CONTENTS:

ABSTRACT

First-In First-Out (FIFO) memory structures are widely used to buffer the transfer of data between processing blocks. High performance and high complexity digital systems increasingly are required to transfer data between modules of differing and even unrelated clock frequencies. This project presents an encompassing description of the motivation and design decisions for a robust and scalable FIFO architecture.The proposed design utilizes an efficient memory array structure and can be further modified to operate in applications where multiple clock cycles of latency exist between the data producer, FIFO, and the data consumer. In this project the objective is to design, verify and synthesize a synchronous FIFO using binary coded read and write pointers to address the memory array**.** The RTL description for the FIFO is written using Verilog HDL, and design is simulated and synthesized using NC-SIM and RTL compiler provided by Cadence Design Systems, Inc. respectively.

1.INTRODUCTION:

WHAT IS A FIFO (first in first out)?

In computer programming, FIFO (first-in, first-out) is an approach to handling program work requests from queues or stacks so that the oldest request is handled first. In hardware it is either an array of flops or Read/Write memory that store data given from one clock domain and on request supplies with the same data to other clock domain following the first in first out logic. The clock domain that supplies data to FIFO is often referred as WRITE OR INPUT LOGIC and the clock domain that reads data from the FIFO is often referred as READ OR OUTPUT LOGIC. FIFOs are used in designs to safely pass multi-bit data words from one clock domain to another or to control the flow of data between source and destination side sitting in the same clock domain. If read and write clock domains are governed by same clock signal the FIFO is said to be SYNCHRONOUS and if read and write clock domains are governed by different (asynchronous) clock signals FIFO is said to be ASYNCHRONOUS.

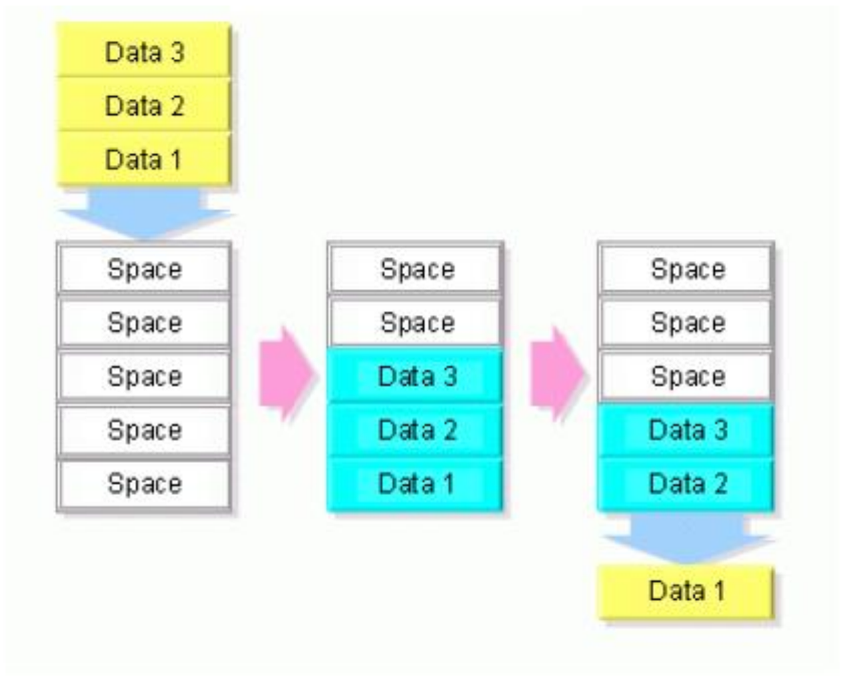


Figure 1: Data Flow through FIFO

FIFO full and *FIFO empty* flags are of great concern as no data should be written in full condition and no data should be read in empty condition, as it can lead to loss of data or generation of non relevant data. The full and empty conditions of FIFO are controlled using binary or gray pointers. In this report we deal with binary pointers only since we are designing SYNCHRONOUS FIFO. The gray pointers are used for generating *full* and *empty* conditions for ASYNCHRONOUS FIFO. The reason why they are used is beyond the scope of this document.

SYNCHRONOUS FIFO

A synchronous FIFO refers to a FIFO design where data values are written sequentially into a memory array using a clock signal, and the data values are sequentially read out from the memory array using the same clock signal. In synchronous FIFO the generation of *empty* and *full* flags is straight forward as there is no clock domain crossing involved. Considering this fact user can even generate programmable *partial empty* and *partial full* flags which are needed in many applications.

2.DESIGN OF FIFO:

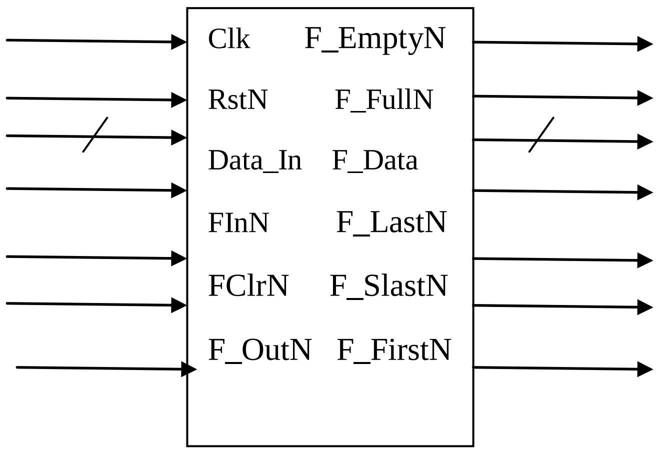


Figure 2: Blackbox diagram of FIFO

FIFO is a First-In-First-Out memory queue with control logic that manages the read and write operations, generates status flags, and provides optional handshake signals for interfacing with the user logic. It is often used to control the flow of data between source and destination. FIFO can be classified as synchronous or asynchronous depending on whether same clock or different (asynchronous) clocks control the read and write operations. In this project the objective is to design, verify and synthesize a synchronous FIFO using binary coded read and write pointers to address the memory array**.** FFIO full and empty flags are generated and passed on to source and destination logics, respectively, to pre-empt any overflow or underflow of data. In this way data integrity between source and destination is maintained**.**

A more efficient method to construct a FIFO is to create a circular buffer using an array of memory elements designed so that data can be written to and read from arbitrary locations in the memory array. These structures are also called parallel FIFOs and are often built using common SRAM or DRAM memories. The random access nature of memory arrays enables low minimum latencies, and high energy efficiencies compared to linear FIFOs. Scalability is also dramatically improved due to the fact that clock distribution is not directly affected by the FIFO size and the memory density is higher. In the event of a bu_er size change, generally only very minor control logic changes are required.

2.1 WORKING OF FIFO:

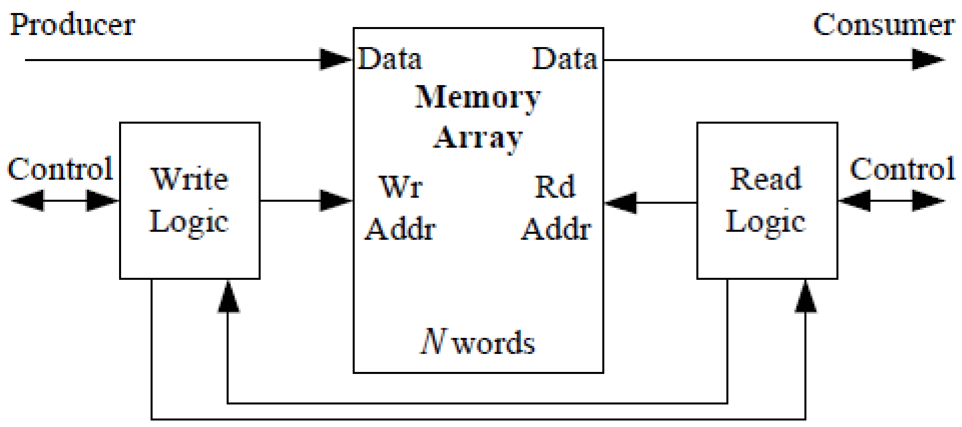


Figure 3: FIFO block diagram with write and read control logic

Figure 3 depicts the basic building blocks of a synchronous FIFO which are: ***memory array***, ***write control logic*** and ***read control logic***. The memory array can be implemented either with array of flip-flops or with a dual-port read/write memory. Both of these implementations allow simultaneous read and write accesses. This simultaneous access gives the FIFO its inherent synchronization property. There are no restrictions regarding timing between accesses of the two ports. This means simply, that while one port is writing to the memory at one rate, the other port can be reading at another rate totally independent of one another. The only restriction placed is that the simultaneous read and write access should not be from/to the same memory location(flowthrough condition).

The Synchronous FIFO has a single clock port *Clk* for both data-read and data-write operations. Data presented at the module's data-input port *Data_In* is written into the next available empty memory location on a rising clock edge when the write-enable input *FinN* is high. The full status output *F_FullN* indicates that no more empty locations remain in the module's internal memory. Data can be read out of the FIFO via the module's data-output port *F_Data* in the order in which it was written by asserting read-enable signal *F_OutN* prior to a rising clock edge. The memory-empty status output *F_EmptyN* indicates that no more data resides in the module's internal memory.

Note that the read and write control blocks in single-clock FIFOs share a common clock so their separation represents a functional division, not necessarily a physical one. The write circuitry controls data being written to the FIFO and tests whether the memory is full. The read circuity controls the flow of data out of the FIFO and is concerned with determining when the memory becomes empty.Control circuits for circular FIFOs are more complex than control circuits for linear FIFOs due to the need to manage non-regular write and read accesses with special cases when the array is full or empty.

WRITE CONTROL LOGIC

Write Control Logic is used to control the write operation of the FIFO's internal memory. It generates binary-coded write pointer which points to the memory location where the incoming data is to be written. Write pointer is incremented by one after every successful write operation. Additionally it generates FIFO full F_FullN, F_LastN (only single space remianing),F_SlastN(2 spaces remaining) signals which in turn are used to prevent any data loss. For example if a write request comes when FIFO is full then ***Write Control Logic*** stalls the write into the memory till the time *F_FullN* flag gets de-asserted. It intimates the stalling of write to source by not sending any acknowledgement in response to the write request.

READ CONTROL LOGIC

Read Control Logic is used to control the read operation of the FIFO's internal memory. It generates binary-coded read pointer which points to the memory location from where the data is to be read. Read pointer is incremented by one after every successful read operation. Additionally it generates FIFO empty and only one data value in FIFO, flags which in turn are used to prevent any spurious data read. For example if a read request comes when FIFO is empty then ***Read Control Logic*** stalls the read from the memory till the time *F_EmptyN* flag gets de-asserted.

MEMORY ARRAY:FIFO MEM BLOCK

Memory Array is an array of flip-flops which stores data. Number of data words that the memory array can store is often referred as DEPTH of the FIFO. Length of the data word is referred as WIDTH of the FIFO. Besides flop-array it comprises read and write address decoding logic.

