;
#20 write_data= 8'b0010_0011;
#20 write_data= 8'b0011_0100;
#20 write_data= 8'b0100_0101;
#20 write_data= 8'b0101_0110;
#20 write_request=1'b0;read_request=1'b0;
#20;
#20 $stop;
end
endmodule
```

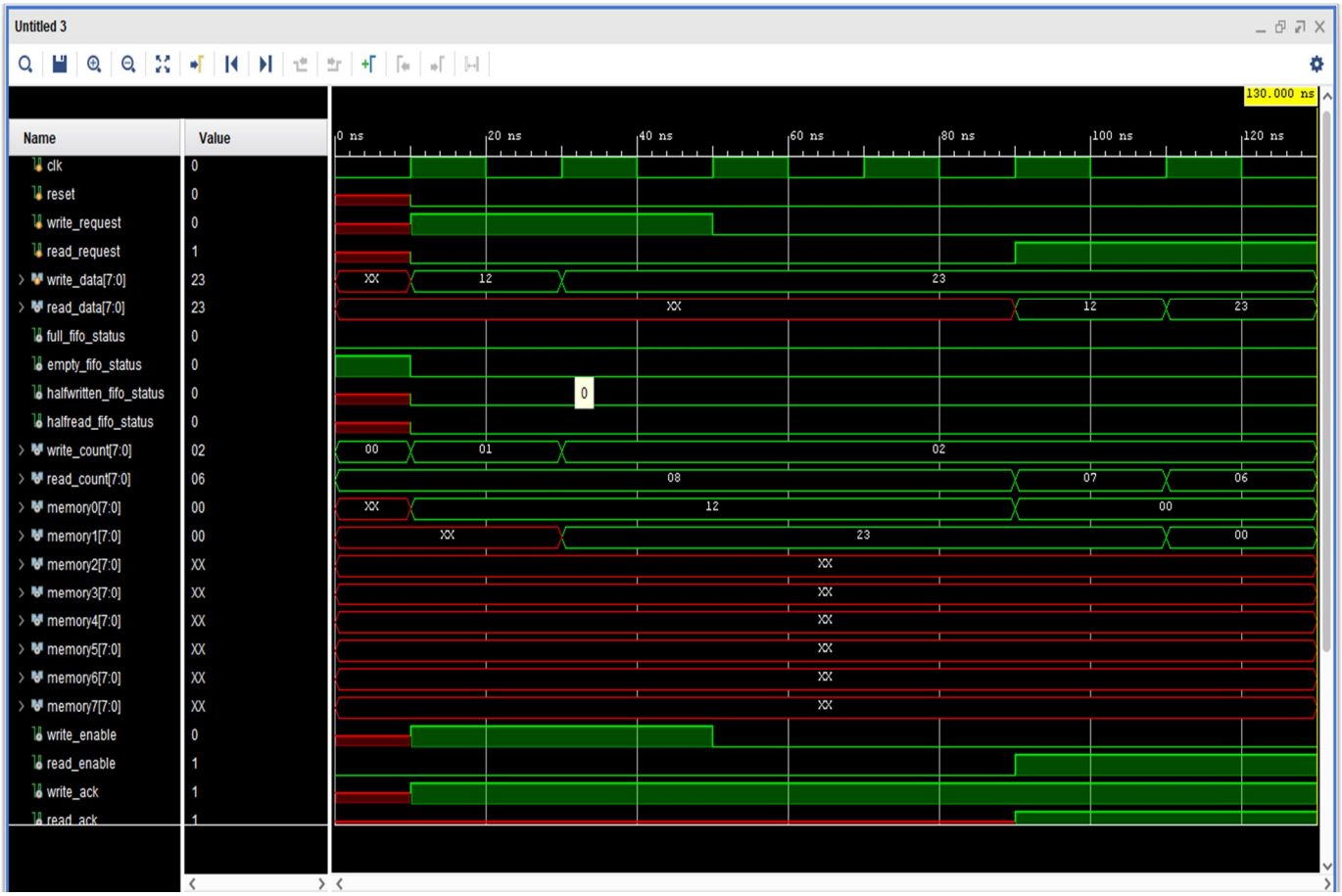
K. Simulation Using Vivado For Writing And Reading All 8 Data



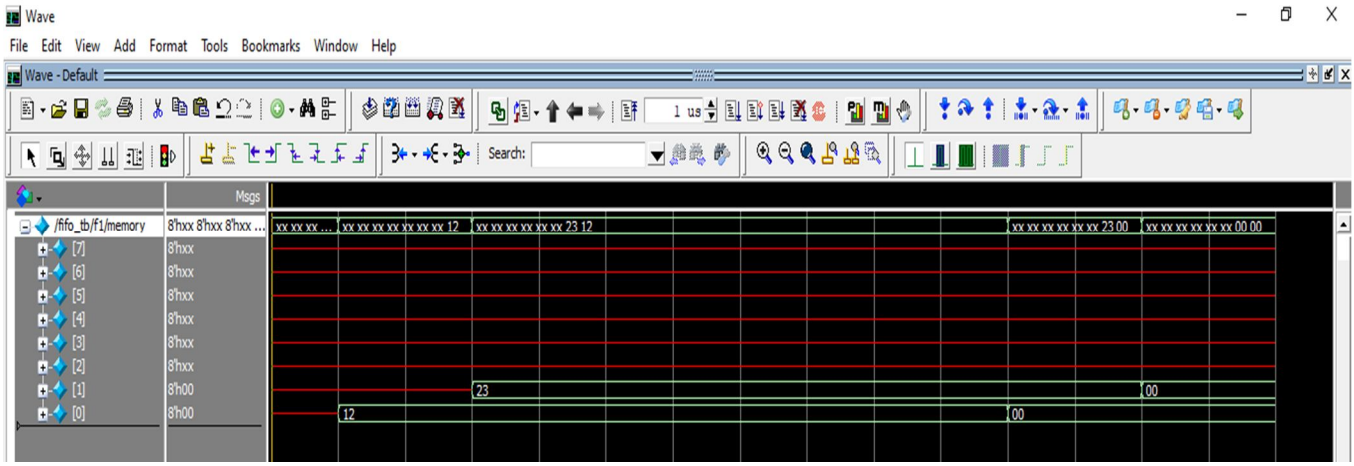
L. Verification Using Modelsim For Writing And Reading All 8 Data



