

DSD Project

# Design and Implementation of FIFO Memory

## Submitted by :

## TANUMON ROY (174259)

## HARISH PATIDAR (174213)

## Submitted to :

## Dr. V. Narendar

## Dr. M. Satish

**Date of Submission : 24/04/2019**

Contents

# Introduction

# Architecture

# Flowchart

# Operation

# Applications

# APPENDIX

# Verilog Code

# Results

# Acknowledgement

Introduction

In every item of digital equipment there is exchange of data between printed circuit boards (PCBs). Intermediate storage or buffering always is necessary when data arrive at the receiving PCB at a high rate or in batches, but are processed slowly or irregularly. Buffers of this kind also can be observed in everyday life (for example, a queue of customers at the checkout point in a supermarket or cars backed up at traffic lights). The checkout point in a supermarket works slowly and constantly, while the number of customers coming to it is very irregular. If many customers want to pay at the same time, a queue forms, which works by the principle of first come, first served. The backup at traffic lights is caused by the sporadic arrival of the cars, the traffic lights allowing them to pass through only in batches.

In electronic systems, buffers of this kind also are advisable for interfaces between components that work at different speeds or irregularly. Otherwise, the slowest component determines the operating speed of all other components involved in data transfer. In a compact-disk player, for instance, the speed of rotation of the disk determines the data rate. To make the reproduced sound fluctuations independent of the speed, the data rate of the A/D converter is controlled by a quartz crystal. The different data rates are compensated by buffering. In this way, the sound fluctuations are largely independent of the speed at which disks rotate.

A FIFO is a special type of buffer. The name FIFO stands for first in first out and means that the data written into the buffer first comes out of it first. There are other kinds of buffers like the LIFO (last in first out), often called a stack memory, and the shared memory. The choice of a buffer architecture depends on the application to be solved. FIFOs can be implemented with software or hardware. The choice between a software and a hardware solution depends on the application and the features desired. When requirements change, a software FIFO easily can be adapted to them by modifying its program, while a hardware FIFO may demand a new board layout. Software is more flexible than hardware. The advantage of the hardware FIFOs shows in their speed.

# FIFO Types

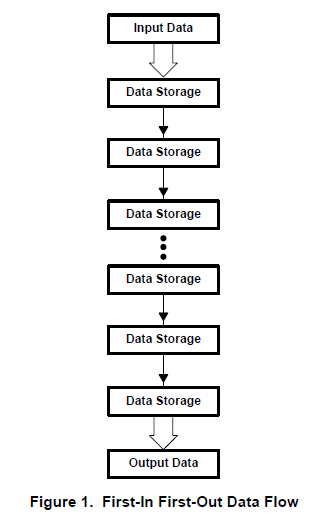
Every memory in which the data word that is written in first also comes out first when the memory is read is a first-in first-out memory. Figure 1 illustrates the data flow in a FIFO. There are three kinds of FIFO:

**Shift register** – FIFO with an invariable number of stored data words and, thus, the necessary synchronism between the read and the write operations because a data word must be read every time one is written

**Exclusive read/write FIFO** – FIFO with a variable number of stored data words and, because of the internal structure, the necessary synchronism between the read and the write operations

**Concurrent read/write FIFO** – FIFO with a variable number of stored data words and possible asynchronism between the read and the write operation

The shift register is not usually referred to as a FIFO, although it is first-in first-out in nature.



Two electronic systems always are connected to the input and output of a FIFO: one that writes and one that reads. If certain timing conditions must be maintained between the writing and the reading systems, we speak of exclusive read/write FIFOs because the two systems must be synchronized. But, if there are no timing restrictions in how the systems are driven, meaning that the writing system and the reading system can work out of synchronism, the FIFO is called concurrent read/write. The first FIFO designs to appear on the market were exclusive read/write because these were easier to implement. Nearly all present FIFOs are concurrent read/write because so many applications call for concurrent read/write versions. Concurrent read/write FIFOs can be used in synchronous systems without any difficulty.

## Exclusive Read/Write FIFOs

In exclusive read/write FIFOs, the writing of data is not independent of how the data are read. There are timing relationships between the write clock and the read clock. For instance, overlapping of the read and the write clocks could be prohibited. To permit use of such FIFOs between two systems that work asynchronously to one another, an external circuit is required for synchronization. But this synchronization circuit usually considerably reduces the data rate.

