<u>ASYNCHRONOUS FIFO DESIGN</u>

SUMMER TRAINING: FABLESS IC DESIGN-2017

PROJECT REPORT

Submitted in partial fulfilment of the requirements for the award of the degree

of

BACHELOR OF TECHNOLOGY

in

ELECTRICAL & ELECTRONICS ENGINEERING

by

Name of the Student: Mukund Shah
Enrollment No: 05211502815

Guided by

Name of the Mentor
Dr. Manoj Sharma

DEPARTMENT OF ELECTRICAL & ELECTRONICS ENGINEERING
BHARATIVIDYAPEETH'S COLLEGE OF ENGINEERING
(AFFILIATED TO GURU GOBIND SINGH INDRAPRASTHA UNIVERSITY, DELHI)
NEW DELHI – 110053
NOV 2017

CANDIDATE’S DECLARATION

It is hereby certified that the work which is being presented in the B. Tech Major Project Report entitled **"ASYNCHROMOUS FIFO DESIGN"** in partial fulfilment of the requirements for the award of the degree of **Bachelor of Technology** and submitted in the **Department of Electrical & Electronics Engineering** of **Bharati Vidyapeeth Engineering College, New Delhi (Affiliated to Guru Gobind Singh Indraprastha University, Delhi)** is an authentic record of our own work carried out during a period from **12/07/2017 to 18/07/ 2017** under the guidance of **Dr.Manoj Sharma, Assistant Professor.**

The matter presented in the B. Tech Major Project Report has not been submitted by me for the award of any other degree of this or any other Institute.

(Mukund Shah)
(En. No: 05211502815)

The B. Tech Summer Training: Fabless IC Design-2017 Project Viva-Voce Examination of **Mukund Shah (Enrollment No: 05211502815),** has been held on 0**9/11/2017**

(Signature of External Examiner) **Project Coordinator**

ABSTRACT

An interesting technique for doing FIFO design is to perform asynchronous comparisons between the FIFO write and read pointers that are generated in clock domains that are asynchronous to each other. The asynchronous FIFO pointer comparison technique uses fewer synchronization flip-flops to build the FIFO. The asynchronous FIFO comparison method requires additional techniques to correctly synthesize and analyze the design, which are detailed in this paper. To increase the speed of the FIFO, this design uses combined binary/Gray counters that take advantage of the builtin binary ripple carry logic. The fully coded, synthesized and analyzed RTL Verilog model is included[1].

An asynchronous FIFO refers to a FIFO design where data values are written to a FIFO buffer from one clock domain and the data values are read from the same FIFO buffer from another clock domain, where the two clock domains are asynchronous to each other. Asynchronous FIFOs are used to safely pass data from one clock domain to another clock domain.

Data which is to be written and the address where it has to be written is supplied at the input port write data and write address. At the positive edge of the clock when Write enable is enabled so now the data is been written into the FIFO memory, now it has to be Read out, for that to happen Read enable should be Enabled and the address from which the data has to be read should be specified at the input port Read address. This is the Memory operation in brief .now we have to control the memory in such a way that it meets the requirements of the FIFO.

Various fifo design’s simulation and synthesis reports were compared The gray code style pointer design is the best although it is slower than the other 2 binary style ones because it is the one which gives the most stable results as the problem of ripple carry is not there in gray code hence less chance is there for metastability to arise.[1]

ACKNOWLEDGEMENT

We express our deep gratitude to **Dr.Manoj Sharma**, Assistant Proffessor, Department of Electrical & Electronics Engineering for his valuable guidance and suggestion throughout my project work. We are thankful to **Dr.Manoj Sharma ,** Project Coordinators for their valuable guidance.

We would like to extend my sincere thanks to **Head of the Department, Ms.Anuradha Basu** for her time to time suggestions to complete my project work.I am also thankful to **Dr.Dharmendra Saini, Director** for providing me the facilities to carry out my project work.

Note: Students may thank to the persons whom they would like to acknowledge.

Sign
(Mukund Shah)
(En. No: 05211502815)

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

LIST OF ABBREVIATION

b.s:binary style

g.s:graycode style

async:asynchronous

CHAPTER 1: INTRODUCTION

1.0 INTRODUCTION

An asynchronous FIFO refers to a FIFO design where data values are written to a FIFO buffer from one clock domain and the data values are read from the same FIFO buffer from another clock domain, where the two clock domains are asynchronous to each other. Asynchronous FIFOs are used to safely pass data from one clock domain to another clock domain. There are many ways to do asynchronous FIFO design, including many wrong ways. Most incorrectly implemented FIFO designs still function properly 90% of the time. Most almost-correct FIFO designs function properly 99%+ of the time[2].

1.1 Passing multiple asynchronous signals

Attempting to synchronize multiple changing signals from one clock domain into a new clock domain and insuring that all changing signals are synchronized to the same clock cycle in the new clock domain has been shown to be problematic. FIFOs are used in designs to safely pass multi-bit data words from one clock domain to another. Data words are placed into a FIFO buffer memory array by control signals in one clock domain, and the data words are removed from another port of the same FIFO buffer memory array by control signals from a second clock domain[2].

1.2 Synchronous FIFO pointers

For synchronous FIFO design (a FIFO where writes to, and reads from the FIFO buffer are conducted in the same clock domain), one implementation counts the number of writes to, and reads from the FIFO buffer to increment (on FIFO write but no read), decrement (on FIFO read but no write) or hold (no writes and reads, or simultaneous write and read operation) the current fill value of the FIFO buffer. The FIFO is full when the FIFO counter reaches a predetermined full value and the FIFO is empty when the FIFO counter is zero

1.3 Asynchronous FIFO pointers

In order to understand FIFO design, one needs to understand how the FIFO pointers work. The write pointer always points to the next word to be written; therefore, on reset, both pointers are set to zero, which also happens to be the next FIFO word location to be written. On a FIFO-write operation, the memory location that is pointed to by the write pointer is written, and then the write pointer is incremented to point to the next location to be written. Similarly, the read pointer always points to the current FIFO word to be read. Again on reset, both pointers are reset to zero, the FIFO is empty and the read pointer is pointing to invalid data (because the FIFO is empty and the empty flag is asserted). As soon as the first data word is written to the FIFO, the write pointer increments, the empty flag is cleared, and the read pointer that is still addressing the contents of the first FIFO memory word, immediately drives that first valid word onto the FIFO data output port, to be read by the receiver logic. The fact that the read pointer is always pointing to the next FIFO word to be read means that the receiver logic does not have to use two clock periods to read the data word. If the receiver first had to increment the read pointer before reading a FIFO data..