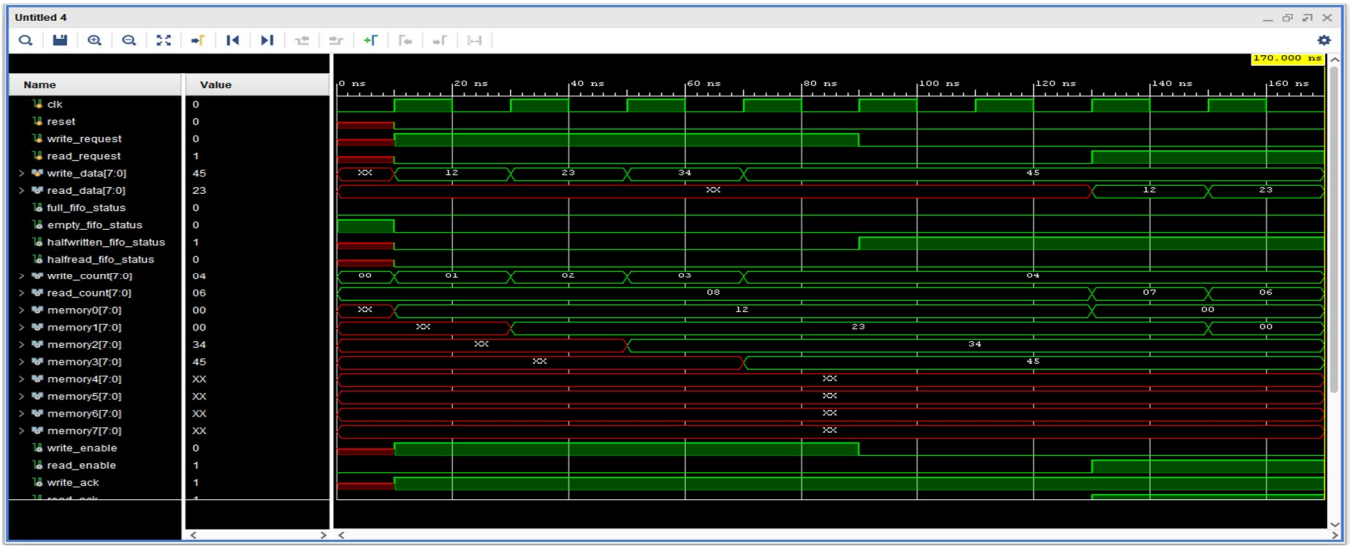
M. Simulation Using Vivado For Writing And Reading 2 Data



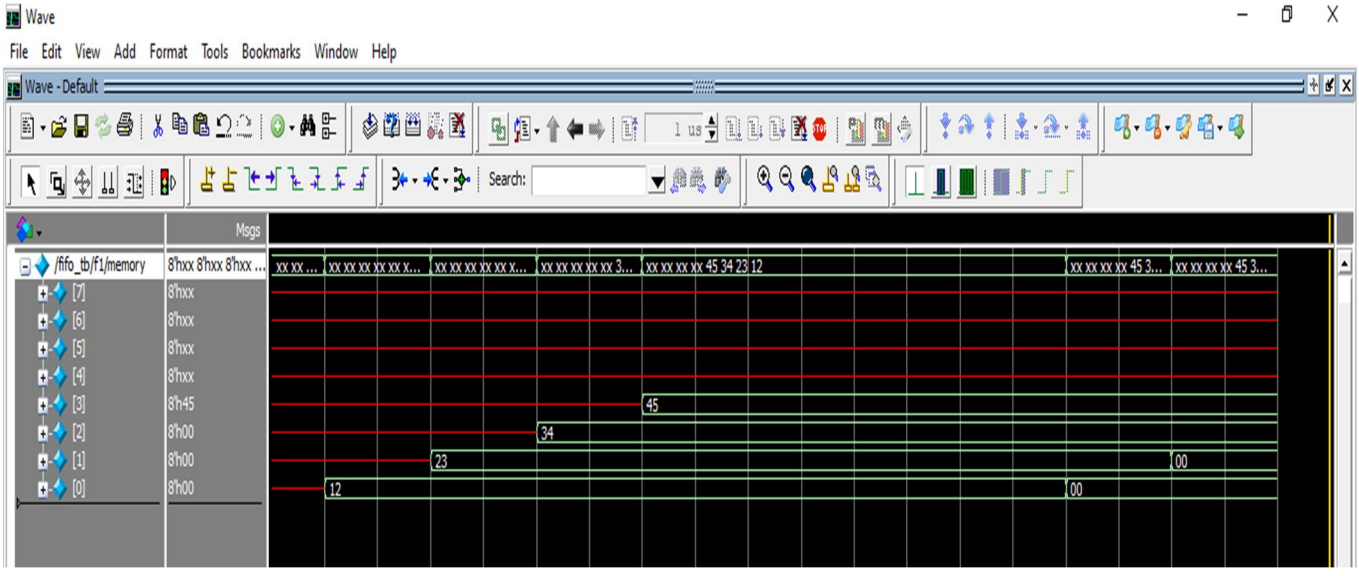
N. Verfication Using Modelsim For Writing And Reading 2 Data



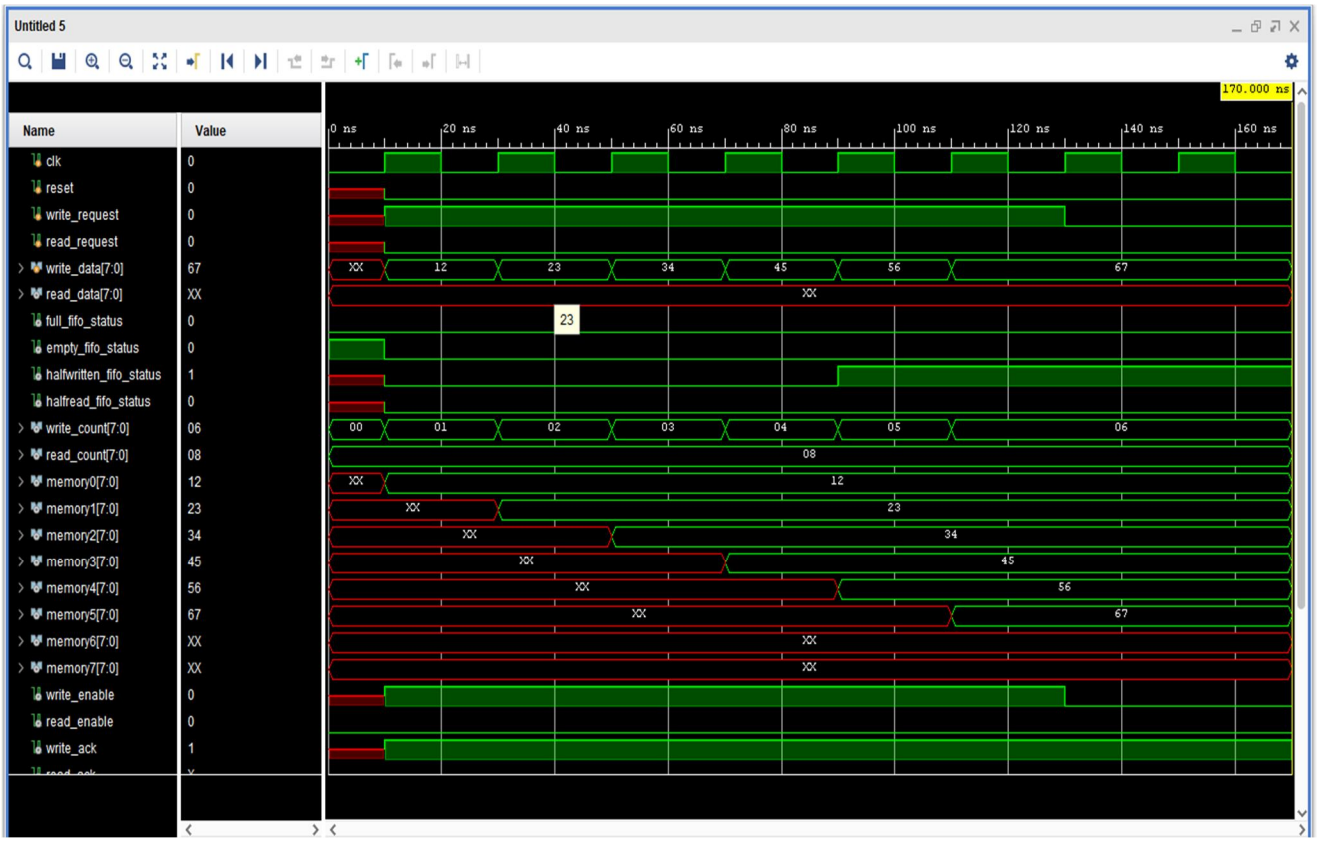
O. Simulation Using Vivado For Writing 4 Data And Reading 2 Data



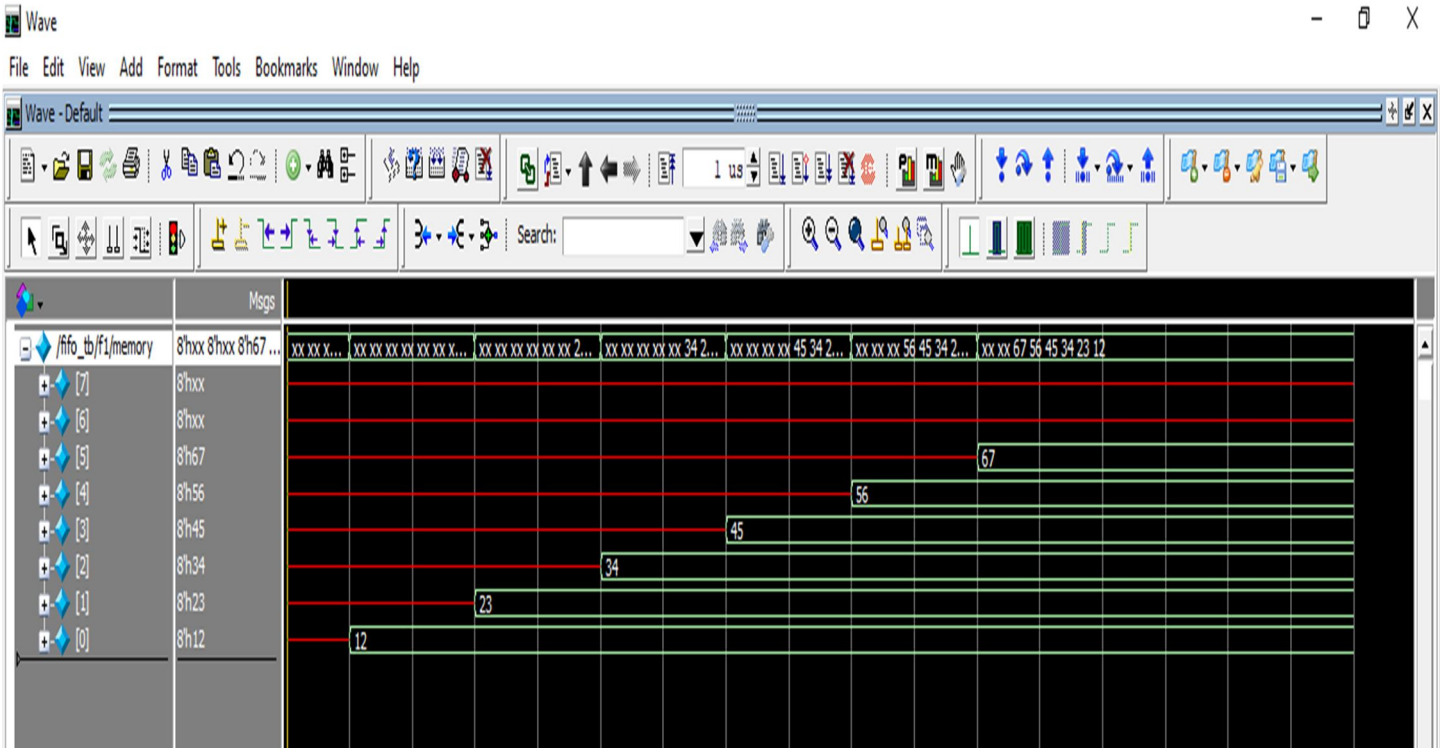