## Concurrent Read/Write FIFOs

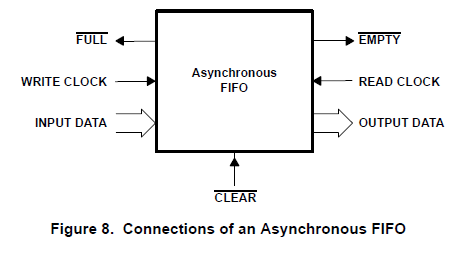
In concurrent read/write FIFOs, there is no dependence between the writing and reading of data. Simultaneous writing and reading are possible in overlapping fashion or successively. This means that two systems with different frequencies can be connected to the FIFO. The designer need not worry about synchronizing the two systems because this is taken care of in the FIFO. Concurrent read/write FIFOs, depending on the control signals for writing and reading, fall into two groups:

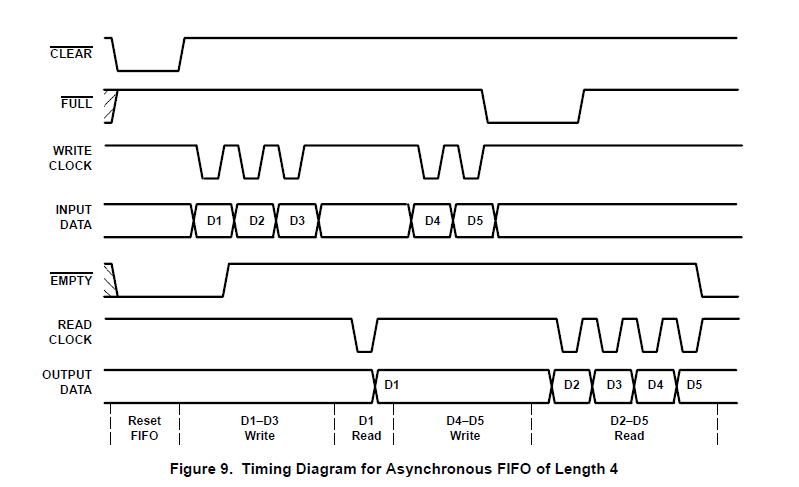
**Synchronous FIFOs**

**Asynchronous FIFOs**

**Asynchronous FIFOs**

The control signals of an asynchronous FIFO correspond most closely to human intuition and were, in the past, the only kind of FIFO driving. The block diagram in Figure 8 shows the control lines of an asynchronous FIFO, and Figure 9 illustrates the typical timing on these lines in a read and write operation.

****



ÎÎ

ÎÎ

ÎÎ

ÏÏ

ÏÏ

The control lines WRITE CLOCK and FULL are used to write data. When a data word is to be written into an asynchronous FIFO, it is first necessary to check whether there is space available in the FIFO. This is done by querying the FULL status line.

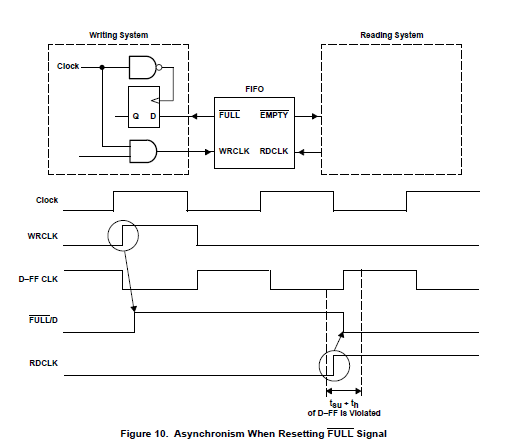
If free space is indicated, the data word is applied to the data inputs and written into the FIFO by a clock edge on the WRITE CLOCK input.

In analogous fashion, the control lines READ CLOCK and EMPTY are used to read data. In this case, the EMPTY status output has to be queried before reading, because data can be read out only if it is stored in the FIFO. Then, a clock edge is applied to the READ CLOCK input, causing the first word in the data queue to appear on the data output.

The timing diagram in Figure 9 shows the resetting of the FIFO that is always necessary at the beginning. Then, three data words are written in. The data words D1 through D3 appear one after the other on the INPUT DATA inputs and clock edges are applied to WRITE CLOCK for transfer of the data. Once the first data word has been written into the FIFO, the EMPTY signal changes from low level to high level. Another two data words are written into the FIFO before the first read cycle. The subsequent reading out of the first data word with the aid of a clock edge on READ CLOCK does not alter the status signals. With the writing of another two data words, the FIFO is full. This is indicated by the FULL signal. Finally, the four data words D2 through D5 remaining in the FIFO are read out. Thus, the FIFO is empty again, so the EMPTY status line shows this by low level.

The disadvantage of a FIFO of this kind is that the status signals cannot be fully synchronized with the read and write clock.

An example of this is shown in Figure 10.