The functionality of ***Memory Array*** is relatively straight forward as described below:

1. If *write enable* signal is high DATA present on *write data* is written into the row pointed by *write addr* on the next rising edge of the clock signal *clk*. Note that *write enable* is asserted only when *wdata valid* is high and FIFO is not full to avoid any data corruption
2. If *read enable* signal is high the DATA present in the row pointed by *Read addr* is sent onto the *read data* bus on the next rising edge of the clock signal *clk.* Note that *read enable* is asserted only when *read req* is high and FIFO is not empty to avoid any spurious data being sent to the requestor
3. It can handle simultaneous read and write enables as long as their addresses do not match

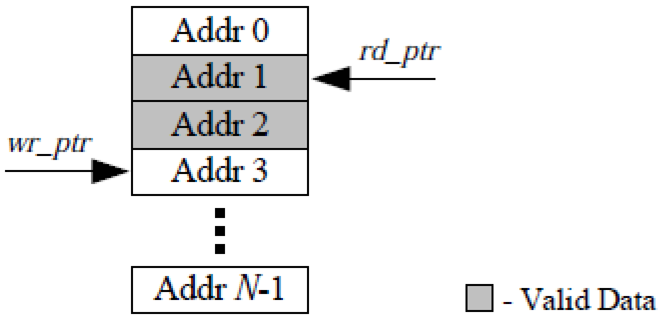


Figure 4:write and read address pointer scheme

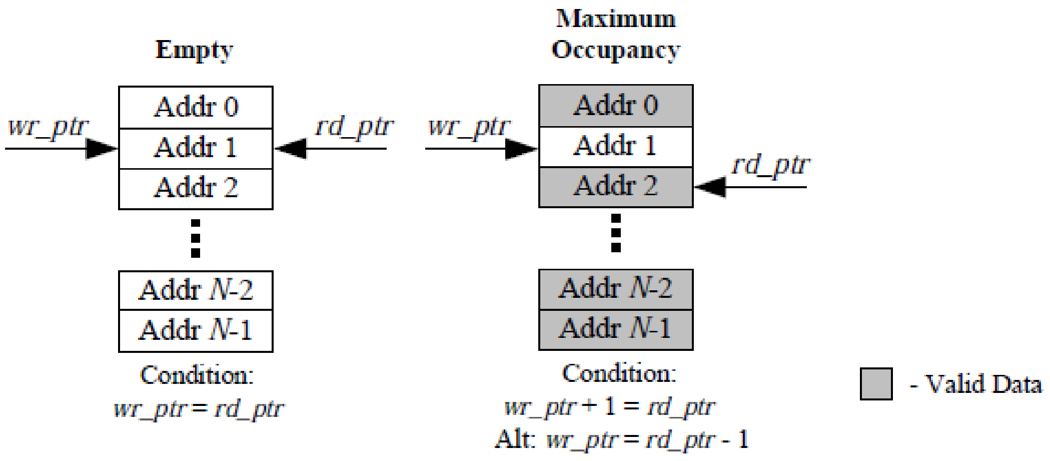


Figure 5:FIFO defined as empty if wr_ptr= rd_ptr

All parallel FIFOs must keep track of three mutually-exclusive states: 1) empty (also the initial state), 2) full, and 3) data occupancy between empty and full. Tracking valid data within a FIFO is typically accomplished with one of two approaches. The first method involves using an N-bit register(counter) where each bit represents the validity of data in the memory and N is the number of words in the memory. A second method is to maintain read and write (or head and tail) address pointers which indicate the beginning and end of the valid data range in the memory. Figure 2.5 shows a scheme where the write address pointer (wr ptr) indicates the memory location for the next data write, and the read address pointer (rd ptr) indicates the location of the next memory read. Other schemes are possible, the most common of which offset their pointers by one memory location from the example shown here.

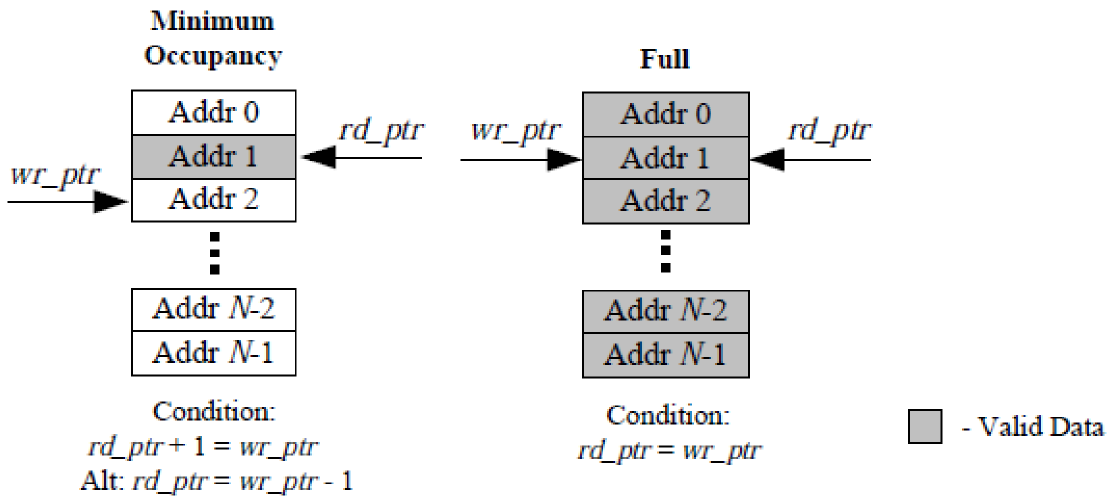


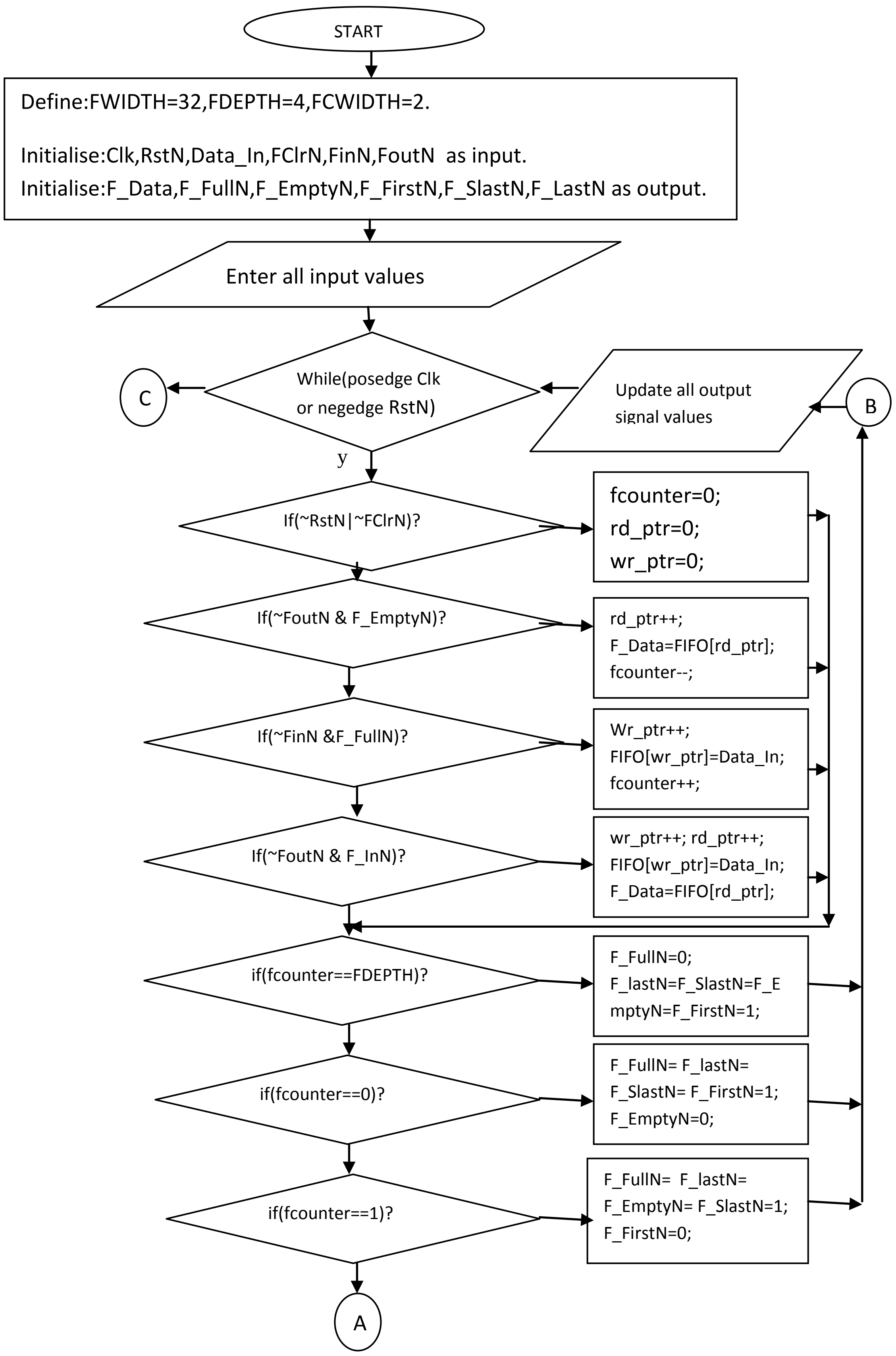
Figure 6: FIFO defined as full when rd_ptr=wr_ptr

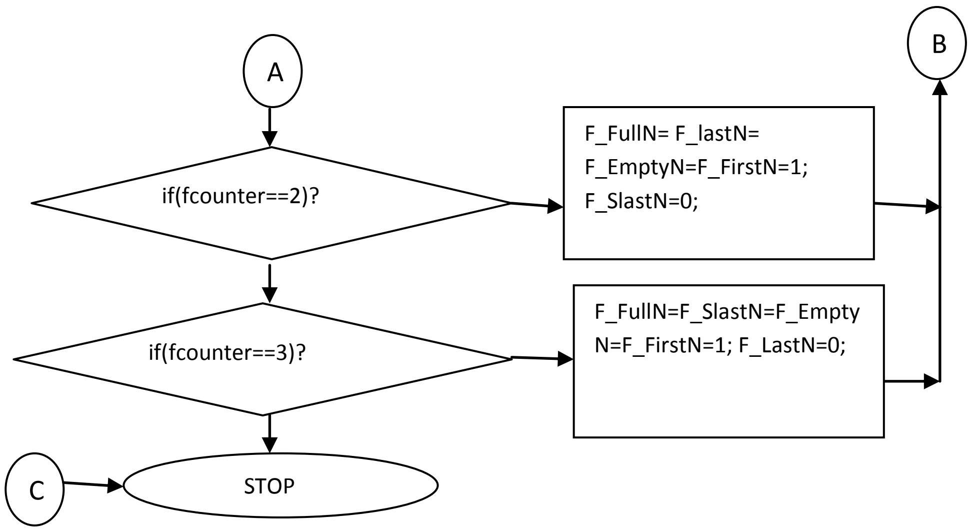
Given the use of read and write pointers, there are two primary methods to define the empty and full conditions. As shown in Figure 2.6, the first possibility is to define the empty condition as occurring when wr ptr is equal to rd ptr. The “maximum occupancy" (full) condition is indicated when wr ptr + 1 = rd ptr or alternatively, when wr ptr = rd ptr -1. The second case is shown in Figure 2.7 where the full condition is indicated by equality of rd ptr and wr ptr and the “minimum occupancy" (one datum) condition is indicated by rd ptr = wr ptr +1. Clearly, these schemes present problems in representing all possible states because the case where rd ptr = wr ptr becomes ambiguous without either keeping track of the pointer history or preventing the pointers from reaching this state in either the full or empty condition as mentioned above.

PORT LIST:

PIN NO.	PIN NAME	SIAGNAL TYPE	INDICATION
1	Clk	active high input	clock signal is provided to synchronise all operations in the FIFO.
2	RstN	active low input	this signal when asserted low, resets the fifo. All counters are set to 0.
3	Data_In	32-bit input	it is a 32-bit data input into FIFO.
4	FInN	active low input	write into FIFO signal
5	FOutN	active low input	read from FIFO signal
7	F_EmptyN	active low output	signal indicating that FIFO is empty
8	F_FullN	active low output	signal indicating that FIFO is full
9	F_LastN	active low output	signal indicating that FIFO has space for one last data value
10	F_SlastN	active low output	signal indicating that FIFO has space for two data values
11	F_FirstN	active low output	signal indicating that there is only one data value in FIFO.
12	F_Data	32-bit output	it is 32-bit data output from FIFO.