P. Verfication Using Modelsim For Writing 4 Data And Reading 2 Data



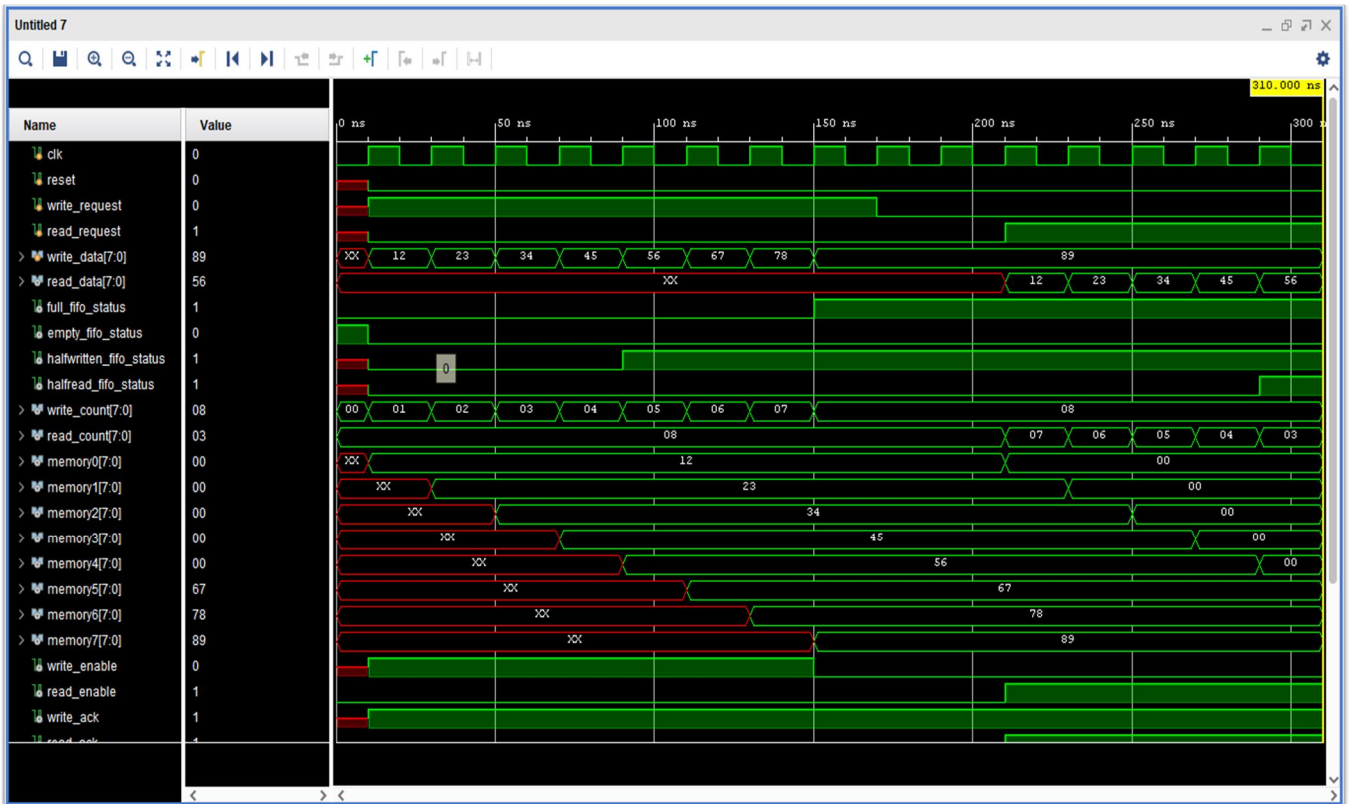
Q. Simulation Using Vivado For Writing 6 Data And Checking Full Flag Status



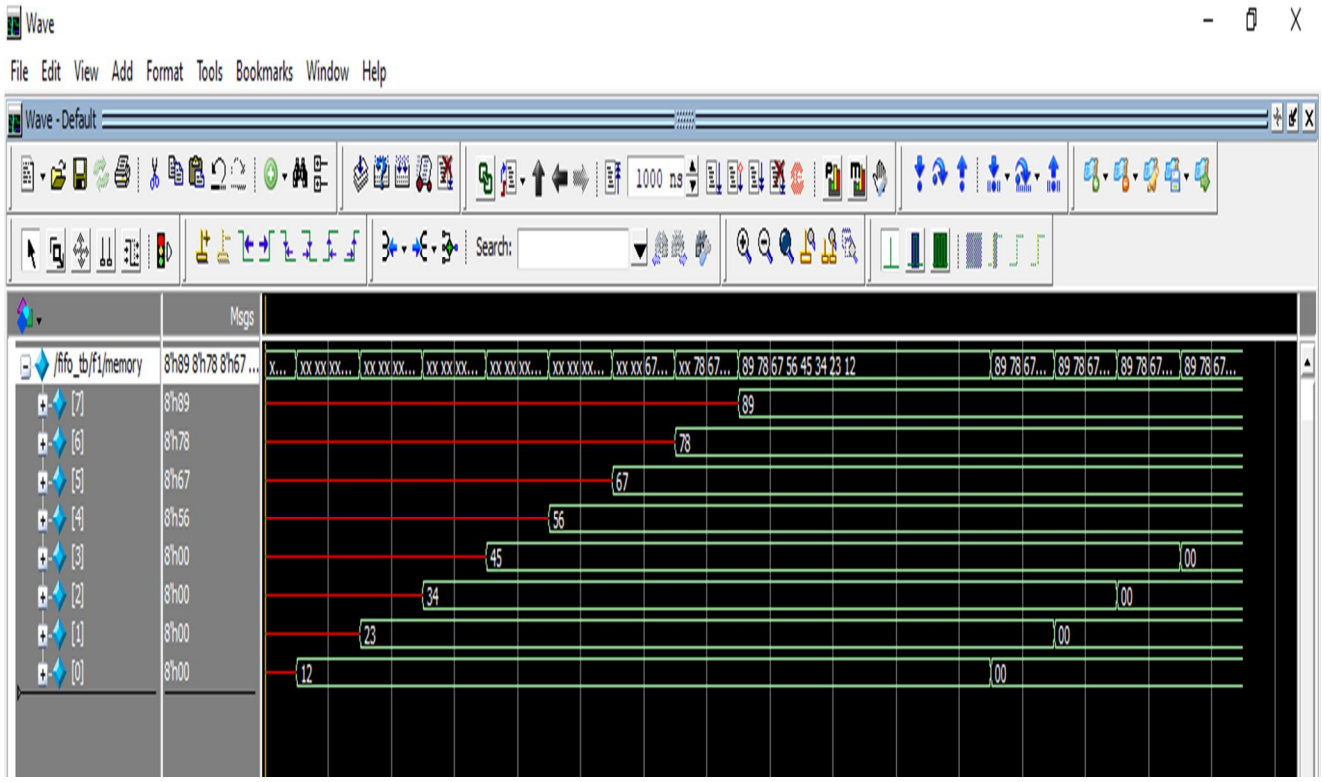
R. Verfication Using Modelsim For Writing 6 Data And Z Full Flag Status



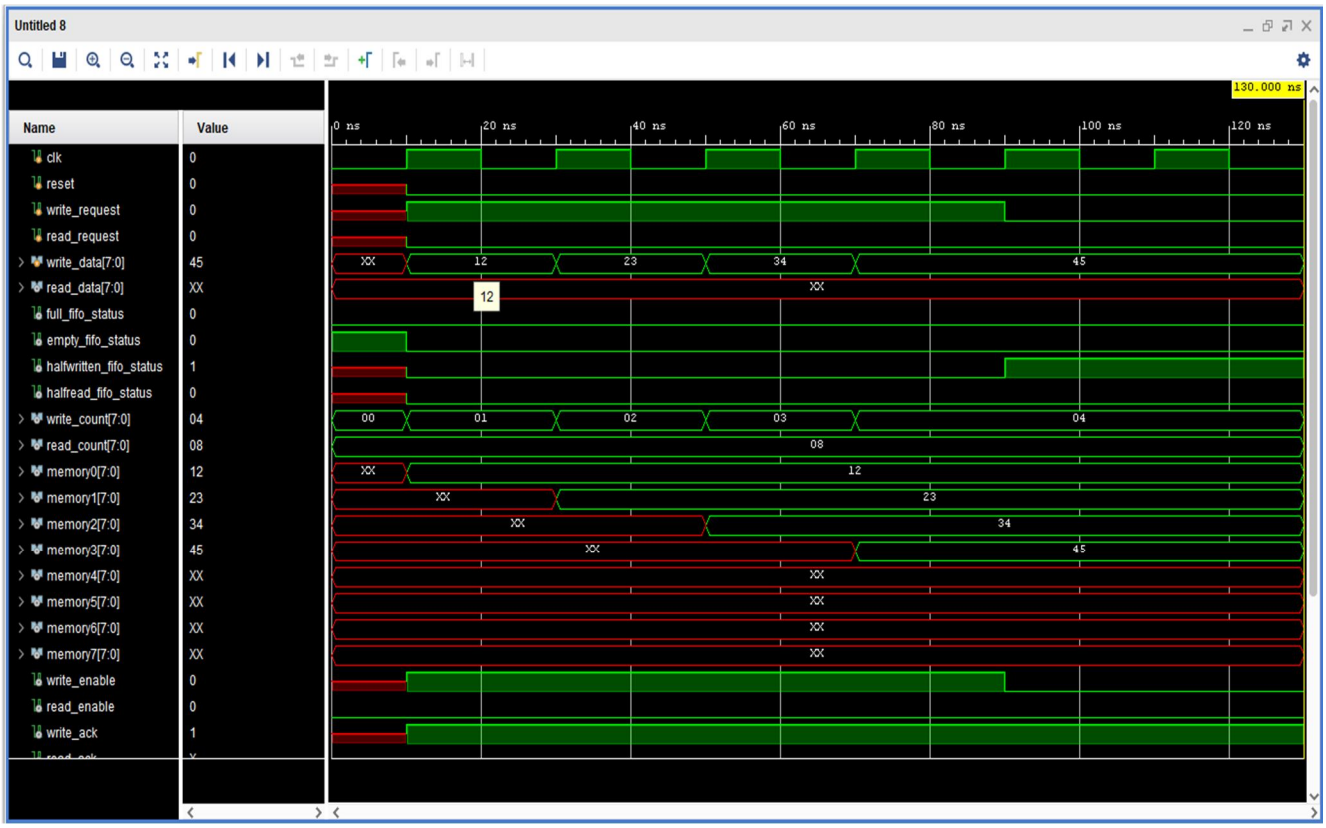
S. Simulation Using Vivado For Reading 4 Data And Checking Half Read Flag