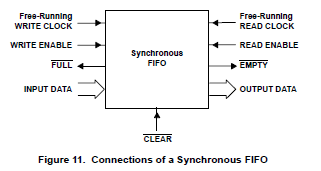


If there is space in the FIFO for only one data word, the next write cycle sets the FULL signal. Then, the writing system queries the FULL signal with the aid of its D flip-flop and waits until there is again space in the FIFO. When a data word is read, READ CLOCK resets the FULL status line. This reset is synchronous with the reading system but asynchronous to the writing system. In the worst case, the setup or hold time of the flip-flop in the writing system is violated. This flip-flop goes into a metastable state, the results of which were discussed previously.

The problem described above also occurs with the EMPTY status signal. EMPTY should be synchronous with the reading system, but it is reset by the writing system. So, the resetting of EMPTY is inevitably asynchronous to the reading system. This asynchronism is a result of the system, and synchronization is not possible within the asynchronous FIFO. If synchronization becomes necessary, the designer must provide it externally. Nevertheless, there are wide-ranging application possibilities for asynchronous FIFOs.

**Synchronous FIFOs**

Synchronous FIFOs are controlled based on methods of control proven in processor systems. Every digital processor system works synchronized with a system-wide clock signal. This system timing continues to run even if no actions are being executed. Enable signals, also often called chip-select signals, start the synchronous execution of write and read operations in the various devices, such as memories and ports. The block diagram in Figure 11 shows all the signal lines of a synchronous FIFO. It requires a free-running clock from the writing system and another from the reading system. Writing is controlled by the WRITE ENABLE input synchronous with WRITE CLOCK. The FULL status line can be synchronized entirely with WRITE CLOCK by the free-running clock. In an analogous manner, data words are read out by a low level on the READ ENABLE input synchronous with READ CLOCK. Here, too, the free-running clock permits 100 percent synchronization of the EMPTY signal with READ CLOCK.

****

Thus, synchronous FIFOs are integrated easily into common processor architectures, offering complete synchronism of the FULL and EMPTY status signals with the particular free-running clock. Figure 12 shows the typical waveform in a synchronous FIFO. WRITE CLOCK and READ CLOCK are free running. The

writing of new data into the FIFO is initialized by a low level on the WRITE ENABLE line. The data are written into the FIFO with the next rising edge of WRITE CLOCK. In analogous fashion, the READ ENABLE line controls the reading out of data synchronous with READ CLOCK.

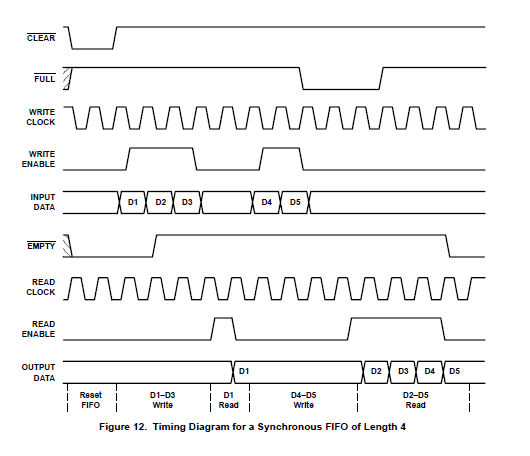
All status lines within the FIFO can be synchronized by the two free-running-clock signals. The FULL line only changes its level synchronously with WRITE CLOCK, even if the change is produced by the reading of a data word. Likewise, the EMPTY signal is synchronized with READ CLOCK. A synchronous FIFO is the only concurrent read/write FIFO in which the status signals are synchronized with the driving logic.ÎÎ

ÎÎ

ÏÏ

ÏÏ

ÏÏ

****

Architecture

All the kinds of FIFOs described in *FIFO Types* can be implemented in different hardware architectures. The architecture of conventional FIFOs has constantly been developed. Initially, FIFOs worked by the fall-through principle. Today, FIFOs are nearly always based on an SRAM, which produced a considerable increase in the number of data words stored, despite the faster speed. All possible hardware architectures also are found in software FIFOs.

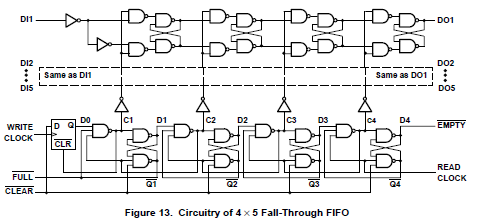
## Fall-Through FIFOs

Fall-through FIFOs were the first FIFO generation. The customers queuing at the checkout point of a supermarket could easily have been the model for this variant. The first customer goes right up to the checkout point, while all others queue behind. Once the first customer has paid and left the front of the queue, the other customers all move up one place

.

**Architecture**

Figure 13 shows one possible design for a fall-through FIFO as implemented, for example, in the SN74S225. In the top half of the illustration, there is a row of latches for storing the data words. The clock generator in the bottom half controls data transfer, the shifting of data, and the data output of the chain of latches. When a new word is written into the FIFO, it “falls through” the entire row of latches and is stored at the last free location. The reading out of a word causes the remaining words to be shifted one position in the direction of the output. A fall-through FIFO, like any other FIFO, has to be reset before it is used so that the clock generator is in a defined initial state.