3.ALGORITHM FLOW CHART:


START
Define:FWIDTH=32,FDEPTH=4,FCWIDTH=2.
Initialise:Clk,RstN,Data_In,FClrN,FinN,FoutN as input.
Initialise:F_Data,F_FullN,F_EmptyN,F_FirstN,F_SlastN,F_LastN as output.
Enter all input values
While(posedge Clk or negedge RstN)
C
Update all output signal values
B
y
If(~RstN|~FClrN)?
fcounter=0; rd_ptr=0; wr_ptr=0;
If(~FoutN & F_EmptyN)?
rd_ptr++; F_Data=FIFO[rd_ptr]; fcounter--;
If(~FinN &F_FullN)?
Wr_ptr++; FIFO[wr_ptr]=Data_In; fcounter++;
If(~FoutN & F_InN)?
wr_ptr++; rd_ptr++; FIFO[wr_ptr]=Data_In; F_Data=FIFO[rd_ptr];
if(fcounter==FDEPTH)?
F_FullN=0; F_lastN=F_SlastN=F_EmptyN=F_FirstN=1;
if(fcounter==0)?
F_FullN= F_lastN= F_SlastN= F_FirstN=1; F_EmptyN=0;
if(fcounter==1)?
F_FullN= F_lastN= F_EmptyN= F_SlastN=1; F_FirstN=0;
A


A
if(fcounter==2)?
F_FullN= F_lastN=
F_EmptyN=F_FirstN=1;
F_SlastN=0;
B
if(fcounter==3)?
F_FullN=F_SlastN=F_Empty
N=F_FirstN=1; F_LastN=0;
C
STOP

4.VERILOG SOURCE CODE FOR FIFO DESIGN:

Memory block module:

```
`resetall
`timescale 1ns/1ns
`view vlog
`define FWIDTH 32
`define FDEPTH 4
`define FCWIDTH 2

module FIFO_MEM_BLK(clk,writeN,wr_addr, rd_addr, data_in,data_out);

input clk;       //input clk
input writeN;    //write signal to put data into FIFO
input [(`FCWIDTH-1):0] wr_addr;   //write address
input [(`FCWIDTH-1):0] rd_addr;   //read address
input [(`FWIDTH-1):0]  data_in;    //datain in to memory block
output [(`FWIDTH-1):0]   data_out;  //data out from the memory block
FIFO

wire [(`FWIDTH-1):0] data_out;
reg [(`FWIDTH-1):0] FIFO [0:(`FDEPTH-1)];

assign data_out =FIFO[rd_addr];

always@(posedge clk)
begin
if(writeN==1'b0)
FIFO[wr_addr]<= data_in;
end
endmodule
`noview
```

FIFO module:

```
`timescale 1ns / 1ns
`define FWIDTH 32       //width of the FIFO
`define FDEPTH 4        //depth of the FIFO
`define FCWIDTH 2       //counter width of the FIFO 2 to power FCWIDTH


module fifo(Clk,RstN,Data_In,FClrN,FInN,FOutN,
F_Data,F_FullN,F_LastN,F_SLastN,F_FirstN,F_EmptyN);

    input Clk;                              //clk signal

    input RstN;                             //low asserted reset signal

    input [(`FWIDTH-1):0] Data_In;          //data into FIFO

    input FClrN;                            //clear signal to FIFO

    input FInN;                             //write into FIFO signal

    input FOutN;                            //read from FIFO signal
    output [(`FWIDTH-1):0] F_Data;      // FIFO data out

    output F_FullN;                  //FIFO full indicating signal

    output F_LastN;                  //FIFO last but one signal

    output F_SLastN;                 //FIFO Slast but one signal

    output F_EmptyN;                 //FIFO empty indicating signal

    output F_FirstN;              // signal indicating only 1word in
FIFO


    reg F_FullN;

    reg F_EmptyN;

    reg F_LastN;

    reg F_SLastN;

    reg F_FirstN;

    reg [`FCWIDTH:0] fcounter;    //counter indicates num of data in
FIFO

    reg [(`FCWIDTH-1):0] rd_ptr;        // current read pointer

    reg [(`FCWIDTH-1):0] wr_ptr;        //current write pointer

    wire [(`FWIDTH -1):0] FIFODataOut;   //data out from FIFO MemBlk

    wire [(`FWIDTH-1):0] FIFODataIn;    //data into FIFO MemBlk

    wire ReadN=FOutN;

   wire WriteN=FInN;


 assign F_Data = FIFODataOut;
 assign FIFODataIn = Data_In;


`uselib view=vlog
      FIFO_MEM_BLK memblk(.clk(Clk),

                      .writeN(WriteN),
                      .rd_addr(rd_ptr),
                      .wr_addr(wr_ptr),
                      .data_in(FIFODataIn),
                      .data_out(FIFODataOut)
                      );

`nouselib
```

```
always@(posedge Clk or negedge RstN)
begin
if(!RstN) begin
fcounter <= 0;
rd_ptr <= 0;
wr_ptr <= 0;
end
     else begin
          if(!FClrN) begin
            fcounter <= 0;
            rd_ptr <= 0;
            wr_ptr <= 0;
          end

            else begin
                 if (!WriteN && F_FullN)
                     wr_ptr <= wr_ptr + 1;
                 if(!ReadN && F_EmptyN)
                    rd_ptr <= rd_ptr + 1;
                 if(!WriteN && ReadN && F_FullN)
                   fcounter <= fcounter + 1;
                 else if(WriteN && !ReadN && F_EmptyN)
                        fcounter <= fcounter -1;
                 end

            end

     end

always@(posedge Clk or negedge RstN)
begin

if(!RstN)
F_EmptyN <= 1'b0;


else begin
     if(FClrN== 1'b1)
         begin
         if(F_EmptyN==1'b0 && WriteN==1'b0)
            F_EmptyN <= 1'b1;
         else if(F_FirstN== 1'b0 && ReadN== 1'b0 && WriteN== 1'b1)
                 F_EmptyN <= 1'b0;
         end
     else
         F_EmptyN <= 1'b0;
     end

end

always@(posedge Clk or negedge RstN)
begin
if(!RstN)
F_FirstN <= 1'b1;
else begin
     if(FClrN== 1'b1) begin
                     if((F_EmptyN==1'b0 && WriteN==1'b0)||(fcounter==2
&& ReadN==1'b0 && WriteN==1'b1))
```

```
                        F_FirstN <= 1'b0;
                     else if (F_FirstN==1'b0 && (WriteN ^ ReadN ))
                              F_FirstN <= 1'b1;
                     end

     else begin
          F_FirstN <= 1'b1;
     end
end
end


always@(posedge Clk or negedge RstN)
begin
if(!RstN)
  F_SLastN <= 1'b1;
else begin
     if(FClrN==1'b1) begin
                     if((F_LastN==1'b0 && ReadN==1'b0 && WriteN==
1'b1)||(fcounter==(`FDEPTH -3) && WriteN==1'b0 && ReadN==1'b1))
                        F_SLastN <= 1'b0;
                     else if (F_SLastN==1'b0 && (ReadN ^ WriteN))
                              F_SLastN <= 1'b1;
                     end
      else
             F_SLastN <= 1'b1;

      end
end



always@(posedge Clk or negedge RstN)
begin
if(!RstN)
F_LastN <= 1'b1;
else begin
     if(FClrN== 1'b1)begin

                     if((F_FullN== 1'b0 && ReadN==
1'b0)||(fcounter==(`FDEPTH-2) && WriteN==1'b0 && ReadN==1'b1))
                        F_LastN <= 1'b0;
                     else if(F_LastN==1'b0 && (ReadN ^ WriteN))
                             F_LastN <=1'b1;
                     end

     else
         F_LastN <= 1'b1;
     end
end


always@(posedge Clk or negedge RstN)
begin
if(!RstN)
F_FullN <= 1'b1;
else begin
     if(FClrN==1'b1) begin
```

```
                      if(F_LastN==1'b0 && WriteN==1'b0 && ReadN==1'b1)
                         F_FullN <= 1'b0;
                      else if(F_FullN==1'b0 && ReadN==1'b0)
                              F_FullN <= 1'b1;
                      end
     else
         F_FullN <= 1'b1;
     end
end

endmodule
```

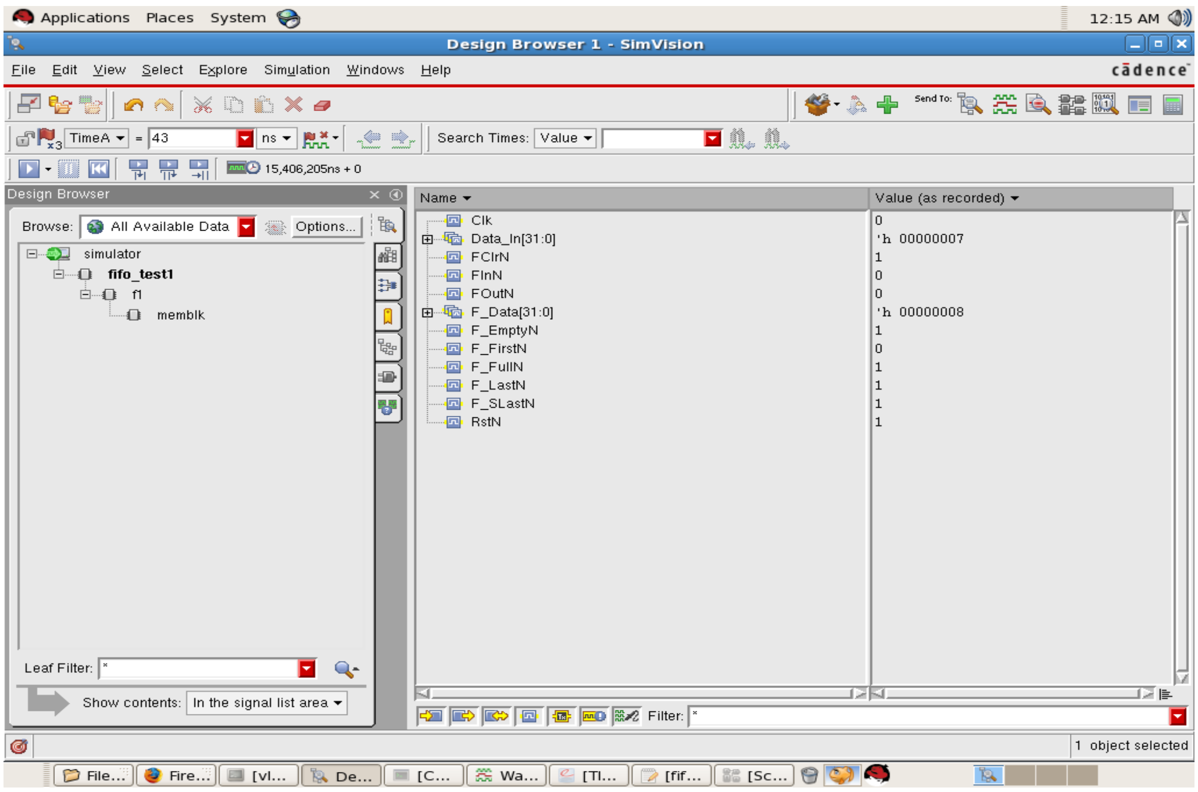
5.TEST BENCH FOR FIFO:

```
`resetall
`timescale 1ns/1ns
`view vlog
module fifo_test1;
reg Clk,RstN,FClrN,FInN,FOutN;
reg [31:0] Data_In;
wire F_FullN,F_LastN,F_SLastN,F_EmptyN,F_FirstN ;
wire [31:0] F_Data;
`uselib view=vlog
fifo f1(Clk,RstN,Data_In,FClrN,FInN,FOutN,
F_Data,F_FullN,F_LastN,F_SLastN,F_FirstN,F_EmptyN);
`nouselib

initial Clk = 0;
always #5 Clk = ~Clk;


initial
       begin

RstN =1'b0;FClrN=1'b1;FInN=1'b1;FOutN=1'b1; Data_In=32'd0;#10;  //reset
condition check
RstN =1'b1;FClrN=1'b1;FInN=1'b0;FOutN=1'b0; Data_In=32'd7;#10;  //read
& write check
RstN =1'b1;FClrN=1'b1;FInN=1'b0;FOutN=1'b1; Data_In=32'd10;#10; //
write into fifo 1st data word
RstN =1'b1;FClrN=1'b1;FInN=1'b0;FOutN=1'b1; Data_In=32'd9;#10;  //write
into fifo 2nd data word
RstN =1'b1;FClrN=1'b1;FInN=1'b0;FOutN=1'b1; Data_In=32'd8;#10;  //
write 3rd data word ,check for F_SlastN flag
RstN =1'b1;FClrN=1'b1;FInN=1'b0;FOutN=1'b1; Data_In=32'd7;#10;  //
write last word ,check last word flag  & fifo full flag
RstN =1'b1;FClrN=1'b1;FInN=1'b1;FOutN=1'b0; Data_In=32'd6;#10;  // read
from fifo 1st word stored
RstN =1'b1;FClrN=1'b1;FInN=1'b1;FOutN=1'b0; Data_In=32'd5;#10;  // read
second word
RstN =1'b1;FClrN=1'b1;FInN=1'b1;FOutN=1'b0; Data_In=32'd4;#10;  //read
3rd word
RstN =1'b1;FClrN=1'b1;FInN=1'b1;FOutN=1'b0; Data_In=32'd3;#10;  // read
lst word check fifo empty flag
RstN =1'b1;FClrN=1'b0;FInN=1'b1;FOutN=1'b1; Data_In=32'd2;#10;   //
clear  fifo
$display(" ",  Clk,RstN,FClrN,FInN,FOutN, Data_In,F_FullN,F_LastN,
F_SLastN,F_FirstN,F_EmptyN,F_Data);


end
endmodule

`noview
```