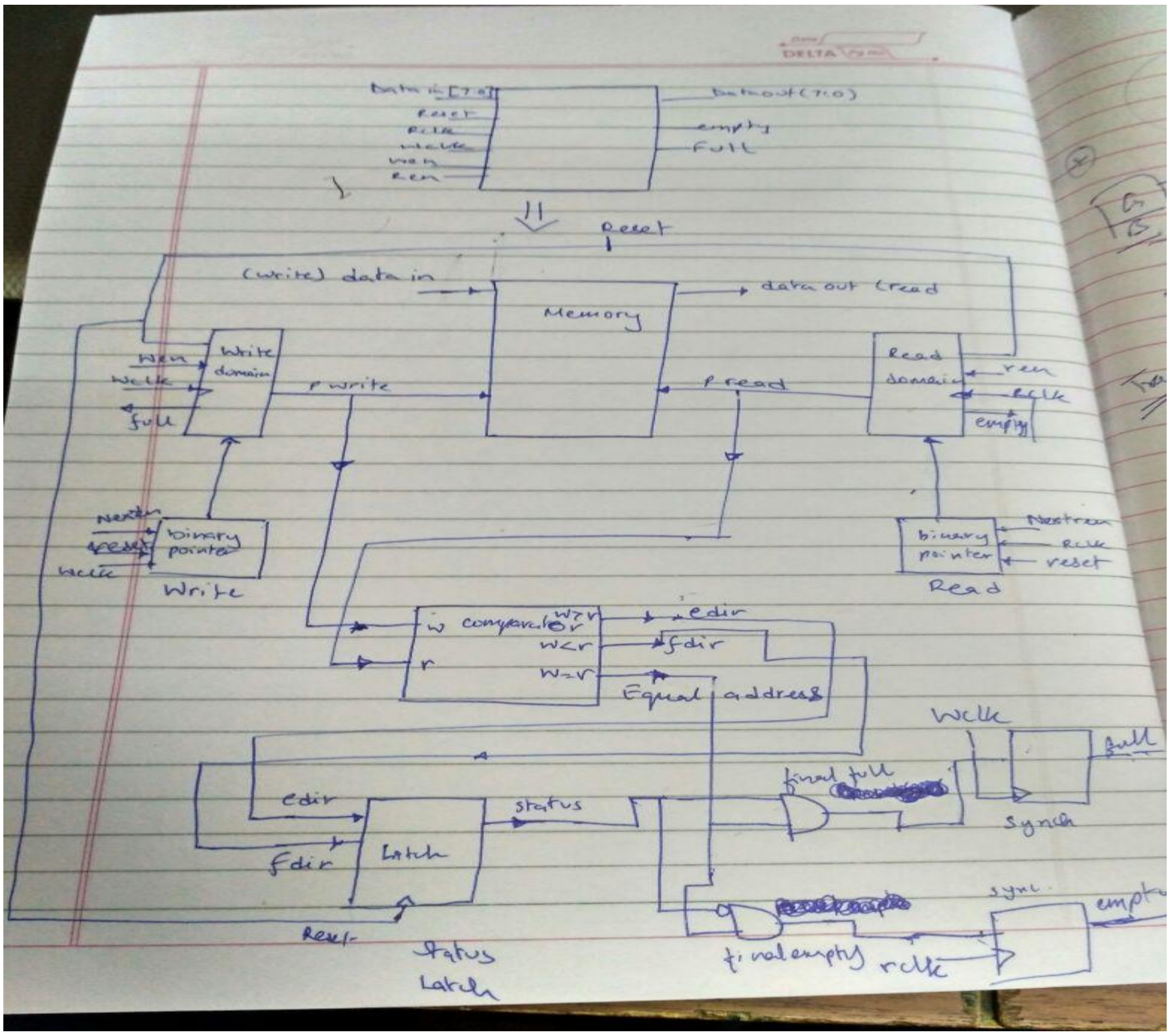


Fig 1.0 Asynchronous FIFO Design

1.4 DESCRIPTION OF FIFO DESIGNED

The above figure's refers of an Asynchronous FIFO, it will be better if each block is explained

FIFO MEMROY

This is the heart of the FIFO, the depth of memory is 8 bits and width is 8 bits,
It has an the following **inputs**
Write Data (8 bit), Write Enable, Read Enable, Write Clock, Write address (4 bit), Read Address (4 bit) And an **output** i.e. Read Data (8 bit)

2.0 MOTIVATION

There are various methodologies to design a asynchronous fifo

1)using **Binary pointers**

2)using **GrayCode pointers**

2.1 Binary FIFO pointer

considerations Trying to synchronize a binary count value from one clock domain to another is problematic because every bit of an n-bit counter can change simultaneously (example 7->8 in binary numbers is 0111->1000, all bits changed). One approach to the problem is sample and hold periodic binary count values in a holding register and pass a synchronized ready signal to the new clock domain. When the ready signal is recognized, the receiving clock domain sends back a synchronized acknowledge signal to the sending clock domain. A sampled pointer must not change until an acknowledge signal is received from the receiving clock domain. A count-value with multiple changing bits can be safely transferred to a new clock domain using this technique. Upon receipt of an acknowledge signal, the sending clock domain has permission to clear the ready signal and re-sample the binary count value. Using this technique, the binary counter values are sampled periodically and not all of the binary counter values can be passed to a new clock domain. FIFO full occurs when the write pointer catches up to the synchronized and sampled read pointer. The synchronized and sampled read pointer might not reflect the current value of the actual read pointer but the write pointer will not try to count beyond the synchronized read pointer value. FIFO empty occurs when the read pointer catches up to the synchronized and sampled write pointer. The synchronized and sampled write pointer might not reflect the current value of the actual write pointer but the read pointer will not try to count beyond the synchronized write pointer value.[2].

2.4 Gray code fifo pointer

The first fact to remember about a Gray code is that the code distance between any two adjacent words is just 1 (only one bit can change from one Gray count to the next). The second fact to remember about a Gray code counter is that most useful Gray code counters must have power-of-2 counts in the sequence. It is possible to make a Gray code counter that counts an even number of sequences but conversions to and from these sequences are generally not as simple to do as the standard Gray code. Also note that there are no odd-count-length Gray code sequences so one cannot make a 23-deep Gray code. This means that the technique described in this paper is used to make a FIFO that is 2n deep.[2]

after the discussion of synchronized Gray code pointers. A common approach to FIFO counter-pointers, is to use Gray code counters. Gray codes only allow one bit to change for each clock transition, eliminating the problem associated with trying to synchronize multiple changing signals on the same clock edge.

3.0 OBJECTIVE

- **The objective of project is to create a synthesizable asynchronous fifo.**
- **Different techniques were used to create a asynchronous fifo**
- **First was to use binary pointer style 1(comparator logic) asynchronous fifo**
- **Second was to create a graycode style (comparator logic) asynchronous fifo**
- **Used to compare both the designs.**

4.0 SUMMARY OF THE REPORT

INTRODUCTION

An asynchronous FIFO refers to a FIFO design where data values are written to a FIFO buffer from one clock domain and the data values are read from the same FIFO buffer from another clock domain, where the two clock domains are asynchronous to each other. Asynchronous FIFOs are used to safely pass data from one clock domain to another clock domain[1].

DESCRIPTION

Data which is to be written and the address where it has to be written is supplied at the input port write data and write address. At the positive edge of the clock when Write enable is enabled so now the data is been written into the FIFO memory, now it has to be Read out, for that to happen Read enable should be Enabled and the address from which the data has to be read should be specified at the input port Read address. This is the Memory operation in brief .now we have to control the memory in such a way that it meets the requirements of the FIFO

RESULT & DISCUSSION

Various fifo design's simulation and synthesis reports were compared

CONCLUSION

The gray code style pointer design is the best although it is slower than the other 2 binary style ones because it is the one which gives the most stable results as the problem of ripple carry is not there in gray code hence less chance is there for metastability to arise.

