Lab Report for NGSpice, Verilog, Yosys, Qrouter, Graywolf and Magic Simulation

Yash Kumar 21BEC059

Introduction to VLSI Design EC301

1 Introduction

This report provides a detailed analysis of fundamental digital logic components: the Inverter, NAND, NOR, XOR gates, and the D Flip-Flop. Each component's unique characteristics, operational theory, and truth table are discussed using Magic and NGSpice simulation.

2 Inverter

2.1 Aim

To understand the operation of the Inverter gate, which is the simplest form of a logic gate, and its role in inverting the logic state of an input signal.

2.2 Theory

The Inverter, or NOT gate, is a basic logic gate that outputs the opposite state to its input. It serves as a fundamental building block in digital circuits.

2.3 Truth Table

Input	Output
0	1
1	0

2.4 Schematic of Inverter

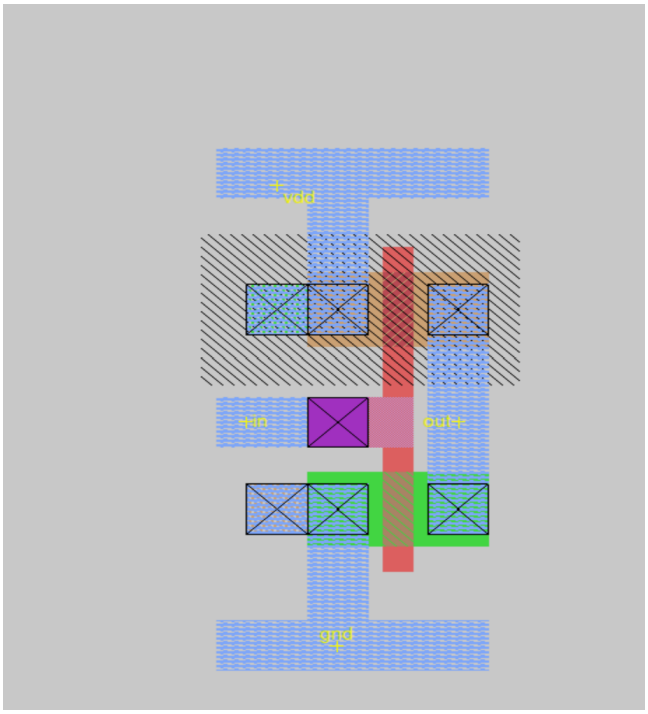


Figure 1: Schematic of Inverter Gate

2.5 SPICE Code for Inverter

```
* SPICE3 file created from inv.ext - technology: scmos

.option scale=1u
.include techfile130.txt
.subckt inv vdd in out gnd
M1000 out in vdd vdd pmos w=6 l=2
+  ad=30 pd=22 as=30 ps=22
M1001 out in gnd Gnd nmos w=6 l=2
+  ad=30 pd=22 as=30 ps=22
C0 gnd Gnd 2.44fF
C1 in Gnd 4.95fF
C2 vdd Gnd 3.95fF
.ends
```