6.COMPILING, ELABORATING & SIMULATING THE DESIGN:

```
[vlsi@305LABSYS06 ~]$ csh
[vlsi@305LABSYS06 ~]$ source cshrcclient
Welcome to Cadence Tools
[vlsi@305LABSYS06 ~]$ ls
a                            dfIIabstract.log1
ade_viva.log                 encounter.cmd
ade_viva.log.cdslck          encounter.cmd1
adtem.v~                     encounter.log
alib-52                      encounter.log1
async.v~                     filenames_2762_D20101118.log
cadence                      filenames_2763_D20090720.log
Cadence_digital_labs         filenames_3179_D20101201.log
cadence_ms_labs_614          filenames.log
cadence_ms_labs_614.tar.gz   INCA_libs
cds.lib                      invr3_test.v
cdsLibEditor.log             invr3.v
CDS.log                      nclaunch.key
CDS.log.1                    NCO
CDS.log.1.cdslck             NCO.tar.gz
CDS.log.2                    ncvlog.log
CDS.log.2.cdslck             padder.v~
CDS.log.3                    panic.log
CDS.log.3.cdslck             patches
Chaitra                      pt_px_tutorial
command.log                  simulation
cshrcclient                  synopsys_cache_B-2008.09-SP3
default.svf                  tg_test.v
Desktop                      vin
[vlsi@305LABSYS06 ~]$ cd Cadence_digital_labs
[vlsi@305LABSYS06 ~/Cadence_digital_labs]$ cd workarea
[vlsi@305LABSYS06 workarea]$ ncvlog fifo_mem_blk.v
ncvlog: 09.20-s045: (c) Copyright 1995-2011 Cadence Design Systems,
Inc.
file: fifo_mem_blk.v
        module designlib.FIFO_MEM_BLK:vlog
                errors: 0, warnings: 0
[vlsi@305LABSYS06 workarea]$ ncvlog fifo.v
ncvlog: 09.20-s045: (c) Copyright 1995-2011 Cadence Design Systems,
Inc.
file: fifo.v
        module designlib.fifo:vlog
                errors: 0, warnings: 0
[vlsi@305LABSYS06 workarea]$ ncvlog fifo_test1.v
ncvlog: 09.20-s045: (c) Copyright 1995-2011 Cadence Design Systems,
Inc.
file: fifo_test1.v
        module designlib.fifo_test1:vlog
                errors: 0, warnings: 0
[vlsi@305LABSYS06 workarea]$ ncelab fifo_test1 -access +rwc -mess
ncelab: 09.20-s045: (c) Copyright 1995-2011 Cadence Design Systems,
Inc.
        Elaborating the design hierarchy:
                Caching library 'lab6_padder_lib' ....... Done
                Caching library 'designlib' ....... Done
        Building instance overlay tables: .................... Done
        Generating native compiled code:
                designlib.FIFO_MEM_BLK:vlog <0x2dec053c>
                        streams:   3, words:   864
                designlib.fifo:vlog <0x7d3be221>
                        streams:  15, words:  5943
                designlib.fifo_test1:vlog <0x693f954c>
                        streams:   9, words: 11647
        Loading native compiled code:     .................... Done
        Building instance specific data structures.
        Design hierarchy summary:
                                 Instances  Unique
                Modules:                 3       3
                Registers:              15      15
                Scalar wires:           12       -
                Vectored wires:          6       -
                Always blocks:           8       8
                Initial blocks:          2       2
                Cont. assignments:       5       5
                Pseudo assignments:      8       8
                Simulation timescale:  1ns
        Writing initial simulation snapshot: designlib.fifo_test1:vlog
[vlsi@305LABSYS06 workarea]$ ncsim fifo_test1 -gui
ncsim: 09.20-s045: (c) Copyright 1995-2011 Cadence Design Systems, Inc.
simvision: 09.20-s045: (c) Copyright 1995-2011 Cadence Design Systems,
Inc.
txe: 09.20-s045: (c) Copyright 1995-2009 Cadence Design Systems, Inc.
```

DESIGN BROWSER:

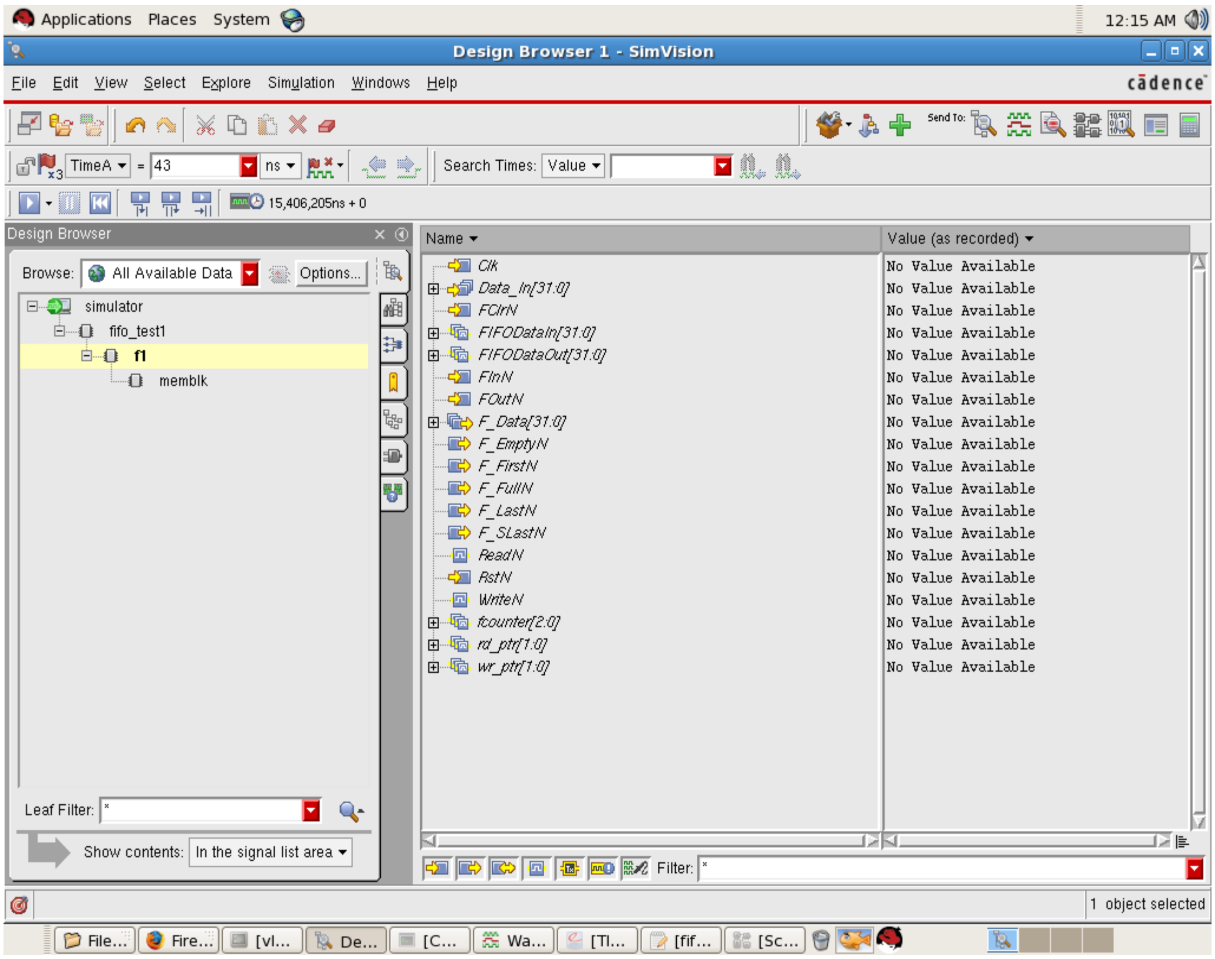

Applications Places System
12:15 AM
Design Browser 1 - SimVision
File Edit View Select Explore Simulation Windows Help
cadence
Send To:
Design Browser
Options...
Show contents:
File... Fire... [vl... De... [C... Wa... [Tl... [fif... [Sc...

TimeA = 43 ns
Search Times: Value
15,406,205ns + 0
Browse: All Available Data

```
simulator
  fifo_test1
    f1
      memblk
```

Leaf Filter: *
Show contents: In the signal list area
Filter: *
1 object selected

```
Name            Value (as recorded)
Clk             0
Data_In[31:0]   'h 00000007
FClrN           1
FInN            0
FOutN           0
F_Data[31:0]    'h 00000008
F_EmptyN        1
F_FirstN        0
F_FullN         1
F_LastN         1
F_SLastN        1
RstN            1
```