Chapter 2: <u>Description</u>

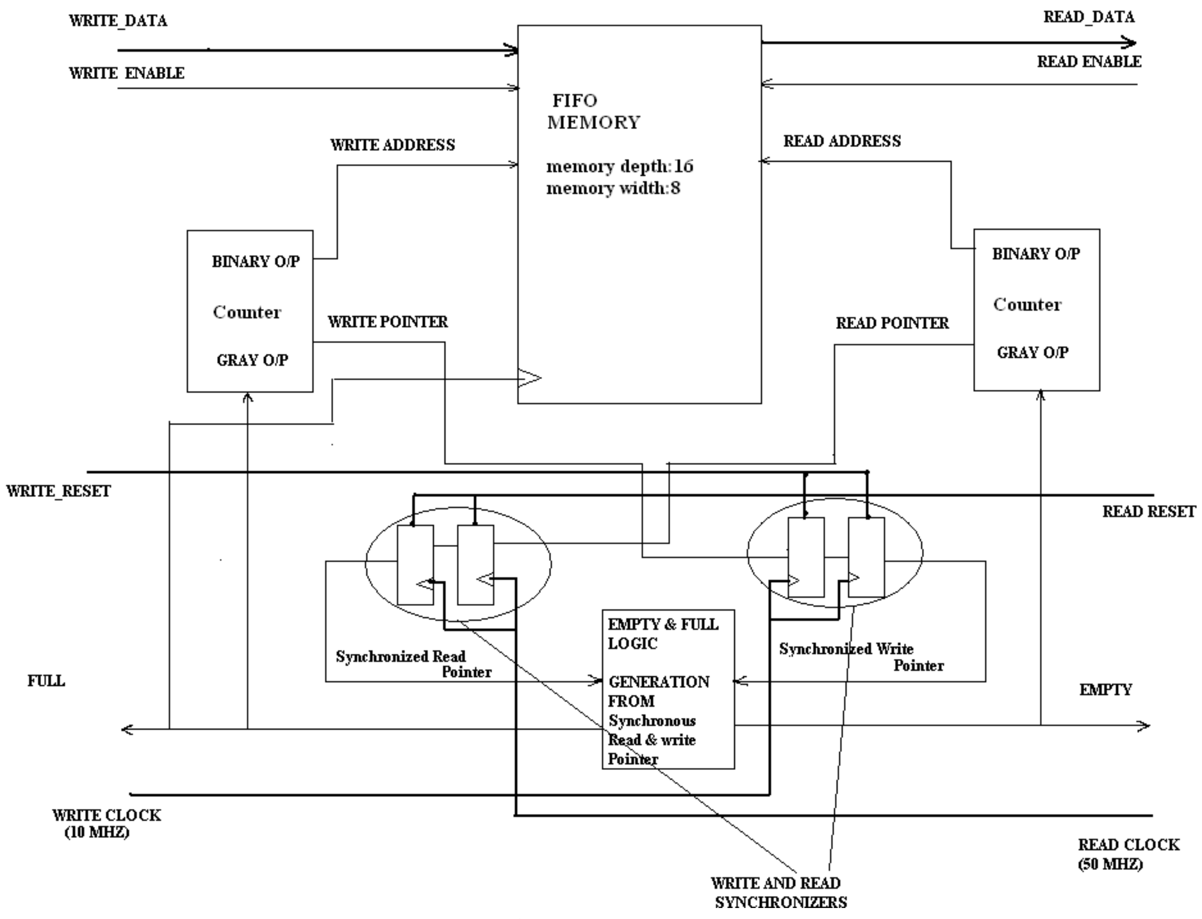


Fig 2.0 Asynchronous FIFO Design

The above figure's refers of an Asynchronous FIFO, it will be better if each block is explained

FIFO MEMROY

This is the heart of the FIFO, the depth of memory is 16 bits and width is 8 bits,
It has an the following **inputs**
Write Data (8 bit), Write Enable, Read Enable, Write Clock, Write address (4 bit), Read Address (4 bit) And an **output** i.e. Read Data (8 bit)
Data which is to be written and the address where it has to be written is supplied at the input port write data and write address. At the positive edge of the clock when Write enable is

enabled so now the data is been written into the FIFO memory, now it has to be Read out, for that to happen Read enable should be Enabled and the address from which the data has to be read should be specified at the input port Read address.
This is the Memory operation in brief .now we have to control the memory in such a way that it meets the requirements of the FIFO.

BINARY & GRAY COUNTER

We need to design a counter which can give Binary and Gray output's, the need for Binary counter is to address the FIFO MEMORY i.e. Write and Read address. And the need of Gray counter is for addressing Read and Write pointers.
Once the counter with binary and Gray code output is designed it is then Port mapped with Memory's Read address, write address, Read pointer, Write Pointer.
The Use Full and Empty logic for addressing the memory
Empty: the counter takes Empty signal and increments the Read address depending on this.
Full: when ever the Full signal is high the counter should not increment write address

SYNCHRONIZER'S

Synchronizers are very simple in operation; they are made of 2 D Flip Flop's.
As the FIFO is operating at 2 different clock domains so there is a need to synchronize the Write and Read pointers for generating empty and full logic which in turn is used for addressing the FIFO memory.
The Figure below shows how synchronization takes place; the logic behind this is very simple.
What we are trying to do over here is , passing the Write Pointer to a D Flip Flop which is driven by the Read clock and in the same manner the Read pointer is fed to a D Flip Flop which is driven by Write Clock, so as a result of this we get Read Pointer (which is operating under Read clock) and Synchronized Write Pointer which is also operating under Read clock, and the same with Write pointer and Synchronized Read Pointer, so now we can compare them and derive a logic for Generating Empty and Full conditions, which is the most important design part of this FIFO

GRAYCODE STYLE 1

A second Gray code counter style, the one described in this paper, uses two sets of registers, one a binary counter and a second to capture a binary-to-Gray converted value. The intent of this Gray code counter style is to utilize the binary carry structure, simplify the Gray-to-binary conversion; reduce combinational logic, and increase the upper frequency limit of the Gray code counter. The binary counter conditionally increments the binary value, which is passed to both the inputs of the binary counter as the next-binary-count value, and is also passed to the simple binary-to-Gray conversion logic, consisting of one 2-input XOR gate per bit position. The converted binary value is the next Gray-count value and drives the Gray code register inputs[1].

FULL AND EMPTY GENERATION

the one described in this paper, divides the address space into four quadrants and decodes the two MSBs of the two counters to determine whether the FIFO was going full or going empty at the time the two pointers became equal.