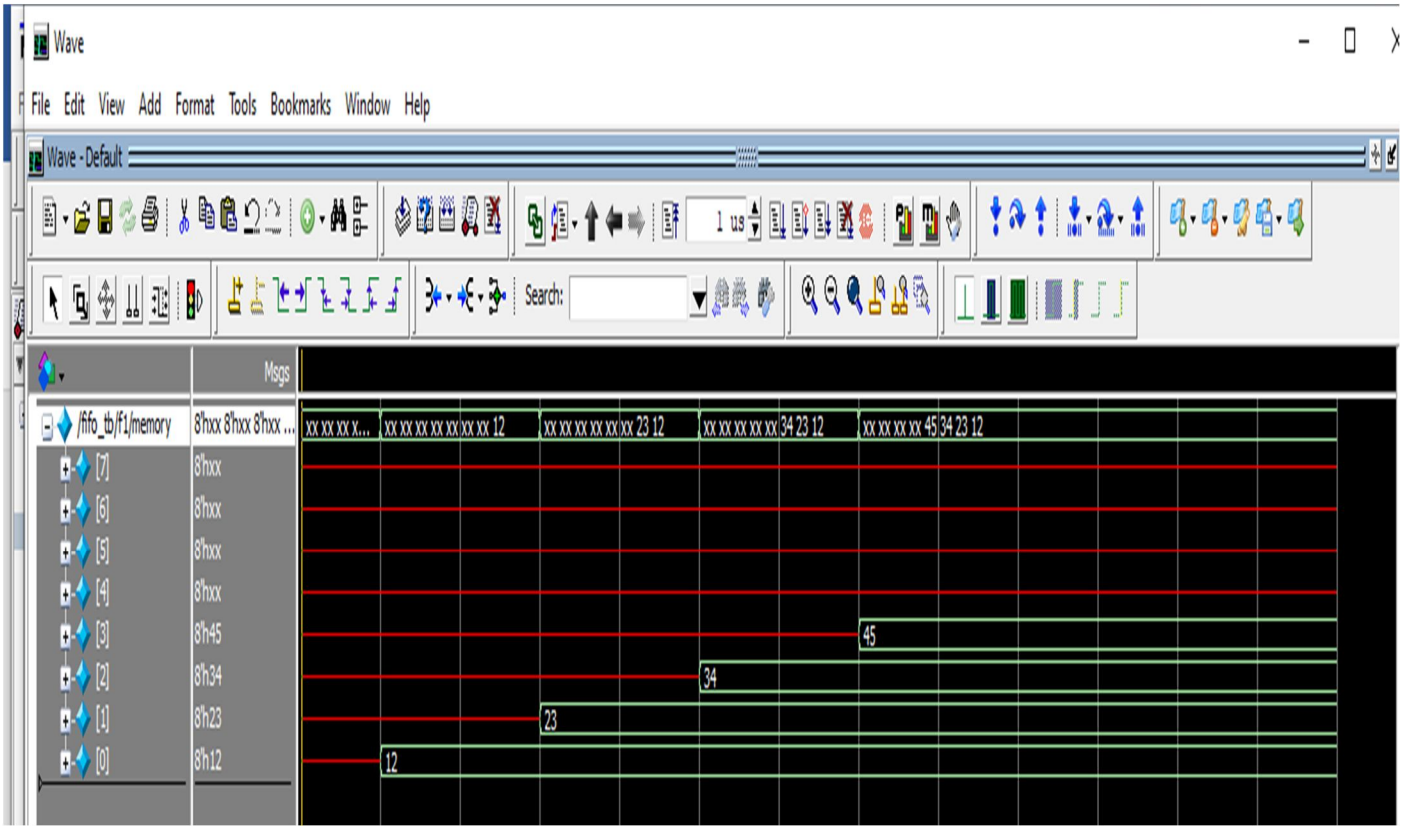
T. Verfication Using Modelsim For Reading 4 Data And Checking Half Read Flag



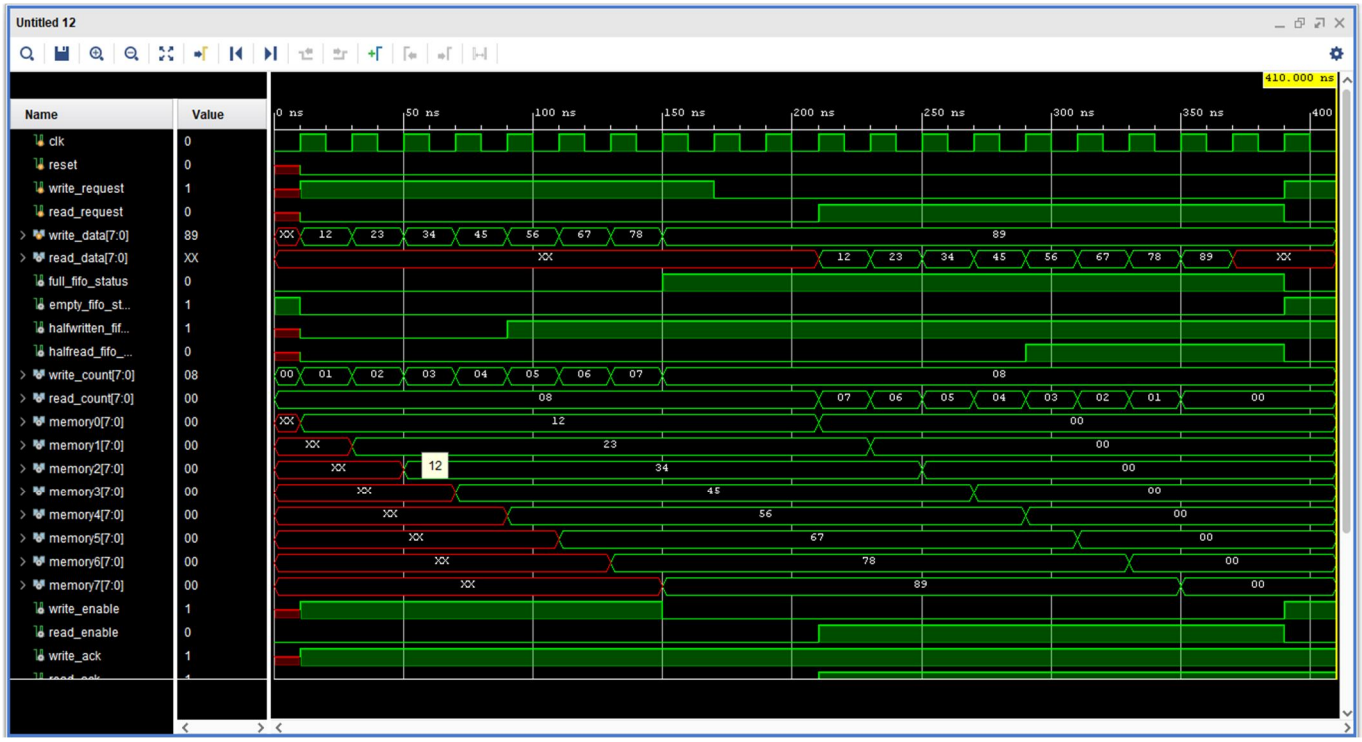
U. Simulation Using Vivado For Writing 4 Data



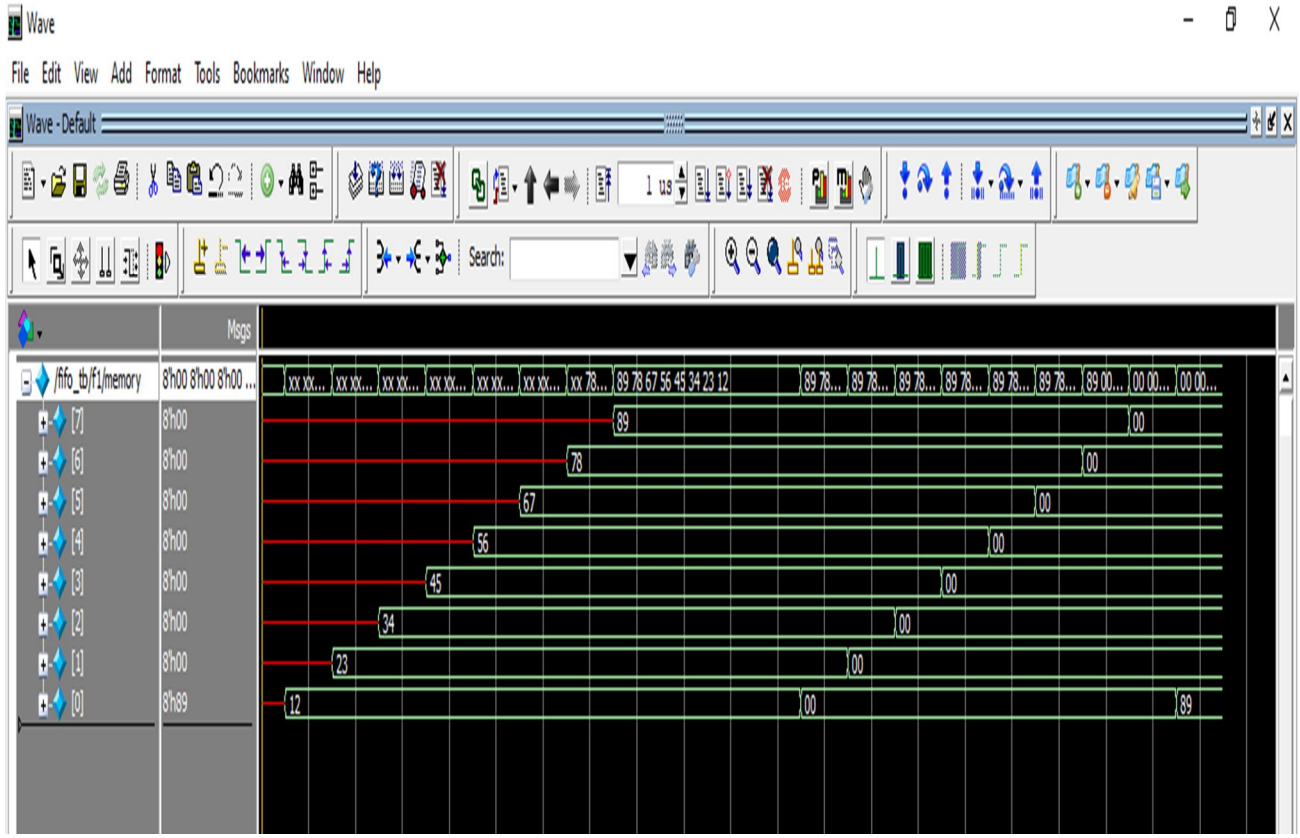
V. Verfication Using Modelsim For Writing 4 Data



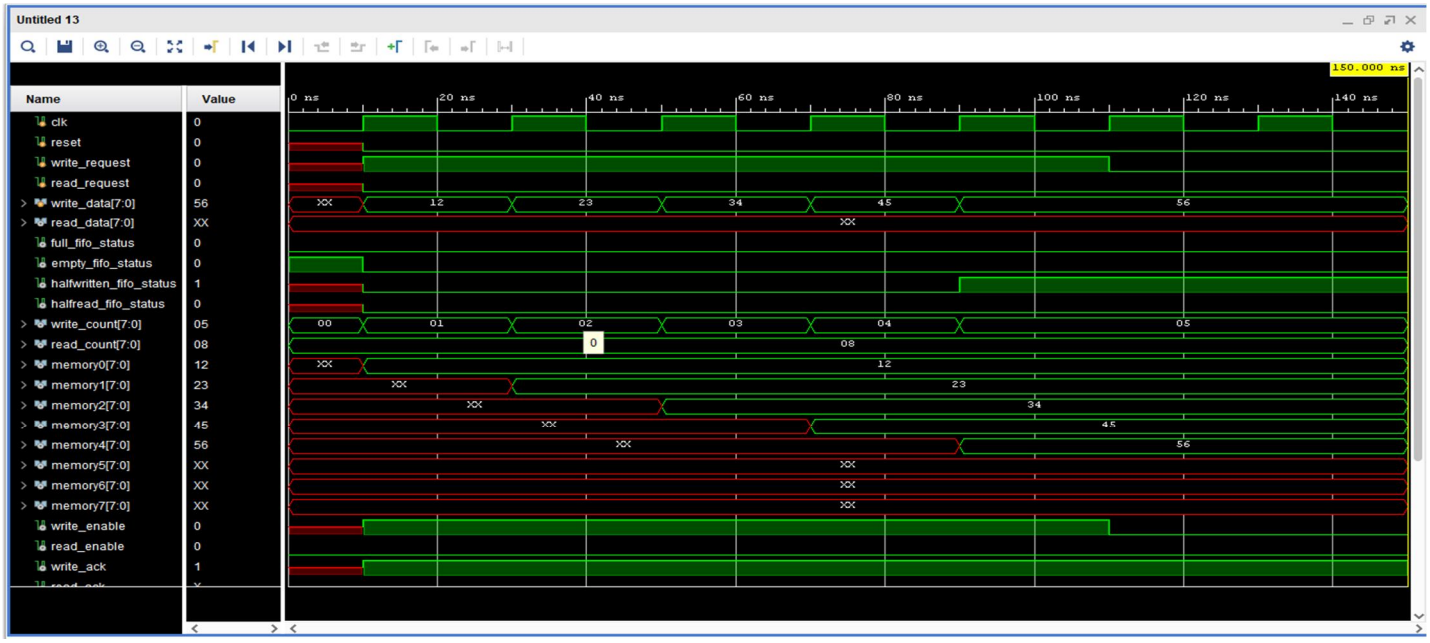
W. Simulation Using VIVADO For Reading All 8 Data And Checking Empty Flag Status