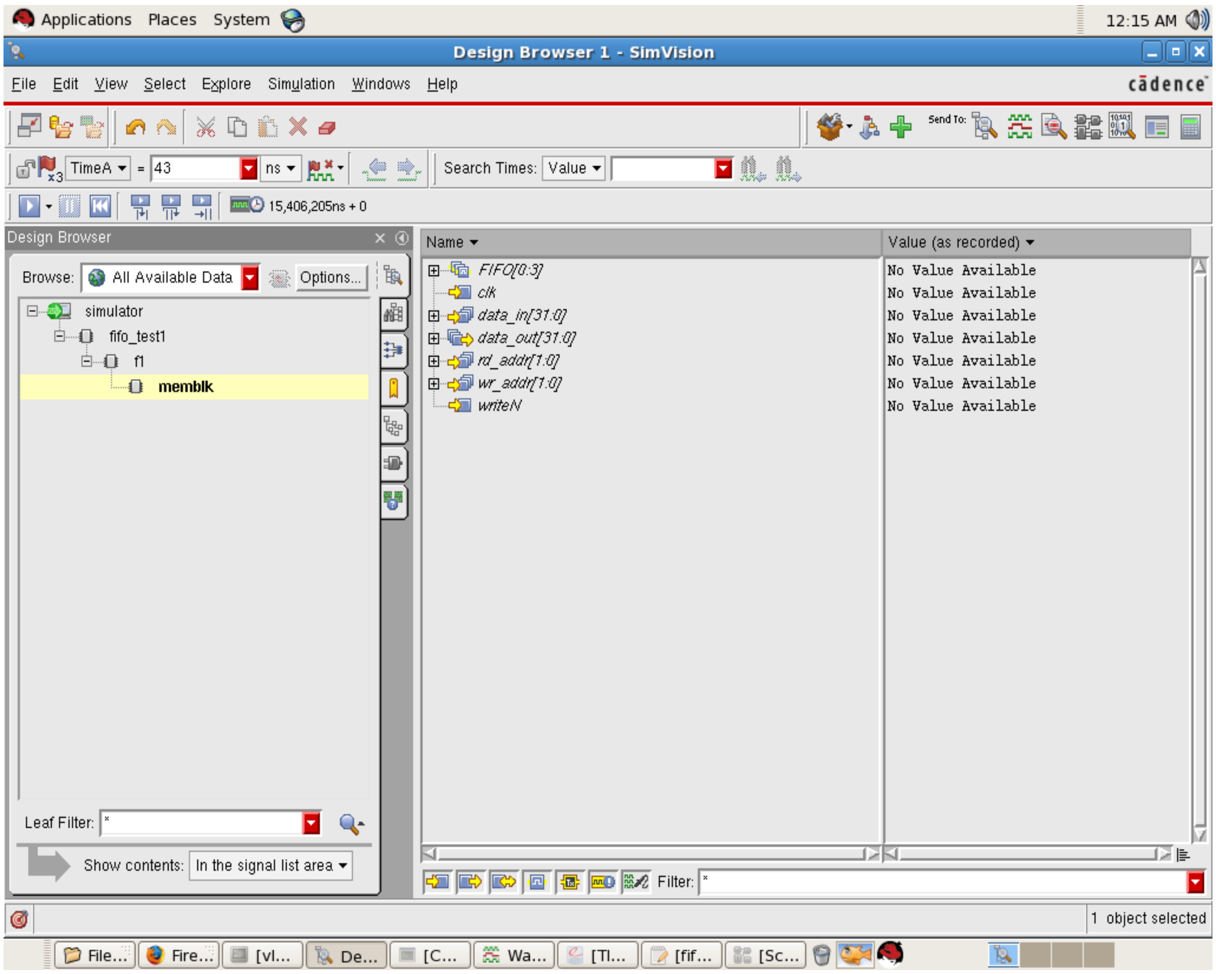

Applications Places System
12:15 AM
Design Browser 1 - SimVision
File Edit View Select Explore Simulation Windows Help
cadence
Send To:
Design Browser
Options...
Show contents:
File... Fire... [vl... De... [C... Wa... [Tl... [fif... [Sc...

TimeA = 43 ns
Search Times: Value
15,406,205ns + 0
Browse: All Available Data

```
simulator
  fifo_test1
    f1
      memblk
```

Leaf Filter: *
Show contents: In the signal list area
Filter: *
1 object selected

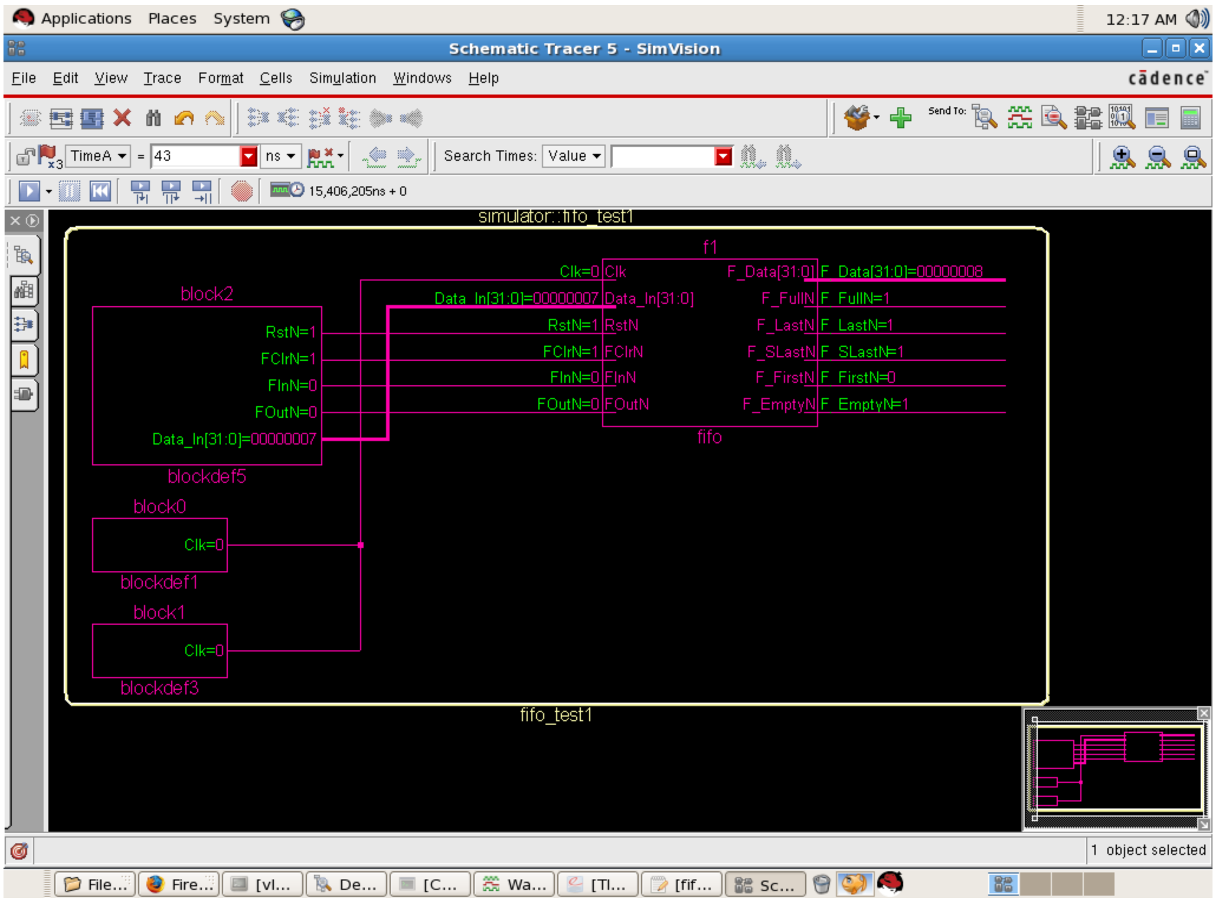
```
Name                Value (as recorded)
Clk                 No Value Available
Data_In[31:0]       No Value Available
FClrN               No Value Available
FIFODataIn[31:0]    No Value Available
FIFODataOut[31:0]   No Value Available
FInN                No Value Available
FOutN               No Value Available
F_Data[31:0]        No Value Available
F_EmptyN            No Value Available
F_FirstN            No Value Available
F_FullN             No Value Available
F_LastN             No Value Available
F_SLastN            No Value Available
ReadN               No Value Available
RstN                No Value Available
WriteN              No Value Available
fcounter[2:0]       No Value Available
rd_ptr[1:0]         No Value Available
wr_ptr[1:0]         No Value Available
```

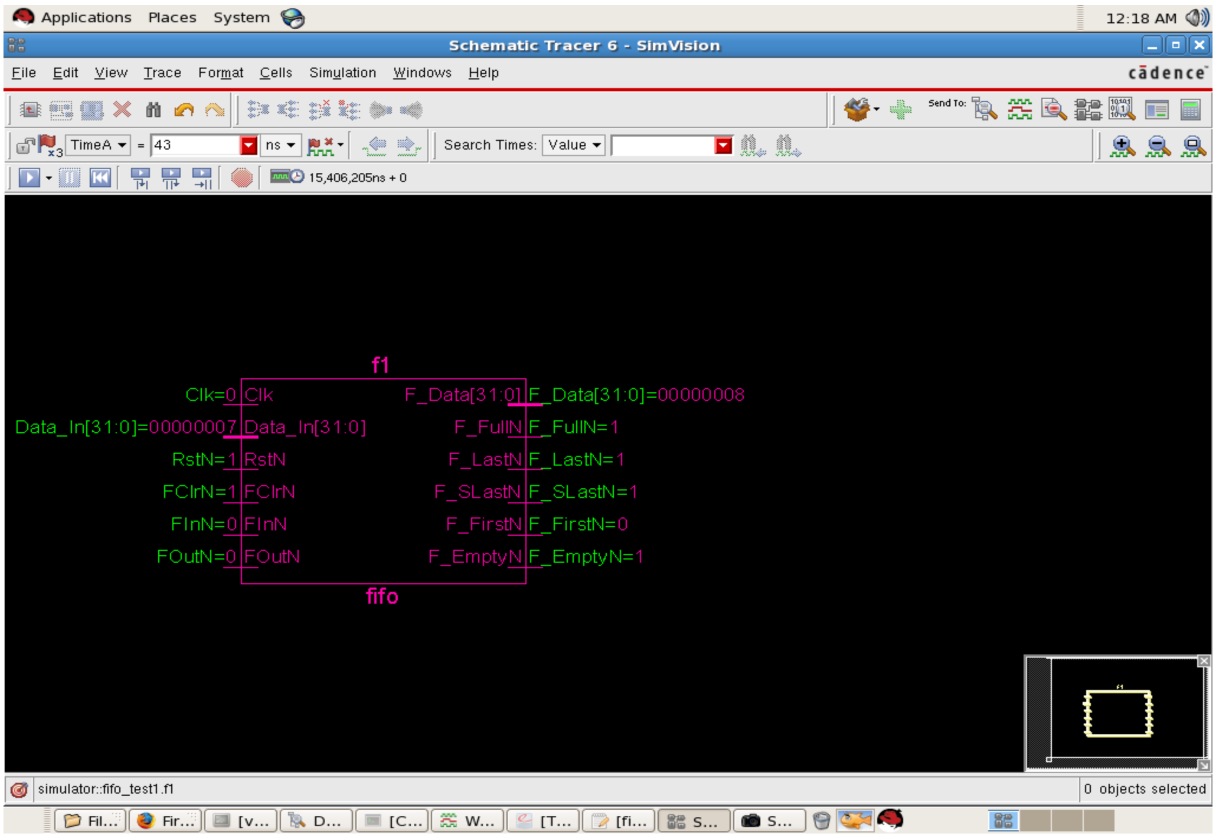

Applications Places System
12:15 AM
Design Browser 1 - SimVision
File Edit View Select Explore Simulation Windows Help
cādence
TimeA = 43 ns
Search Times: Value
15,406,205ns + 0
Design Browser
Browse: All Available Data
Options...
Leaf Filter: *
Show contents: In the signal list area
Filter: *
1 object selected

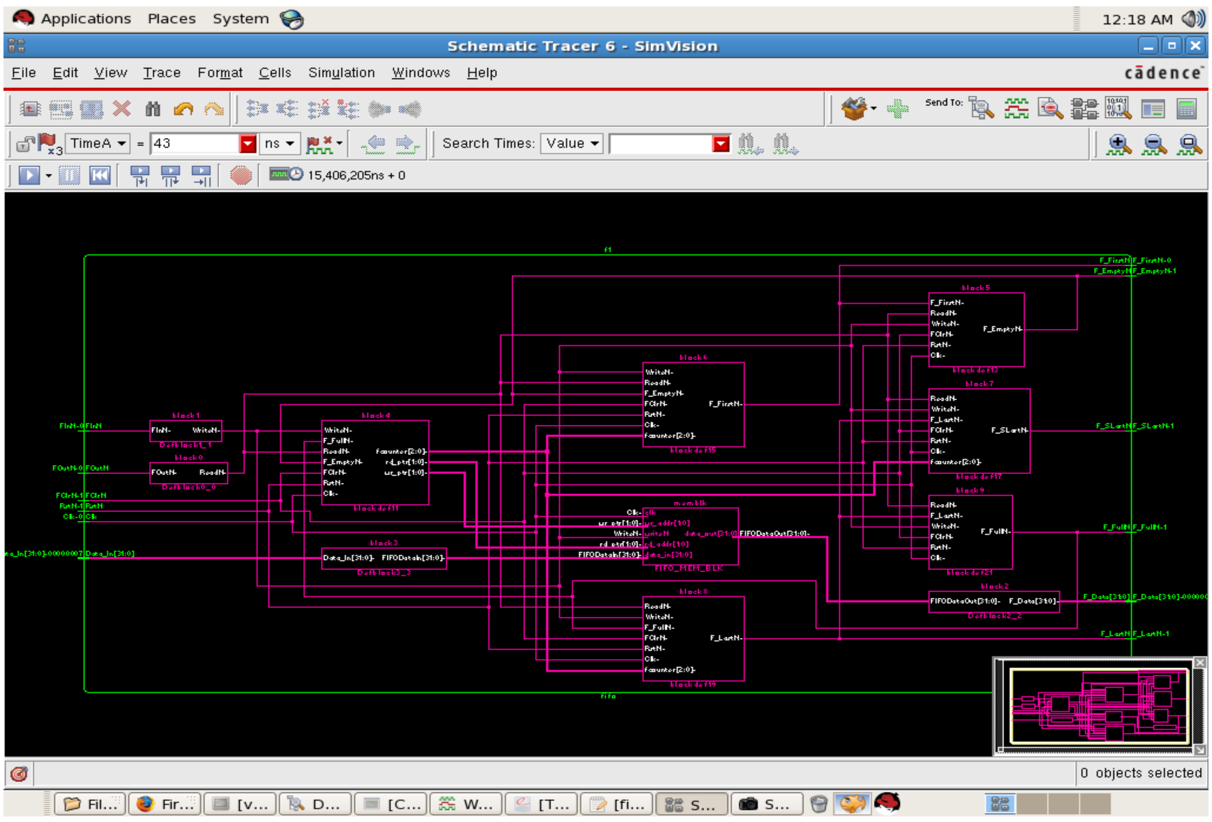
```
simulator
  fifo_test1
    f1
      memblk
```