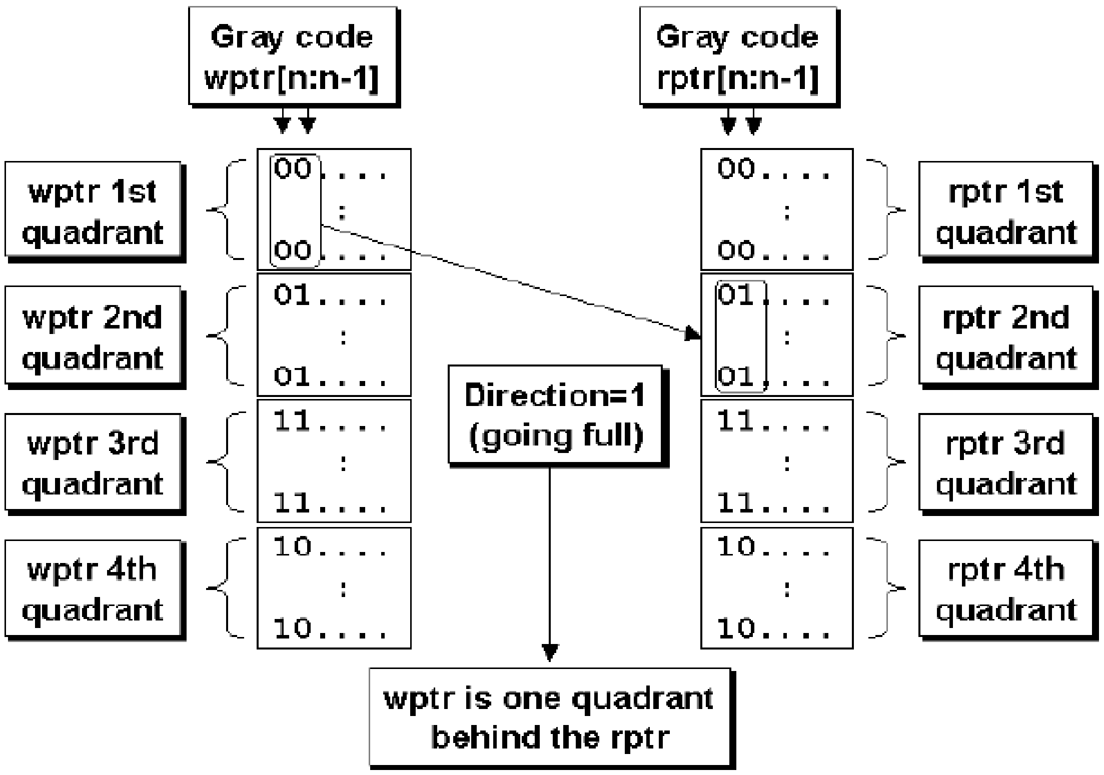


Fig 2.1 - FIFO is going full because the `wptr` trails the `rptr` by one quadrant[1].

If the write pointer is one quadrant behind the read pointer, this indicates a "possibly going full" situation as shown in Figure 1.

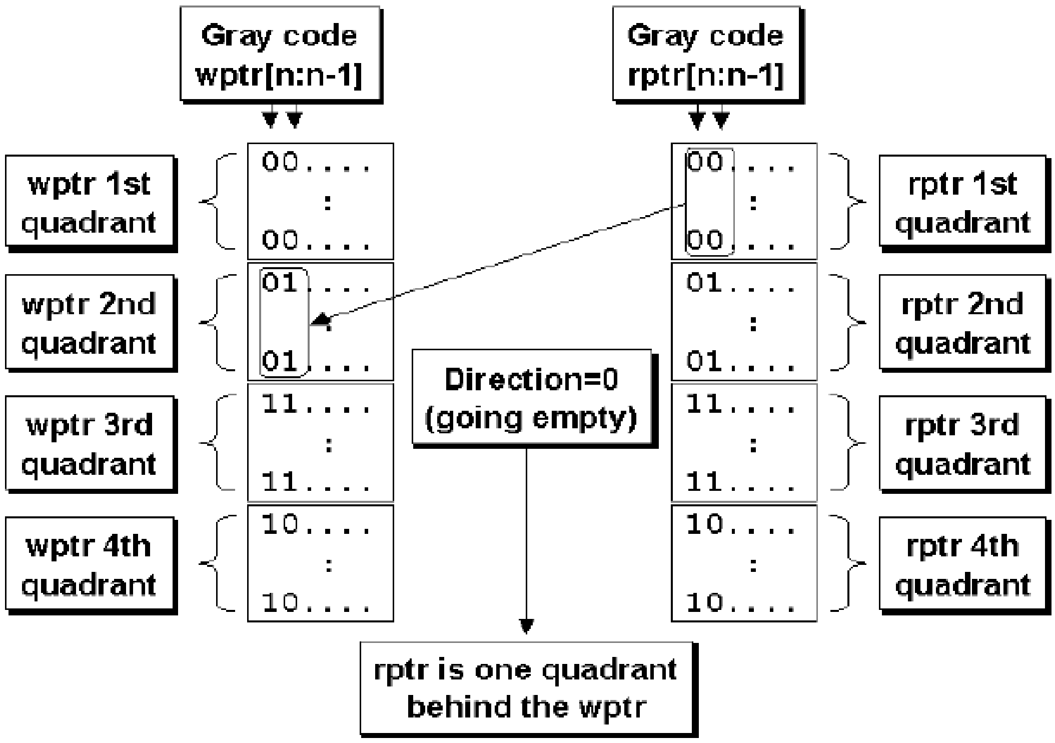


Fig 2.2 - FIFO is going empty because the `rptr` trails the `wptr` by one quadrant.[1]

If the write pointer is one quadrant ahead of the read pointer, this indicates a "possibly going empty" situation as shown in Figure 2.

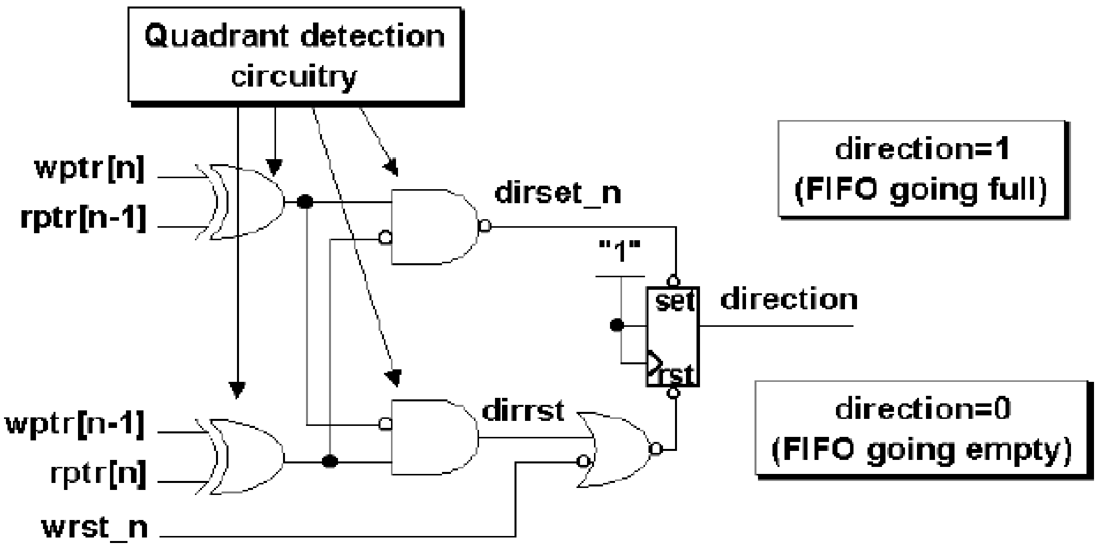

Quadrant detection circuitry
wptr[n]
rptr[n-1]
dirset_n
"1"
set
direction
rst
direction=1 (FIFO going full)
wptr[n-1]
rptr[n]
dirrst
direction=0 (FIFO going empty)
wrst_n

Figure 4 - FIFO direction quadrant detection circuitry

Fig2.3:combinational circuit of quadrant selector logic[1].