****

The internal states produced by this clock generator when shifting in a word are shown in Figure 14. The clock pulses are generated entirely by the internal gate propagation delays. In Figure 14, it is possible to see how the clock generator, when writing in a word, produces a clock pulse at C1, C2, C3, and C4, in turn. With these clock pulses, the data word is shifted through the latches to the last one that is free. The clock generator also produces the necessary status information for each data word, specifying whether a data latch already contains a valid data word or is empty. The time required for a data word to be shifted from the input to the output is called the fall-through time.

Advantages and Drawbacks

There must be a status flip-flop for each data word; therefore, the effort involved in controlling the data latches is justifiable only for very short FIFOs. With long FIFOs, there is also an enormous increase in fall-through time. For the 16 × 5 fall-through FIFO SN74S225, a maximum delay of 400 ns is specified from the READ CLOCK to the FULL signal. A 1024 × 18 FIFO, such as SN74ACT7881, requires a multiple of this time with fall-through architecture. The existence of fall-through architecture has a historical basis. New developments no longer use this principle.

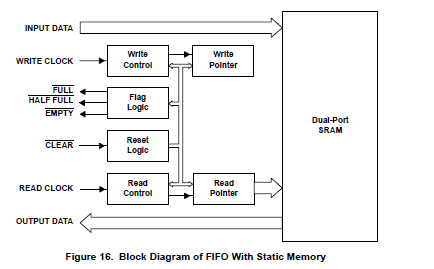
## FIFOs With Static Memory

To counter the disadvantage of a long fall-through time in long FIFOs, the architecture should no longer shift the data words through all memory locations. The problem is solved by a circular memory with two pointers.

In a circular FIFO concept, the memory address of the incoming data is in the write pointer. The address of the first data word in the FIFO that is to be read out is in the read pointer. After reset, both pointers indicate the same memory location. After each write operation, the write pointer is set to the next memory location. The reading of a data word sets the read pointer to the next data word that is to be read out. The read pointer constantly follows the write pointer. When the read pointer reaches the write pointer, the FIFO is empty. If the write pointer catches up with the read pointer, the FIFO is full. Figure 15 illustrates the principle of a circular FIFO with two pointers.

**Architecture**

In the hardware implementation of a circular memory, a dual-port SRAM is used for data storage. The pointers take the form of binary counters, which generate the memory addresses of the SRAM. It is an advantage if the number of memory locations is 2n because a pointer can be implemented as an n-bit binary counter whose carry can be ignored. Figure 16 shows a block diagram of a FIFO with static memory. Read addresses are generated by the READ POINTER and write addresses by the WRITE POINTER. The write-control and read-control blocks control operation during write and read access. The FULL and EMPTY status signals are generated by separate flag logic. Some FIFOs also offer the status signals HALF FULL, ALMOST FULL, and ALMOST EMPTY. A FIFO with static memory also has to be initialized before it is used to set the two pointers and the status logic to defined initial states.



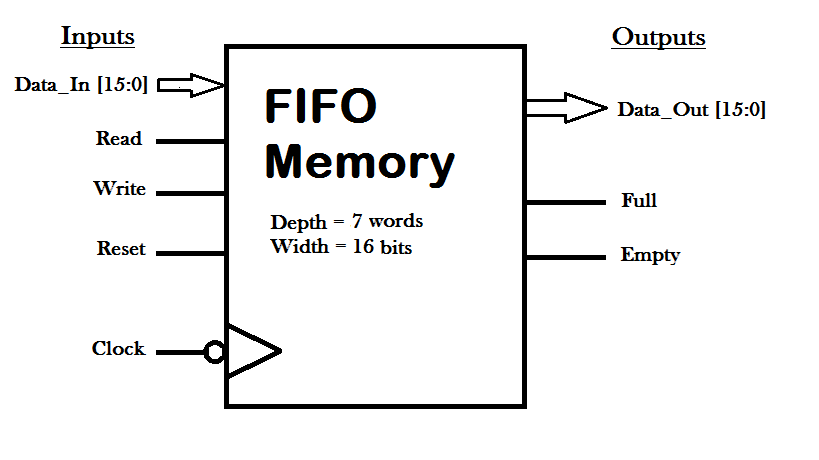
Advantages and Drawbacks

Unlike a fall-through FIFO, the fall-through time of a FIFO with static memory is independent of its length. Therefore, it is possible to create fast FIFOs with a length of several thousand words. The effort that goes into the circuitry of the control logic does not increase to any significant degree with length because only the two pointers and parts of the flag logic have to be expanded. Today’s new developments use only the architecture of the FIFO with static memory.