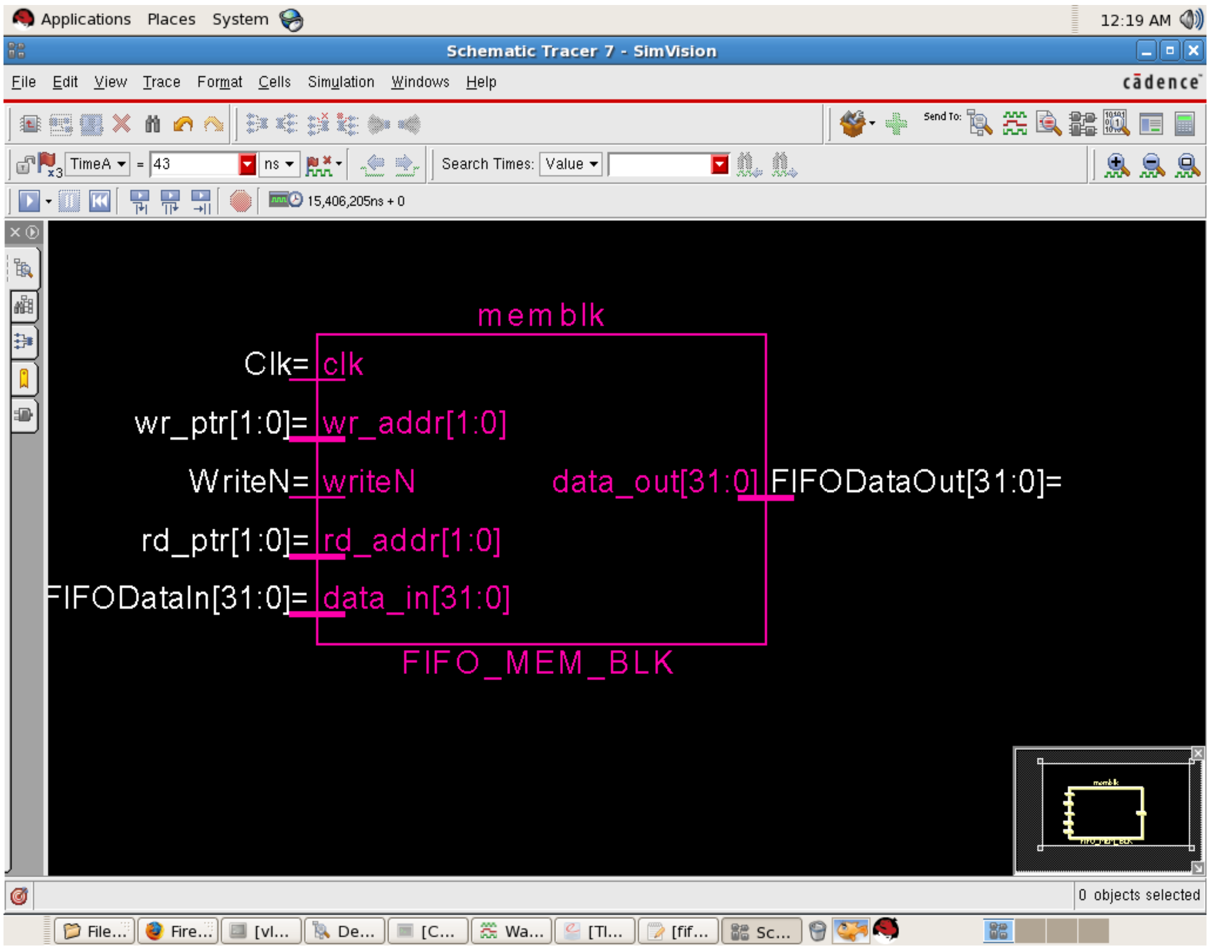
Name	Value (as recorded)
FIFO[0:3]	No Value Available
clk	No Value Available
data_in[31:0]	No Value Available
data_out[31:0]	No Value Available
rd_addr[1:0]	No Value Available
wr_addr[1:0]	No Value Available
writeN	No Value Available

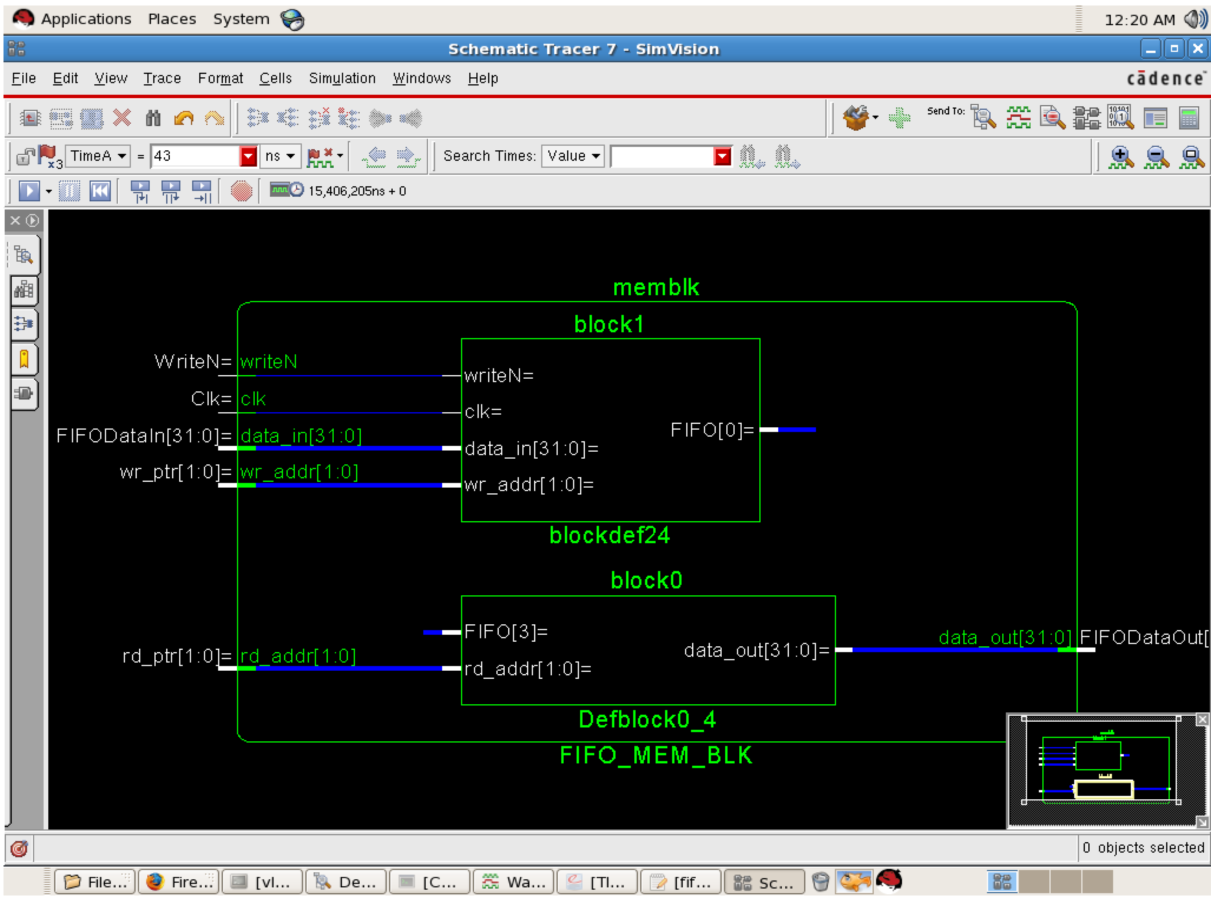
SCHEMATIC VIEW:



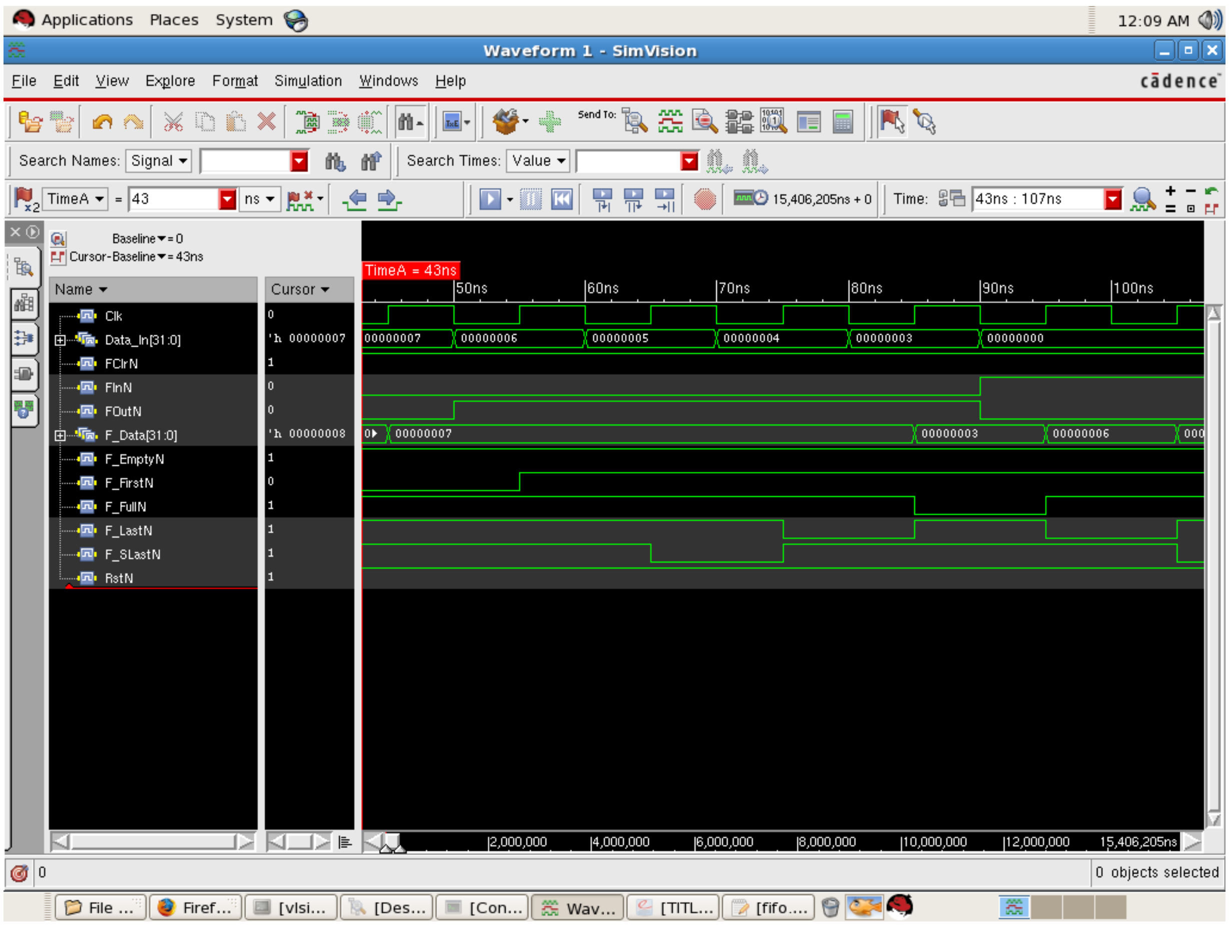



Applications Places System
12:18 AM
Schematic Tracer 6 - SimVision
File Edit View Trace Format Cells Simulation Windows Help
cadence
Send To:
TimeA = 43 ns
Search Times: Value
15,406,205ns + 0
fifo
block1
block0
block4
block3
block5
block6
block7
block8
block9
block2
memblk
FIFO_MEM_BLK
0 objects selected