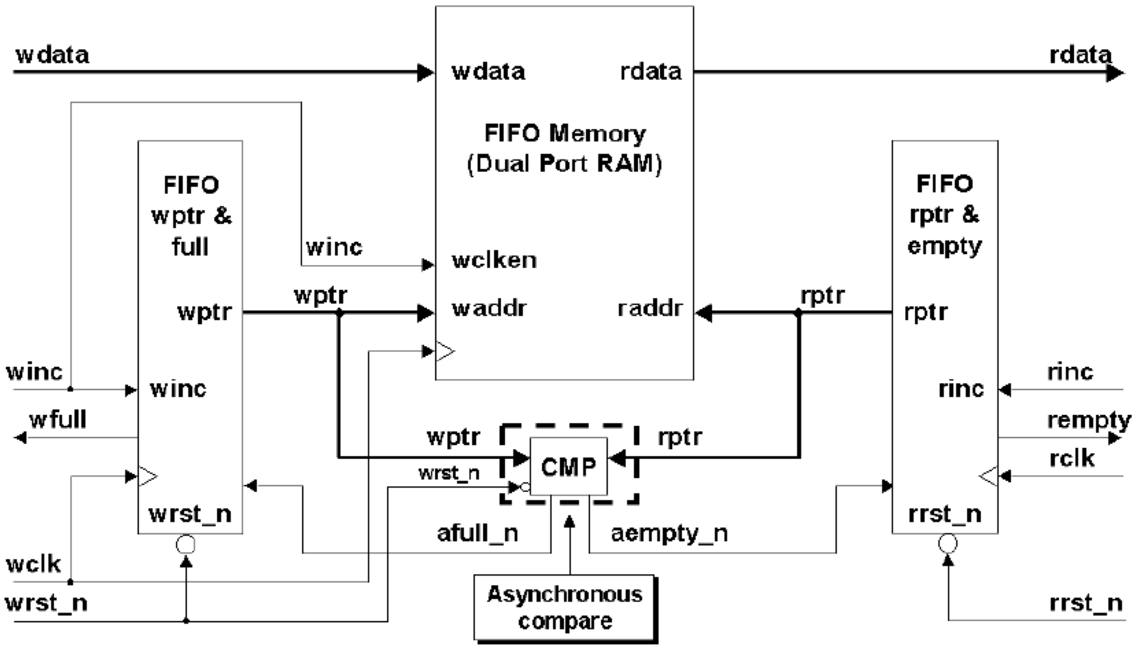


Fig 2.4 :this is the block diag of asynchronous fifo with compare logic[1].

2)<u>BINARY ASYNCHRONOUS FIFO STYLE #1</u>

The logic of this fifo is similar to gray code style ,the comparator logic is used here also.

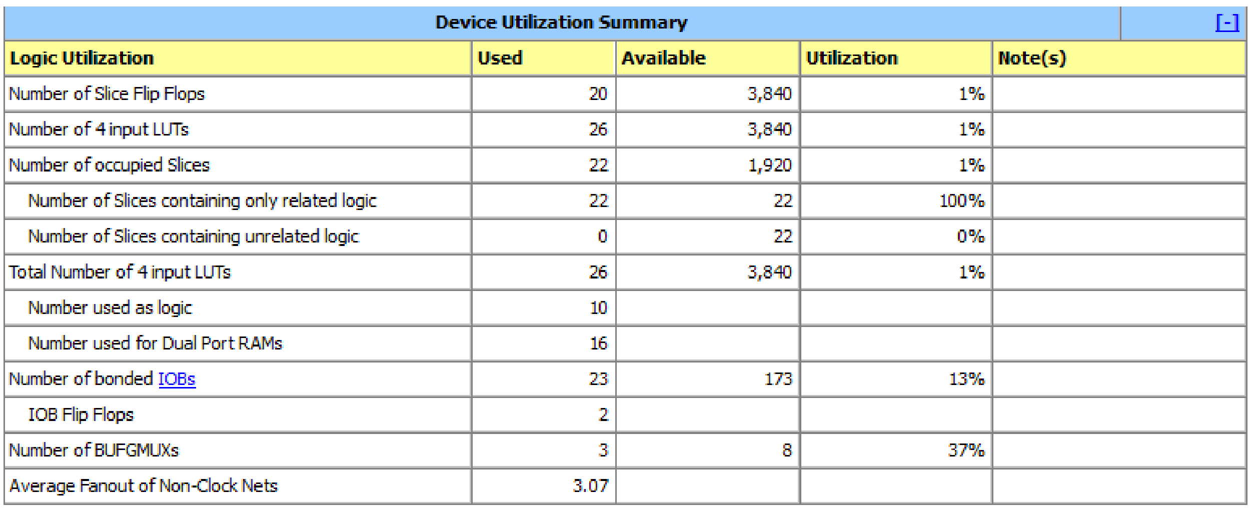
3)BINARY ASYNCHRONOUS FIFO STYLE #2

This style fifo first counts the no. Of enables via separate counters for read and write enables and they are then compared to generate respective full and empty logic.

Chapter 3: Results & Discussion

Binary style 2 asynchronous fifo

Hardware used report



Device Utilization Summary				[-]
Logic Utilization	**Used**	**Available**	**Utilization**	**Note(s)**
Number of Slice Flip Flops	20	3,840	1%	
Number of 4 input LUTs	26	3,840	1%	
Number of occupied Slices	22	1,920	1%	
Number of Slices containing only related logic	22	22	100%	
Number of Slices containing unrelated logic	0	22	0%	
Total Number of 4 input LUTs	26	3,840	1%	
Number used as logic	10			
Number used for Dual Port RAMs	16			
Number of bonded IOBs	23	173	13%	
IOB Flip Flops	2			
Number of BUFGMUXs	3	8	37%	
Average Fanout of Non-Clock Nets	3.07			

Table3.0:hardware report of b.s 2 design

Timing Summary:

Speed Grade: -4

Minimum period: 4.099ns (Maximum Frequency: 243.935MHz)

Minimum input arrival time before clock: 4.334ns

Maximum output required time after clock: 7.165ns

===

Timing constraint: Default period analysis for Clock 'rclk'

Clock period: 4.013ns (frequency: 249.190MHz)

Total number of paths / destination ports: 41 / 22

Timing constraint: Default period analysis for Clock 'wclk'

Clock period: 4.099ns (frequency: 243.935MHz)