Flowchart

Operation

The FIFO memory designed in this project is a 16X7 FIFO. That means each input data going into the FIFO has a length of 16 bits and a total number of 7 such data inputs can be stored in the memory after which it will be full.



The basic block diagram of the FIFO memory, as shown, contains 5 input pins and 3 output pins. The first pin on the input side is the “data input” pin. The 16 bit data goes into the memory via this pin. The next input pin is the “read” pin. When this pin gets ON (when it carries a value “1”) that signifies that data should come out of the memory in the FIFO order from the next clock pulse (negative edge) , the same way that it went into the memory. The next input pin is the “write” pin. When this pin is ON, data goes into the memory one by one with every clock pulse (negative edge). Then comes the “reset” pin. When this pin gets ON, the memory gets cleared, initializing all values back to zero, as it was before any data was put inside the memory. The last input pin is the clock signal input pin. This synchronous FIFO uses a shared clock, so this clock controls all the operations of the memory like reading and writing.

Among the 3 pins on the output side of the FIFO memory, the first one is the “data output” pin. This is the pin through which the 16 bit data comes out of the memory, one by one with every clock pulse, during the READ operation. The next output pin is the “full” pin. When this pin is ON, that signifies that the memory is full and there is no more space available for any more incoming data, hence WRITE operation cannot be done. The last among the output pins s another flag pin, known as the “empty” pin. When this pin is ON, it signifies that the FIFO memory is empty and the READ operation cannot be performed since there is no data available to read.

The verilog code used to model the behaviour of the above described FIFO memory is written in Xilinx VIVADO 2018.3 software. Simulation is also done in the same software with the help of two testbenches. The first testbench shows the working of the FIFO when both the read and write signals are ON i.e., there value is “1”. On the other hand, the second testbench shows an instant when the FIFO memory is getting full and hence the “full” flag pin gets ON i.e., it shows a value “1”.

The program starts with the declaration of the input, output and register pins and then the initialization of the concerned values like “DEPTH” and “MAX\_DEPTH”. Then it checks if the “read”, “write” and “empty” pins are simultaneously ON or not. If so, then the variable “sr\_read\_write\_empty” is made “1”. Thus, this variable becomes a flag variable which tells us when all “read”, “write” and “empty” are ON.

Then comes the **READ operation**. At every negative edge of the given clock pulse, the value of the ”rst” (reset) variable is checked. If it’s ON, then the output data is made zero as well as the pointer variable “head”. If “rst” is “0” then it is checked whether the “read” value is “1” and the “empty” is “0”. This shows that the read operation will be performed and the memory is not empty. If the condition holds true, then the output data value is changed to the value stored in the “head” position of the FIFO memory and then the “head” value is incremented by one.

After this is the **WRITE operation**. At every negative edge of given clock pulse, the value of “rst” is checked, similar to that of read operation. If “rst” is “1”, then same steps are followed as before and the “tail” pointer variable is made zero. But, if ”rst” is “0”, then the input data value is stored in the “tail” position of the fifo memory after which the value of “tail” is also incremented by one.

Now comes the **COUNTER operation**.This operation is very crucial to the execution of the code since it tells us about the number of data elements present in the memory at a particular instant. Similar to the previous two operations, at every negative edge clock pulse the “rst” value is checked. If it’s ON, the the “count” value is made to zero. If not, then depending upon the combination of “read” and “write” values, the “count” value is manipulated. If both are zero, then the “count” value remains the same. If only “read” is ON, then the “count” values is decremented by one. If only “write” value is ON, then the “count” value is increased by one. If both are ON, the again it is checked if the memory is empty or not. If it is empty, then no read can be performed, so only write operation gets executed and correspondingly, the value of “count” is incremented by one. If the memory is not empty, then both operations will get executed and hence the number of data elements in the memory will remain the same, hence, no change is done to the “count” variable.

Lastly, it’s the **Memory SIGNAL control operation**. This operation changes the value of the flag variables “full” and “empty” when the corresponding case comes. This is done by checking the “count” value. If it is zero, that tells us that there is no value inside the memory, so then the “empty” is turned ON. When the “count” value is equal to the MAX\_COUNT, that means the memory is full and so the “full” is turned ON.

Since all these operations run on a loop which gets executed whenever a negative edge of the clock pulse arrives, they continue getting executed by the code until the clock pulse is ceased or as in this case, there is the “$finish” statement in the testbech code which is used during simulation purposes.

Applications

The *First in, First out* (i.e. FIFO) data structure has many applications in both computers and electronics. The data structure is designed to support the digital equivalent of waiting in line.

# A CPU, such as the ZipCPU

Interrupting a CPU costs time and performance. It’s not a simple thing to do, in spite of the hard work of many men to make interrupts fast and efficient. Hence, at some data rates, handling requests one at a time can just cripple a CPU–it’s just spending too much time handling the interrupt.