Applications Places System
12:19 AM
Schematic Tracer 7 - SimVision
File Edit View Trace Format Cells Simulation Windows Help
cadence
Send To:
TimeA = 43 ns
Search Times: Value
15,406,205ns + 0
memblk
Clk= clk
wr_ptr[1:0]= wr_addr[1:0]
WriteN= writeN
rd_ptr[1:0]= rd_addr[1:0]
FIFODataIn[31:0]= data_in[31:0]
data_out[31:0] FIFODataOut[31:0]=
FIFO_MEM_BLK
0 objects selected


Applications Places System
12:20 AM
Schematic Tracer 7 - SimVision
File Edit View Trace Format Cells Simulation Windows Help
cadence
Send To:
TimeA = 43 ns
Search Times: Value
15,406,205ns + 0
memblk
block1
WriteN= writeN
Clk= clk
FIFODataIn[31:0]= data_in[31:0]
wr_ptr[1:0]= wr_addr[1:0]
writeN=
clk=
data_in[31:0]=
wr_addr[1:0]=
FIFO[0]=
blockdef24
block0
FIFO[3]=
rd_addr[1:0]=
data_out[31:0]=
rd_ptr[1:0]= rd_addr[1:0]
data_out[31:0] FIFODataOut[
Defblock0_4
FIFO_MEM_BLK
0 objects selected
File... Fire... [vl... De... [C... Wa... [Tl... [fif... Sc...

7.SIMULATION WAVEFORM:



8.APPLICATIONS:

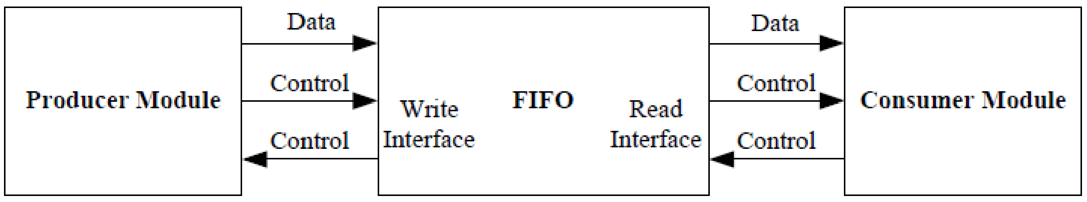


Figure 7:Sytem architecture using FIFO module for communication

1.**FIFO module for communication**:FIFO's are used to safely pass data between two asynchronous clock domains. In System-on-Chip designs there are components which often run on different clocks. So, to pass data from one such component to another we need ASYNCHRONOUS FIFO Some times even if the Source and Requestor sides are controlled by same clock signal a FIFO is needed. This is to match the throughputs of the Source and the Requestor. For example in one case source may be supplying data at rate which the requestor can not handle or in other case requestor may be placing requests for data at a rate at which source can not supply. So, to bridge this gap between source and requestor capacities to supply and consume data a SYNCHRONOUS FIFO is used which acts as an elastic buffer.

2. **Asynchronous Operation of Exclusive Read/Write FIFOs:** In use, the SN74LS224A exclusive read/write FIFO timing conditions on the write (WRITE CLOCK) and read (READ CLOCK) inputs must be maintained to ensure proper functioning of the devices. Concurrent read/write FIFOs are also offered on the market that, for their empty or full state, require timing conditions between the write and read inputs that ensure error-free operation of the FULL and EMPTY flags. In addition to the minimum pulse widths for the two signals (WRITE CLOCK 60 ns minimum, READ CLOCK 30 ns minimum), it is necessary to ensure that alternating write and read instructions are separated by a time window of at least 50 ns. If the write and read signals originate from two sources that work asynchronously to one another, a synchronizing circuit should be used as shown in Figure . This ensures correct operation even if the two signals appear at the same time.

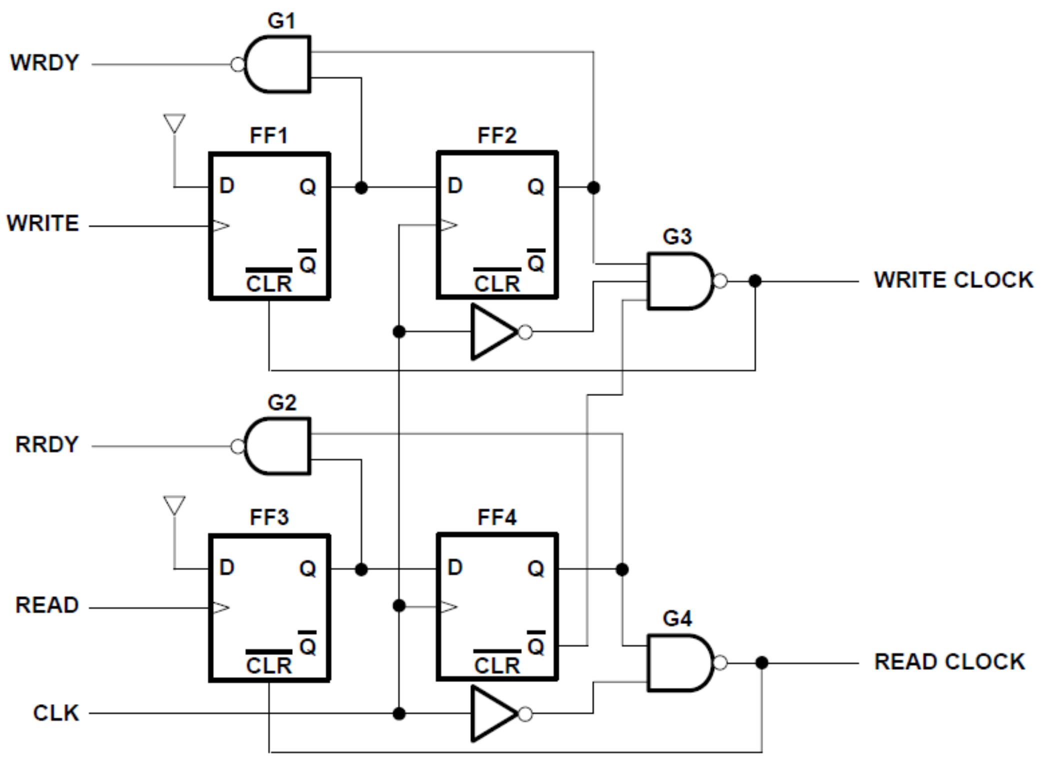


Figure 8: Synchronizing Circuit for Generating WRITE CLOCK and READ CLOCK Signals

3. Connection of Peripherals to Processors: Modern processors are often considerably faster than peripherals that are connected to them. FIFOs can be used so that the processing speed of a processor need not be reduced when it exchanges data with a peripheral. If the peripheral is sometimes faster than the processor, a FIFO can again be used to resolve the problem. Different variations of circuitry are possible, depending on the particular problem. In some cases, the processor reads input data over a unidirectional peripheral, for example, from an A/D converter .A FIFO can buffer a certain amount of input data, then use an interrupt so that the processor reads the data. This interrupt can be triggered by a HALF FULL, ALMOST FULL, or FULL flag.

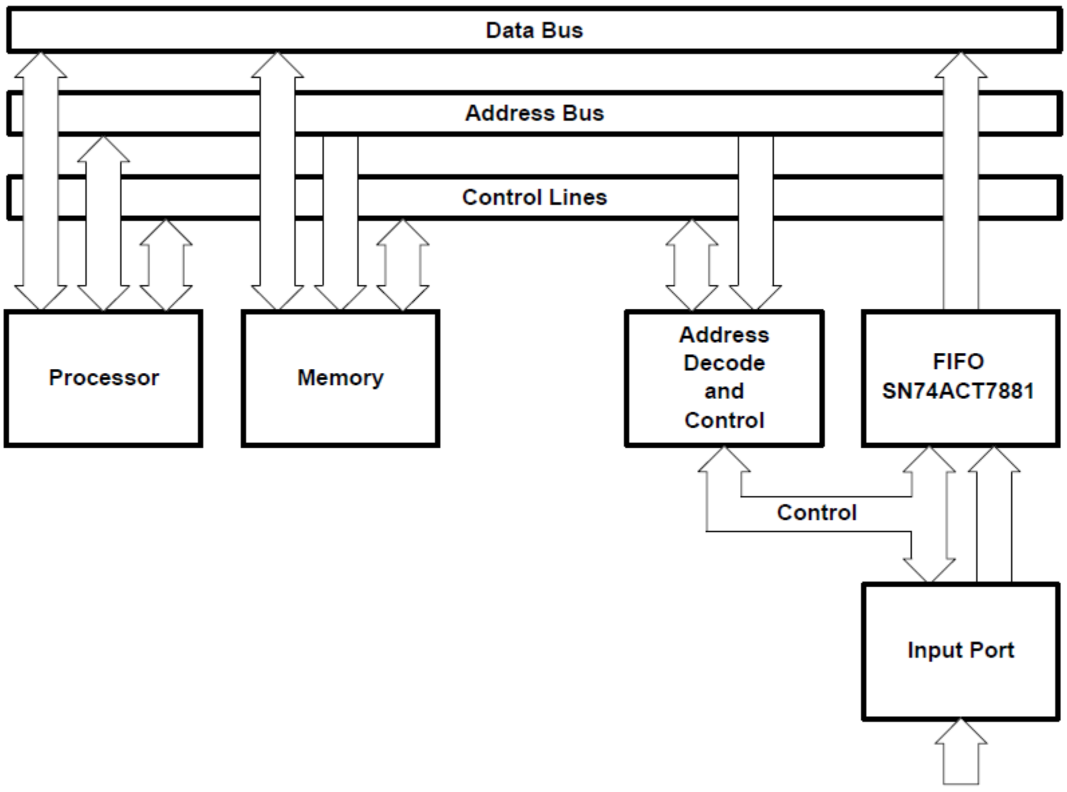


Figure 9:connection to unidirectional peripheral

Often the connected peripherals are bidirectional, like a parallel port, a serial port, a hard-disk controller, or the interface with a magnetic-tape drive. In these cases, there is the possibility of using a bidirectional FIFO like SN74ACT2235. Two independent FIFOs are implemented in this device, each with a word width of 9 bits and memory depth of 1024 words.Figure shows a block diagram for such an application