Total number of paths / destination ports: 41 / 38

Simulation result

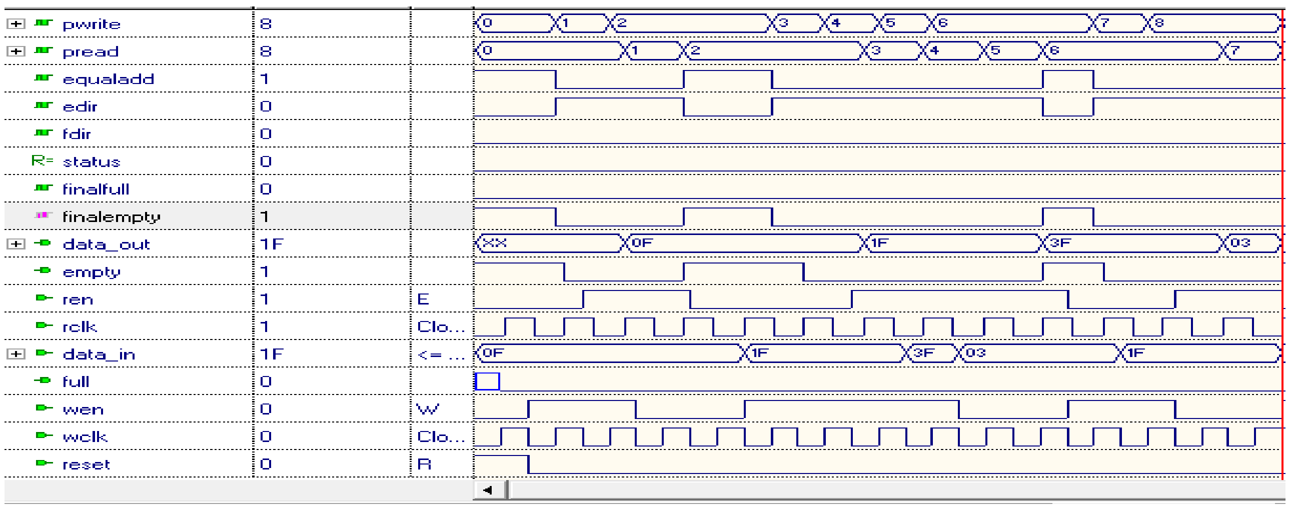
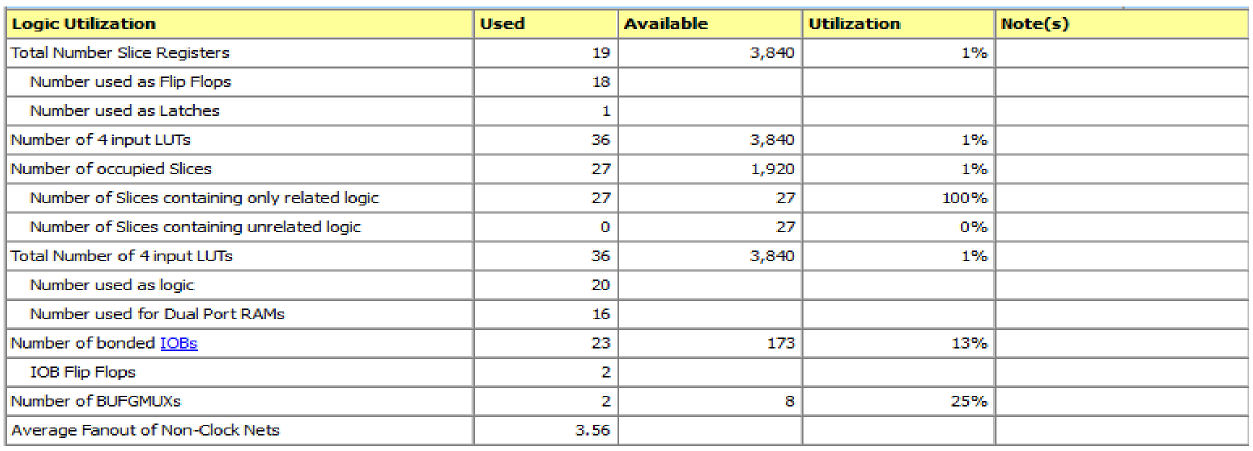


Fig3.0:simulation result of b.s 2 async fifo

Binary style 1 asynchronous fifo

Hardware used report



Logic Utilization	Used	Available	Utilization	Note(s)
Total Number Slice Registers	19	3,840	1%	
Number used as Flip Flops	18			
Number used as Latches	1			
Number of 4 input LUTs	36	3,840	1%	
Number of occupied Slices	27	1,920	1%	
Number of Slices containing only related logic	27	27	100%	
Number of Slices containing unrelated logic	0	27	0%	
Total Number of 4 input LUTs	36	3,840	1%	
Number used as logic	20			
Number used for Dual Port RAMs	16			
Number of bonded IOBs	23	173	13%	
IOB Flip Flops	2			
Number of BUFGMUXs	2	8	25%	
Average Fanout of Non-Clock Nets	3.56			

Table3.1:hardware report of b.s 2 async fifo

Timing Summary:

Speed Grade: -4

Minimum period: 4.122ns (Maximum Frequency: 242.579MHz)

Minimum input arrival time before clock:

4.232ns Maximum output required time after clock: 7.165ns

Timing constraint: Default period analysis for Clock 'rclk'

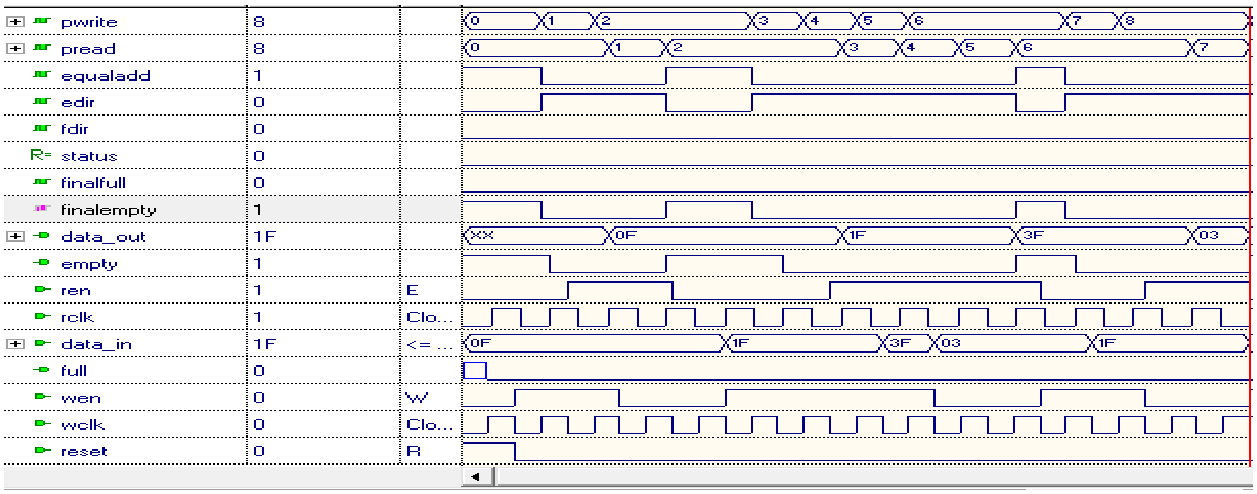
Clock period: 3.987ns (frequency: 250.815MHz)

Timing constraint: Default period analysis for Clock 'wclk'

Clock period: 4.122ns (frequency: 242.579MHz)