To deal with this, peripheral hardware is often created so that the CPU can mange many items at the same time–spreading the cost of the interrupt across many accesses.

Perhaps a good analogy here might be an airport taxi line. Only one person (family) can get into any taxi at a time, and then all the taxi’s move forward so the next person can get into the next taxi. In this case, when an airplane arrives (the CPU gets busy), the line suddenly swells as many people get off and start waiting in line for a taxi.

# SDRAM Memory

Memory is also another candidate for a FIFO. Unlike the airport taxi line, Memory acts more like a school cafeteria buffet line. Only one class is allowed to use the line at a time, yet there are several stations that each student needs to visit to get his lunch. Hence, in the case of memory, you want to make many transactions at once, fill the pipeline, and then get the whole data through the line as soon as possible so the next class can come through.

In this case, you want a FIFO that can be filled, or nearly filled, and then make all its transactions at once and release the memory so that something else can use it.

(Example: DMA Controller, and a Wishbone-UART debugging bus FIFO)

# A Universal Asynchronous Receiver Transmitter (UART), sometimes known as a serial port

Computers like to load a serial port with data. However, the serial port can typically only handle one item every so many clocks. Often, the serial port can operate much faster than the computer can interrupt, but slower than the computer can issue characters to the port.

In this case, a FIFO can be used to allow the computer to write many characters to the port. These characters will then “wait in line” to be transmitted, and when the line is (nearly) empty, the computer can then send a whole bunch more characters to the port–keeping the port busy at all times.

A similar case for a FIFO can be made on receive as well. The computer just waits for the line to fill up, before it processes all of the elements in the line quickly.

# Audio

Audio is very much like the UART above. The CPU can write at one speed, but the audio may only be able to read at another. A FIFO can allow a CPU the ability to write a lot of things to the audio port at once, after which the audio hardware reads the samples out one by one.

Audio receiving is in many ways like some buses I’ve traveled on. The line for the bus grows and grows. Once it reaches a certain length (in the case of the FIFO), the “bus” comes along and clears the line.

# Video

Unlike UARTs and Audio, the video FIFOs I’ve worked with have rarely (if ever) required the CPU’s attention. They usually work in the background. When receiving video data, once a FIFO buffer fills, the buffer is then dumped into memory. Likewise on the transmit: the video controller reads from memory, fills its buffer, and then slowly sends it to the display one pixel at a time.

Appendix (Verilog Code)

## Main code :

module FIFO\_mem(

input clk,

input rst,

input [15:0] din,

output reg [15:0] dout,

input read,

input write,

output reg empty,

output reg full

);

parameter DEPTH = 3, MAX\_COUNT = 3'b111;

reg [(DEPTH-1):0] tail;

reg [(DEPTH-1):0] head;

reg [(DEPTH-1):0] count;

reg [15:0] fifo [0:MAX\_COUNT];

reg sr\_read\_write\_empty;

always@(negedge clk)

begin

if(rst)

sr\_read\_write\_empty <= 1'b0;

else if(read == 1'b1 && write == 1'b1 && empty == 1'b1)

sr\_read\_write\_empty <= 1'b1;

else

sr\_read\_write\_empty <= 1'b0;

end

//READ Operation

always@(negedge clk)

begin

if(rst)

begin

dout=16'd0;

head <= 3'b000;

end

else if(read == 1'b1 && empty == 1'b0)

begin

dout <= fifo[head];

head <= head + 1'b1;

end

end

//WRITE Operation

always@(negedge clk)

begin

if(rst)

tail <= 3'b000;

else if(write == 1'b1 && full == 1'b0)

begin

fifo[tail] <= din;

tail <= tail + 1'b1;

end

end

//COUNTER Operation

always@(negedge clk)

begin

if(rst)

count <= 3'b000;

else

begin

case({read,write})

2'b00: count <= count;

2'b01: if(count != MAX\_COUNT)

count <= count + 1;

2'b10: if(count != 3'b000)

count <= count - 1;

2'b11: if(sr\_read\_write\_empty)

count <= count + 1;

else

count <= count;

default: count <= count;

endcase

end

end

//Memory SIGNAL control

always@(count)

begin

empty <= 1'b0;

full <= 1'b0;

if(count == 3'b000)

empty <= 1'b1;

else if(count == MAX\_COUNT)

full <= 1'b1;

end

endmodule

## Testbench Code 1 :

module test\_FIFO(

);

//Inputs

reg clk;

reg [15:0] din;

reg read;

reg write;

reg rst;

// Outputs

wire [15:0] dout;

wire empty;

wire full;

// Instantiate

FIFO\_mem memory(clk, rst, din, dout, read, write, empty, full);