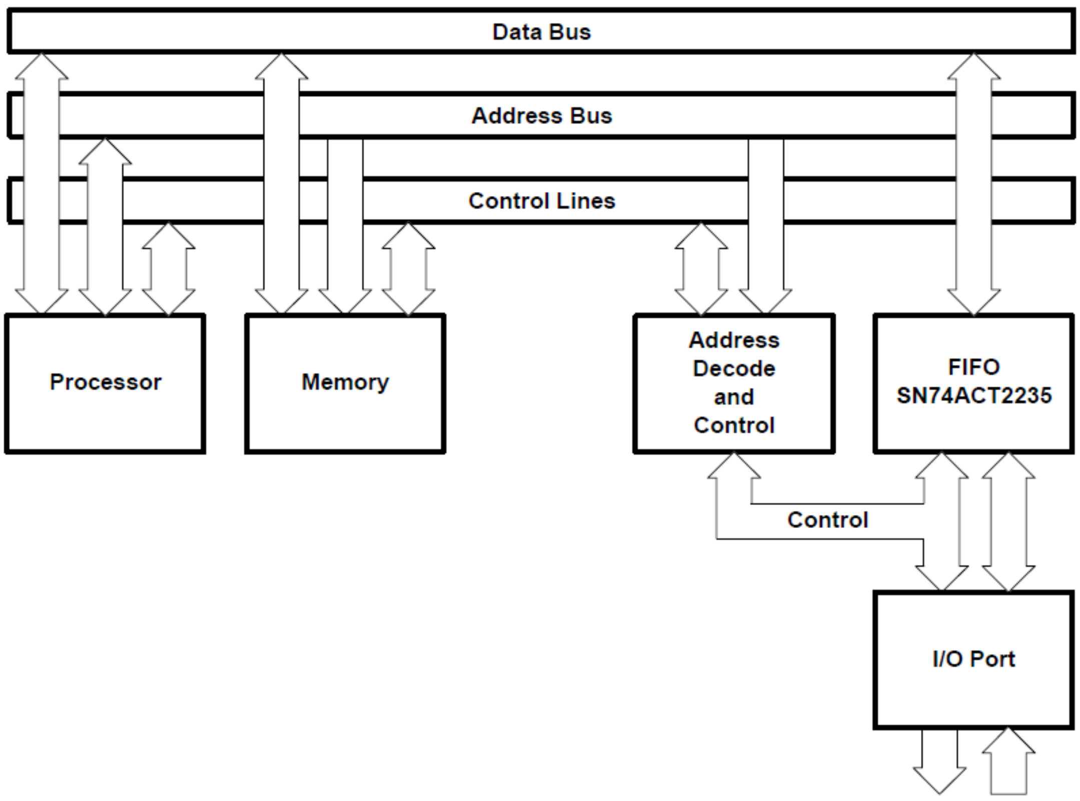


Figure 10: connection to bidirectional peripheral

4. **Block Transfer of Data:** Data are often split into blocks and transmitted on data lines. Computer networks and digital telephone-switching installations are examples of this application. If such a conversion into blocks has to be made at very high speed, it is possible only with appropriate hardware, not through software. A simple solution to this hardware problem is the use of FIFOs (see Figure). A FIFO with a HALF FULL flag should be selected. Furthermore, memory depth should correspond to twice the size of a block. As soon as the HALF FULL flag is set, the send controller starts sending the data block. The send controller consists, in this case, of a counter and some gates and is very easy to implement with just one PAL. The writing of data into the FIFO can be carried on continuously and is independent of the transfer of data blocks.

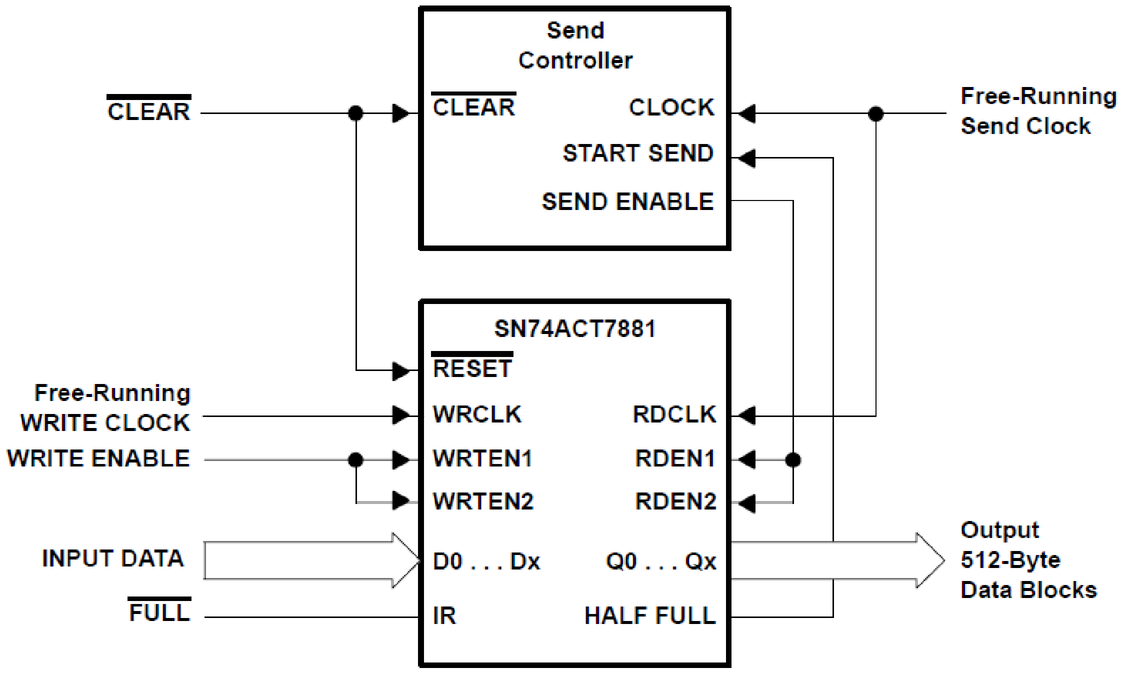


Figure 11:block transfer with synhcronous FIFO

5. **Programmable Delay:**With FIFOs, it is possible to implement a programmable, digital delay line with minimum effort. Because of the programmable AF/AE (ALMOST FULL/ALMOST EMPTY) flag of the SN74ACT7881, only one inverter in addition to the FIFO is Necessary.

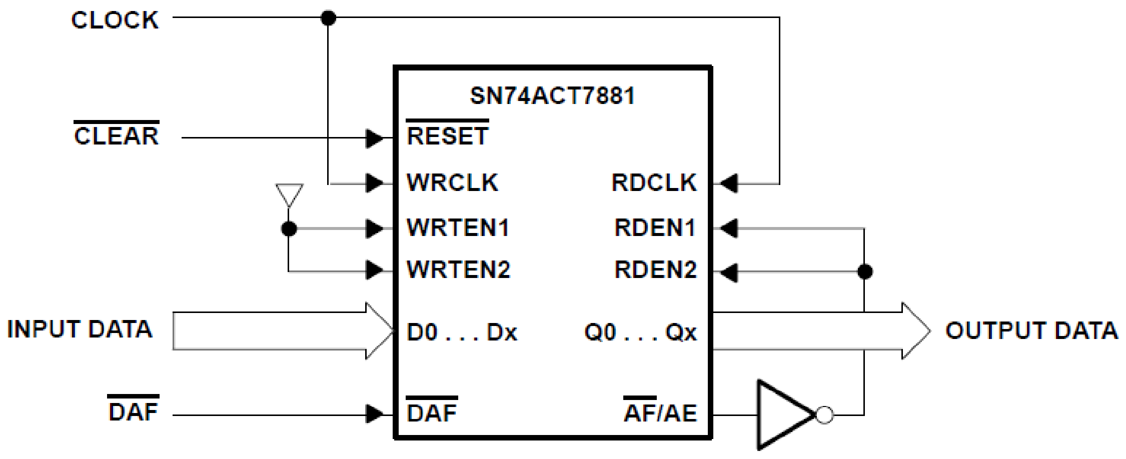


Figure 12:programmable delay with FIFO

6. **Collecting Data Before and After an Event:** By extension of the device in Figure , data words can be collected both before and after the occurrence of an event (see Figure); 1024 data words are always collected. With the aid of the programmable AF/AE flag, it is possible to specify the number of data words that are collected before and after the event. Until the

appearance of the event, the circuit works like that in Figure 33. Here, an RS flip-flop is used to drive the WRTEN input. The TRIGGER WINDOW sets this RS flip-flop and permits data capture. When the event occurs, and the TRIGGER WINDOW signal goes to low level, data are captured until the input-ready (IR) flag is low level and the RS flip-flop is reset. Also, there is an AND gate in the signal path from the AF/AE output to the RDEN1/RDEN2 input. This prevents continued reading out of data after the event. When all the data have been captured, the collected data words can be read out.

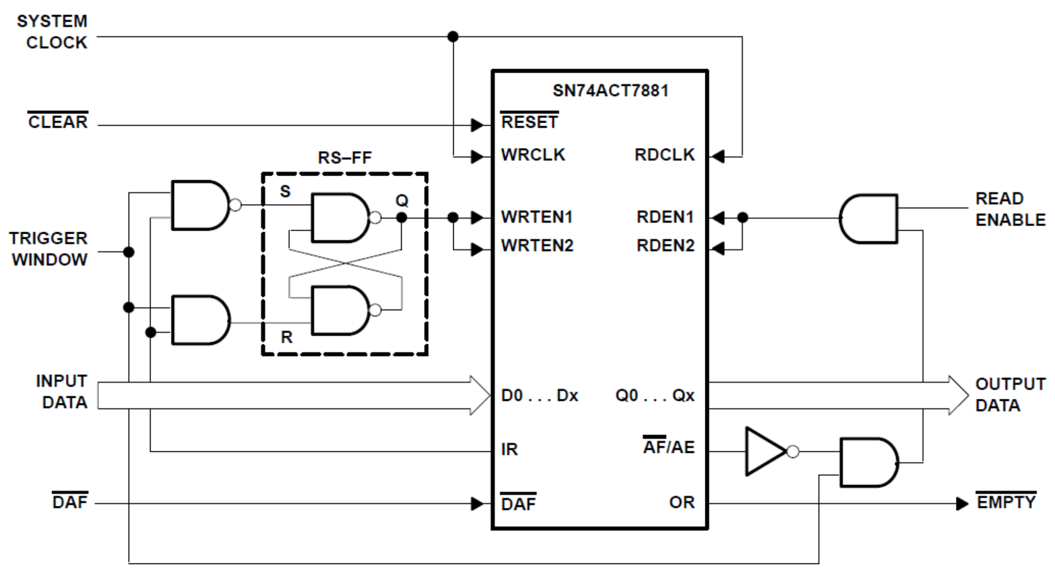


Figure 13:collecting data before and after an event

9.CONCLUSION:

The main objective of the project was to have a working FIFO memory block that functions correctly as per it’s specification & intent, the same was achieved by designing and simulating the FIFO design using CADENCE TOOLS.This design was verified for it’s correct functionality by writing testbench & analyzing the actual response against expected one from simulation waveforms.

10.REFERENCES:

1.E. Brunvand, \Low latency self-timed ow-through _fos," in Advanced Research in VLSI, Mar.1995, pp. 76.

2. Sutherland and S. Fairbanks, \GasP: A minimal FIFO control," in Advanced Research in Asynchronous Circuits and Systems, Mar. 2001, pp. 46.

3.J. T. Yantchev, C. G. Huang, M. B. Josephs, and I. M. Nedelchev, \Low-latency asynchronous FIFO bu_ers," in Proc. Asynchronous Design Methodologies, May 1995.

4. Chris J. Myers, Asynchronous Circuit Design, John Wiley & Sons, Inc., 2001.

5. W. J. Dally and J. W. Poulton, Digital Systems Engineering, Cambridge University Press, Cambridge, UK, 1998.

6.J. F. Wakerly, Digital Design: Principles and Practices, Prentice-Hall, third edition, 1999.

7.M. Hurtado and D. L. Elliot, \Ambiguous behavior of bistable elements," in Allerton Conf. on Circuit and System Theory, Oct. 1975, pp. 605.

8.J. M. Rabaey, A. Chandrakasan, and B. Nikolic, Digital Integrated Circuits, A Design Perspec- tive, Prentice Hall, Upper Saddle River, NJ, 2003.

9. R. Ho, K.W. Mai, and M.A. Horowitz, \The future of wires," in Proceedings of the IEEE, Apr.2001, vol. 89, pp. 490{504}.

10. G. Semeraro, G. Magklis, and et al., \Energy-e_cient processor design using multiple clock domains with dynamic voltage and frequency scaling," in International Symposium on High- Performance Computer Architecture, Feb. 2002, pp. 29.

11. D. M. Chapiro, Globally-Asynchronous Locally-Synchronous Systems, Ph.D. thesis, StanfordUniversity, Stanford, CA, USA, 1984.

12.FIFO architecture,functions and applications,texas instruments.