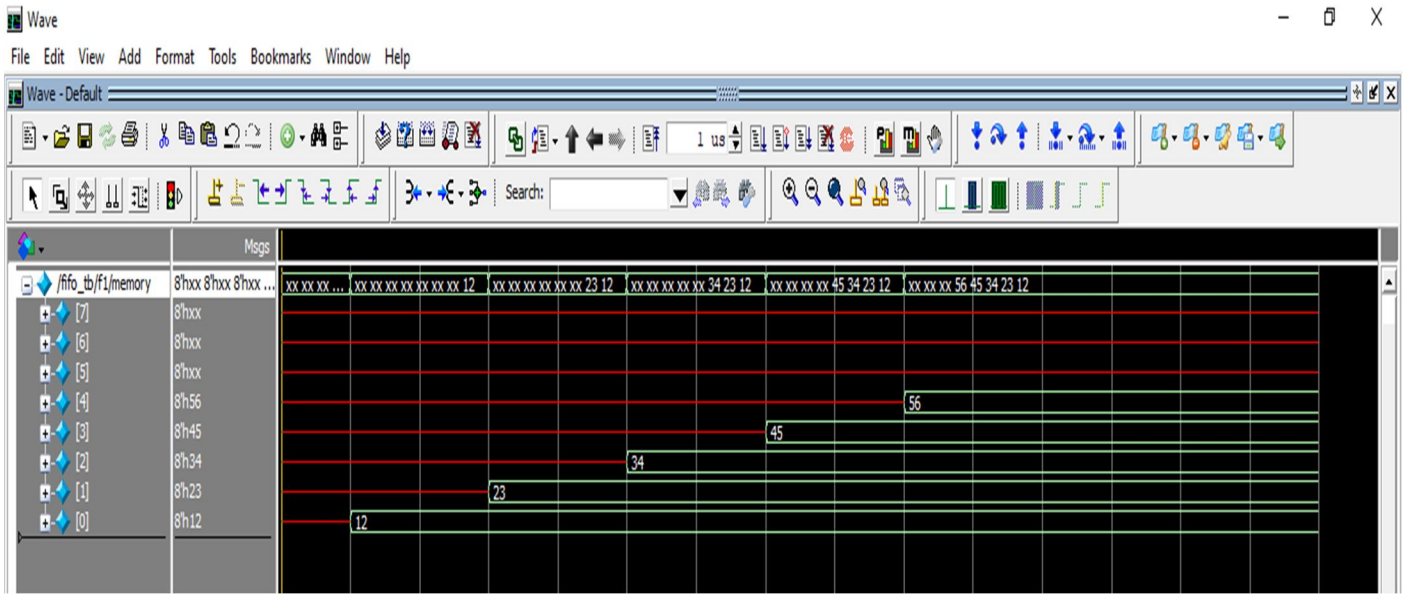
X. Verfication Using Modelsim For Reading All 8 Data And Checking Empty Flag Status



Y. Simulation Using Vivado For Writing 4 Data And Checking Half Write Flag



Z. Verfication Using Modelsim For Writing 4 Data And Checking Half Write Flag



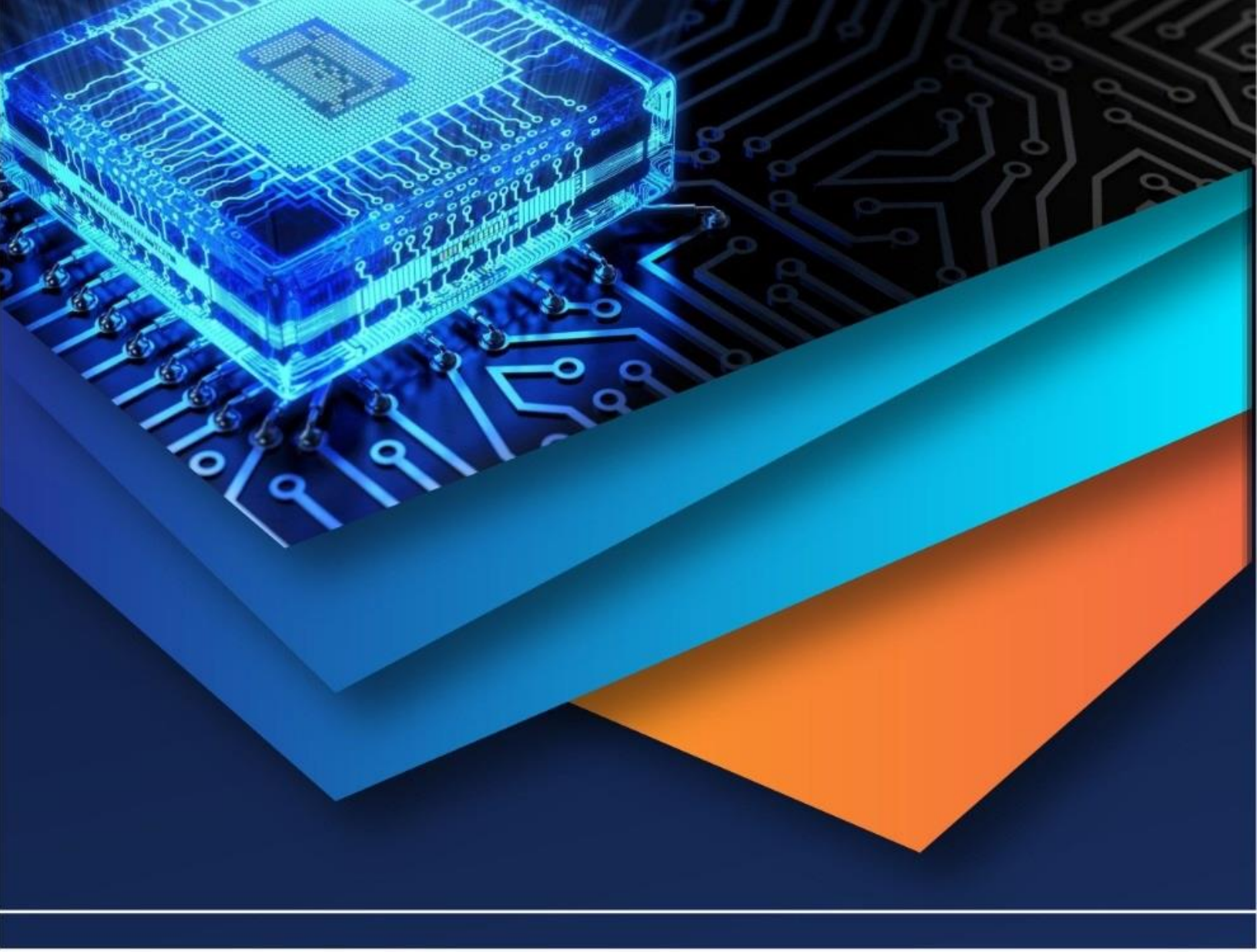
IV. RESULT ANALYSIS AND CONCLUSION

This paper deals with the creation of a synchronous FIFO module and its simulation using Xilinx Vivado and its verification using Modelsim. There are different test benches involved to check the functionality of the code. It tries to verify the correct functionality of data write and data read with proper flag triggers at the expected time. FIFOs are an ideal solution to the problem of moving data between a processor and peripheral device that either operate at different speeds, or use unsynchronized clock sources.

REFERENCES

[1] Miro Panades and A. Greiner. Bi-synchronous FIFO for synchronous circuit communication well suited for network-on-chip in GALS architectures. In First International Symposium on Networks-on-Chip (NOCS'07), pages 83–94, May 2007. doi:10.1109/NOCS.2007.14.

[2] Allen E. Sjogren and Chris J. Myers. Interfacing synchronous and asynchronous modules within a high-speed pipeline. IEEE Transactions on Very Large Scale Integration (VLSI) Systems, 8(5):573–583, October 2000. doi:10.1109/92.894162.



10.22214/IJRASET

45.98

IMPACT FACTOR: 7.129

IMPACT FACTOR: 7.429

INTERNATIONAL JOURNAL FOR RESEARCH

IN APPLIED SCIENCE & ENGINEERING TECHNOLOGY