// Initialize Inputs

initial

begin

clk = 1'b0;

din = 32'h0;

read = 1'b0;

write = 1'b0;

rst = 1'b1;

#60;

rst = 1'b1;

#20;

rst = 1'b0;

write = 1'b1;

din = 16'h0;

#20 din = 16'h11;

#20 din = 16'h12;

#20 din = 16'h13;

#20 din = 16'h14;

#20 din = 16'h15;

#20 begin din = 16'h16; read = 1'b1; end

#20 din = 16'h17;

#20 din = 16'h18;

#20 write = 1'b0;

#140 rst = 1'b1;

#40 $finish;

end

always

#10 clk = ~clk;

endmodule

## Testbench Code 2 :

module test\_FIFO02(

);

//Inputs

reg clk;

reg [15:0] din;

reg read;

reg write;

reg rst;

// Outputs

wire [15:0] dout;

wire empty;

wire full;

// Instantiate

FIFO\_mem memory(clk, rst, din, dout, read, write, empty, full);

// Initialize Inputs

initial

begin

clk = 1'b0;

din = 32'h0;

read = 1'b0;

write = 1'b0;

rst = 1'b1;

#40;

rst = 1'b1;

#20;

rst = 1'b0;

write = 1'b1;

din = 16'h0;

#20 din = 16'h11;

#20 din = 16'h12;

#20 din = 16'h13;

#20 din = 16'h14;

#20 din = 16'h15;

#20 din = 16'h16;

#20 din = 16'h17;

#20 din = 16'h18;

#20 begin write = 1'b0;read = 1'b1; end

#180 rst = 1'b1;

#60 $finish;

end

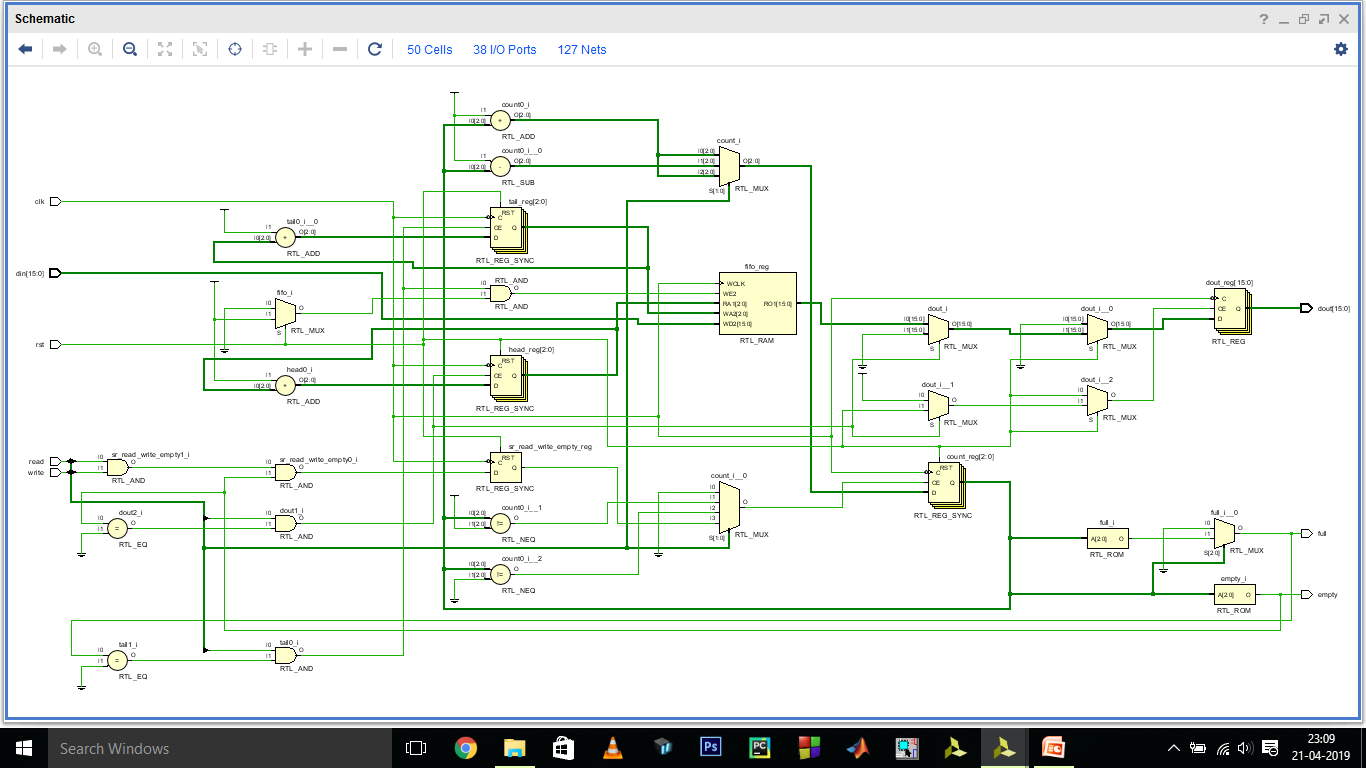
always

#10 clk = ~clk;