Total number of paths / destination ports: 46 / 40

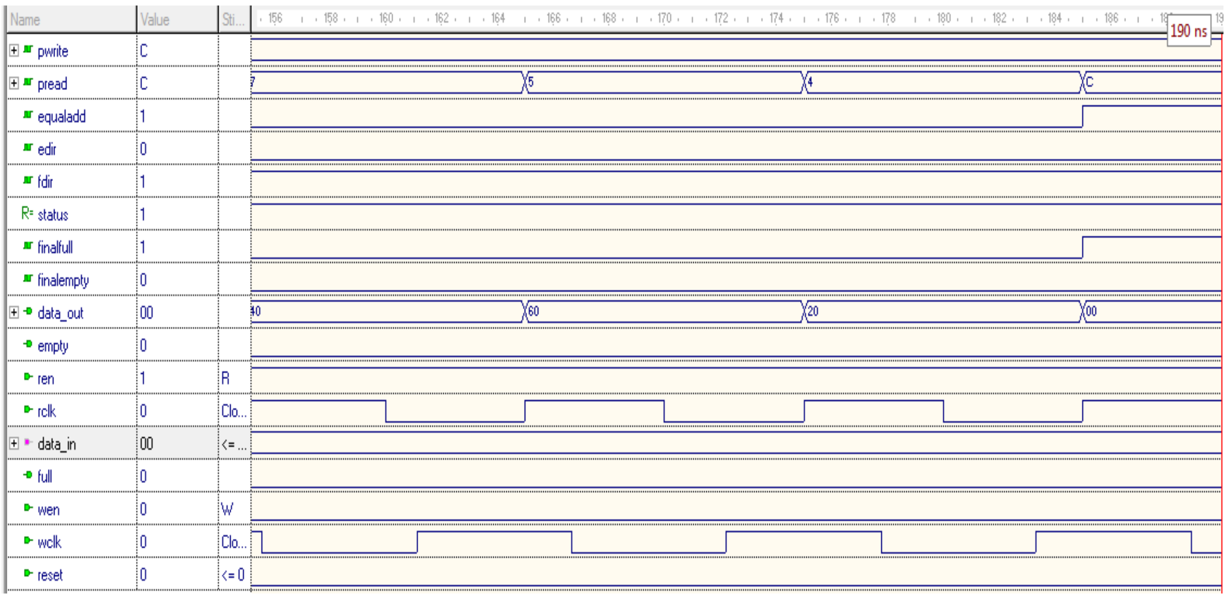
<u>Simulation result</u>



<u>Fig3.1:simulation result of b.s 1 async fifo</u>

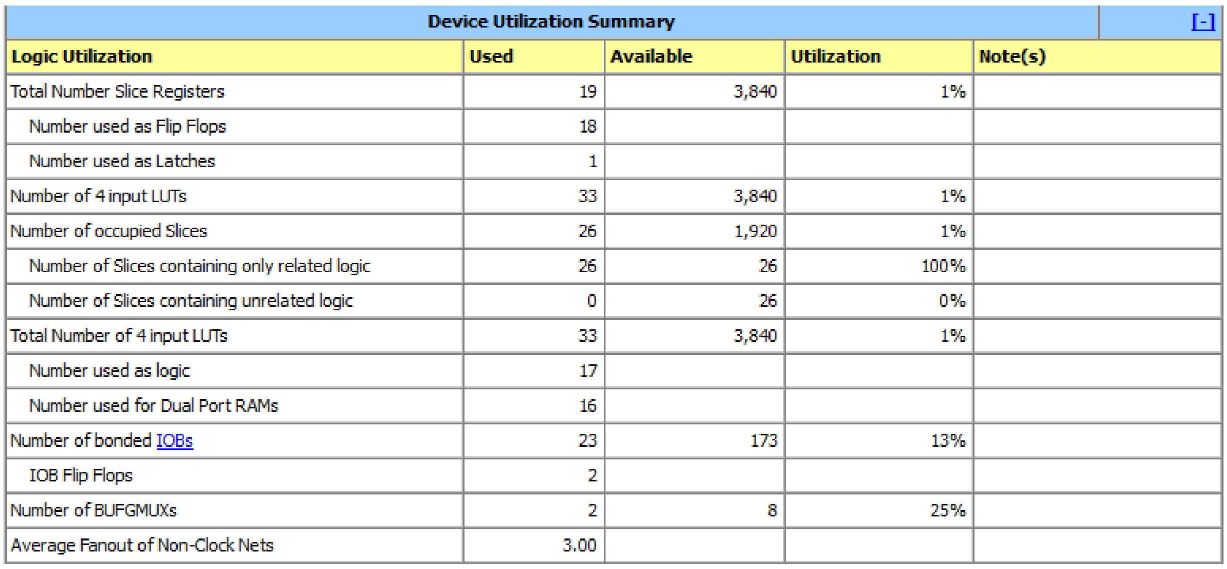
3)<u>Gray code style asynchronous fifo</u>

<u>Simulation result</u>



<u>Fig3.2:simulation result of g.s async fifo</u>

Hardware used report



Device Utilization Summary [-]

Logic Utilization	Used	Available	Utilization	Note(s)
Total Number Slice Registers	19	3,840	1%	
Number used as Flip Flops	18			
Number used as Latches	1			
Number of 4 input LUTs	33	3,840	1%	
Number of occupied Slices	26	1,920	1%	
Number of Slices containing only related logic	26	26	100%	
Number of Slices containing unrelated logic	0	26	0%	
Total Number of 4 input LUTs	33	3,840	1%	
Number used as logic	17			
Number used for Dual Port RAMs	16			
Number of bonded IOBs	23	173	13%	
IOB Flip Flops	2			
Number of BUFGMUXs	2	8	25%	
Average Fanout of Non-Clock Nets	3.00			

Table3.3:hardware report of g.s async fifo

Timing Summary:-

Speed Grade: -4

Minimum period: 5.210ns (Maximum Frequency: 191.939MHz)
Minimum input arrival time before clock: 4.232ns
Maximum output required time after clock: 7.165ns

Timing constraint: Default period analysis for Clock 'rclk'

Clock period: 5.210ns (frequency: 191.939MHz)

Timing constraint: Default period analysis for Clock 'wclk'

Clock period: 4.344ns (frequency: 230.203MHz)

Chapter 4: Conclusion

The gray code style pointer design is the best although it is slower than the other 2 binary style ones because it is the one which gives the most stable results as the problem of ripple carry is not there in gray code hence less chance is there for metastability to arise.

REFERENCES

[1] Simulation and Synthesis Techniques for Asynchronous FIFO Design with Asynchronous Pointer Comparisons by Clifford E. Cummings Peter Alfke Sunburst Design, Inc. Xilinx, Inc.

[2] Simulation and Synthesis Techniques for Asynchronous FIFO Design Clifford E. Cummings, Sunburst Design, Inc

[3] Clock Domain Crossing (CDC) Design & Verification Techniques Using SystemVerilog Clifford E. Cummings Sunburst Design, Inc.