endmodule

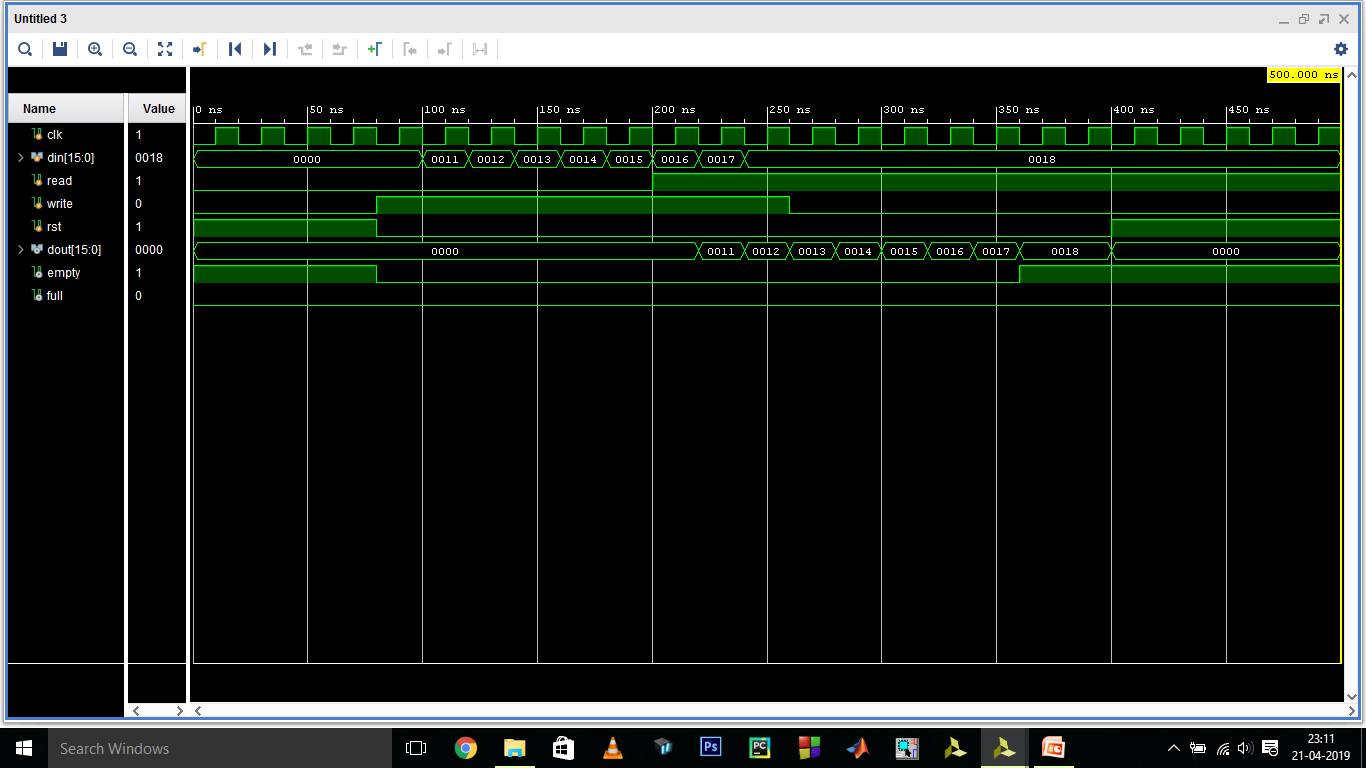
Appendix (Results)

# RTL Schematics :



# Simulation Results :

**For Testbench 1 :**



**For Testbench 2 :**



Acknowledgement

## Sources referred to while doing this project :

* Verilog HDL: A Guide to Digital Design and Synthesis
  + Book by Samir Palnitkar
* <https://surf-vhdl.com/what-is-a-fifo/>
* <http://www.ti.com/lit/an/scaa042a/scaa042a.pdf>
* <https://en.wikipedia.org/wiki/FIFO_(computing_and_electronics)>
* <https://zipcpu.com/blog/2017/07/29/fifo.html>

We would also like to express our special thanks of gratitude to our professors (Dr. V. Narendar and Dr. M. Satish) who gave us the golden opportunity to do this wonderful project on the topic “Design and Implementation of FIFO Memory”, which also helped us in doing a lot of research and we came to know about so many new and useful things, the knowledge of which will definitely help us in the coming years.