[4] verilog by samir palnitkar

Chapter 5:APPENDIX and CODES

Binary style 1

```
module aFifo


    //Reading port
   (output reg  [7:0]        data_out,
    output reg                     empty,
    input wire                     ren,
    input wire                     rclk,
    //Writing port.
    input wire  [7:0]        data_in,
    output reg                     full,
    input wire                     wen,
    input wire                     wclk,

    input wire                     reset);
   /////Internal connections & variables//////
   reg   [7:0]mem [7:0];
   wire  [3:0]pwrite, pread;
   wire                          equaladd;
  // wire                          nextwen, nextren;
   wire                          edir, fdir;
   reg                          status;
   wire                          finalfull, finalempty;
   initial begin
$readmemh("memory.list", mem);
end

   //'data_out' logic:
   always @ (posedge rclk)
       if (ren && (!empty))
           data_out <= mem[pread];

   //'data_in' logic:
   always @ (posedge wclk)
       if (wen && (!full))
           mem[pwrite] <= data_in;






   // (binary counters) :
         binarycounter write
      (.binarycount(pwrite),

       .en(wen),
       .clear(reset),

       .clk(wclk)
      );

   binarycounter read
      (.binarycount(pread),
       .en(ren),
       .clear(reset),
       .clk(rclk)
      );

   // asynchronous comparator logic:
   assign equaladd = (pwrite == pread);
   assign fdir = (pwrite<pread)?1'b1:1'b0;
   assign edir = (pwrite>pread)?1'b1:1'b0;

         //'status' latch
   always @ (fdir, edir, reset)
       if (edir | reset)
           status = 0;  //going empty.
       else if (fdir)
```

```
          status = 1;  //going full.



    assign finalfull = status & equaladd;  //full condition.


    always @ (posedge wclk, posedge finalfull) //'full logic synchronised with
write clock'.
        if (finalfull&&(pwrite==4'b1000))
            full <= 1;
        else
            full <= 0;



    assign finalempty = ~status & equaladd; //empty condition.


    always @ (posedge rclk, posedge finalempty) //'Empty logic synchronised with
read clock'.
        if (finalempty&&(pread==4'b0000))
            empty <= 1;
        else
            empty <= 0;


endmodule
```

<u>2)Binary style 2</u>

```
module nafifo
 #(parameter fifodepth = 8)
     //Reading port
```

```
   (
          output reg  [7:0]        data_out,
    output reg                          empty,
    input wire                          ren,
    input wire                          rclk,
    //Writing port.
    input wire  [7:0]        data_in,
    output reg                          full,
    input wire                          wen,
    input wire                          wclk,

    input wire                          reset);

   /////Internal connections & variables//////
   reg   [7:0]              mem [7:0];
   wire  [3:0]           pwrite, pread;
   reg [3:0] count_ren,count_wen;
   wire full_w_flag,full_r_flag;
   wire empty_w_flag,empty_r_flag;
         wire nextwen,nextren;
         reg wset,rset;
         initial begin
 $readmemh("memory.list", mem);
end

   //'data_out' logic:
   always @ (posedge rclk)
```

```
    if (ren & !empty)
        data_out <= mem[pread];

//'data_in' logic:
always @ (posedge wclk)
    if (wen & !full)
        mem[pwrite] <= data_in;

 //assigning controlled enables
assign      nextwen=wen&!(full);
assign  nextren=ren&!(empty);
// (binary counters) :

       binarycounter write
   (.binarycount(pwrite),

    .en(nextwen),
    .clear(reset),

    .clk(wclk)
   );

binarycounter read
   (.binarycount(pread),
    .en(nextren),
    .clear(reset),
    .clk(rclk)
```

```
        );

//counting no. of times write operation is done using wen
always @ (posedge wclk)
begin

if(reset)
begin
count_wen <= 0;
end

else if(wen==1'b1)
begin
count_wen <= count_wen+1'b1;
end

end

//counting no. of times read operation is done using ren
always @ (posedge rclk)
begin

if(reset)
begin
count_ren <= 0;
end
```

```
else if(ren==1'b1)
begin
count_ren <= count_ren+1'b1;
end

end

//assigning full read and write flags
assign full_w_flag =(count_wen==fifodepth)?1'b1:1'b0;
assign full_r_flag =(count_ren==fifodepth)?1'b1:1'b0;

//assigning empty read and write flags
assign empty_w_flag =(count_wen==0)?1'b1:1'b0;
assign empty_r_flag =(count_ren==0)?1'b1:1'b0;

//syncronising the 2 combinational values
always @ (posedge wclk,full_r_flag)
        begin
         if(full_r_flag)
                wset <= 1;
         else
                wset <=0;
        end

always @ (posedge rclk,reset,empty_w_flag)
        begin
         if(empty_w_flag)
```

```
                    rset <= 1;
          else
                    rset <=0;
         end

//final full condition
assign finalfull = full_w_flag & wset ;

//final empty condition
assign finalempty = rset & empty_r_flag;

always @ (posedge wclk, posedge finalfull) //'full logic synchronised with write
clock'.
begin

if (finalfull)
full <= 1;
else
full <= 0;

end

always @ (posedge rclk, posedge finalempty) //'empty logic synchronised with
write clock'.
begin

if (finalempty)
empty <= 1;
```

```
else
empty <= 0;

end

endmodule
```

3)Gray code style

```
module gnafifo

    //Reading port
   (output reg  [7:0]       data_out,
    output reg                    empty,
    input wire                    ren,
    input wire                    rclk,
    //Writing port.
    input wire  [7:0]       data_in,
    output reg                    full,
    input wire                    wen,
    input wire                    wclk,

    input wire                    reset);

   /////Internal connections & variables//////
   reg   [7:0]mem [7:0];
   wire  [3:0]pwrite, pread;
   wire                        equaladd;
   wire                        nextwen, nextren;
```

```
    wire                              edir, fdir;
    reg                              status;
          wire                              finalfull, finalempty;

initial begin
 $readmemh("memory.list", mem);
end

    //'data_out' logic:
    always @ (posedge rclk)
        if (ren && (!empty))
            data_out <= mem[pread];

    //'data_in' logic:
    always @ (posedge wclk)
        if (wen && (!full))
            mem[pwrite] <= data_in;




    // (binary counters) :


           graycounter write
       (.graycount(pwrite),


        .en(nextwen),
        .clear(reset),
        .clk(wclk)
       );

    graycounter read
       (.graycount(pread),
        .en(nextren),
        .clear(reset),
        .clk(rclk)
       );
     assign nextwen=wen&!(full);
           assign nextren=ren&!(empty);
    //asynchronous compare logic:
    assign equaladd = (pwrite == pread);
    assign fdir = ~((pwrite[3]^pread[2]) &~(pwrite[2]^pread[3]));
    assign edir = ~((~(pwrite[3]^pread[2]) & (pwrite[2]^pread[3])) | ~reset);

           //'status' latch
    always @ (fdir, edir, reset)
        if (edir | reset)
            status = 0;  //going empty.
        else if (fdir)
            status = 1;  //going full.



    assign finalfull = status & equaladd;  //full condition.
```

```
    always @ (posedge wclk, posedge finalfull) //'full logic synchronised with
write clock'.

        begin

                    if (finalfull&&(pwrite==4'b0100))

            full<=1;




        else

           full<=0;

                    end



    assign finalempty = ~status & equaladd; //empty condition.


    always @ (posedge rclk, posedge finalempty) //'Empty logic synchronised with
read clock'.

        begin

                    if (finalempty&&(pread==4'b0000))

            empty<=1;


        else

            empty<=0;

                    end



endmodule
```