2.6 Graph of Inverter

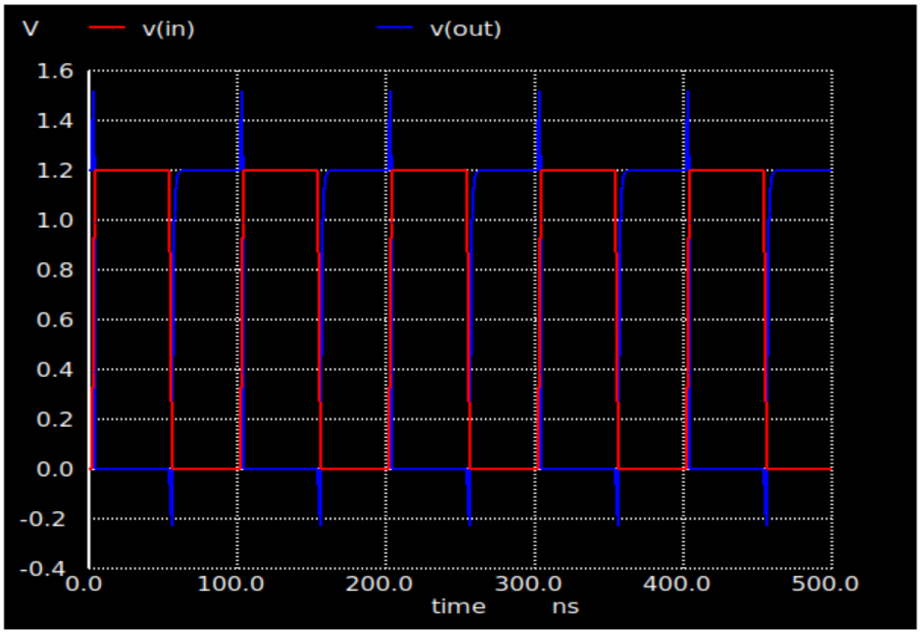


Figure 2: Graph showing the behavior of Inverter Gate

2.7 Results of Inverter Simulation

The Inverter gate was successfully designed and simulated using Magic for layout design and ngspice for circuit simulation. The simulation results confirm the expected behavior of an Inverter, where a logic high input results in a logic low output and vice versa, as illustrated in the graph.

3 NAND Gate

3.1 Aim

To analyze the NAND gate, which is a universal gate, and explore how it forms the basis for constructing other basic gates and complex digital circuits.

3.2 Theory

The NAND gate is a combination of an AND gate followed by an Inverter. It outputs false only when all its inputs are true. It is known as a universal gate due to its ability to create any other logic gate through combinations.

3.3 Truth Table

A	B	Output
0	0	1
0	1	1
1	0	1
1	1	0

3.4 Schematic of NAND

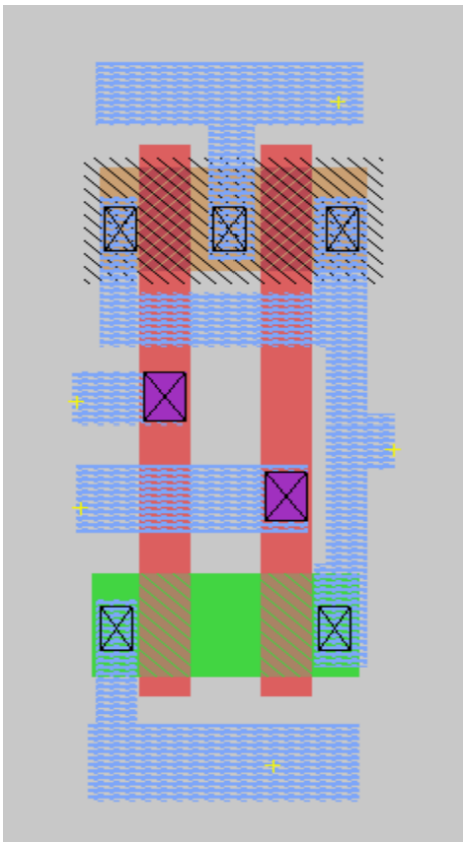


Figure 3: Schematic of NAND Gate

3.5 SPICE Code for NAND

```
* SPICE3 file created from nand.ext - technology: scmos
.include techfile130.txt
.option scale=1u
.subckt NAND Vdd A B Out Gnd
M1000 Vdd A Out w_22_n29# pmos w=31 l=12
+  ad=558 pd=98 as=744 ps=172
M1001 a_48_n148# A Gnd Gnd nmos w=31 l=12
+  ad=558 pd=98 as=372 ps=86
M1002 Out B Vdd w_22_n29# pmos w=31 l=12
+  ad=0 pd=0 as=0 ps=0
M1003 Out B a_48_n148# Gnd nmos w=31 l=12
+  ad=372 pd=86 as=0 ps=0
C0 A w_22_n29# 6.32fF
```

```
C1 B Out 5.76fF
C2 A B 7.20fF
C3 A Out 5.76fF
C4 B w_22_n29# 6.32fF
C5 Out w_22_n29# 4.14fF
C6 Gnd Gnd 32.99fF
C7 Vdd Gnd 64.11fF
C8 Out Gnd 75.44fF
C9 B Gnd 19.02fF
C10 A Gnd 20.17fF
.ends
```

3.6 Graph of NAND

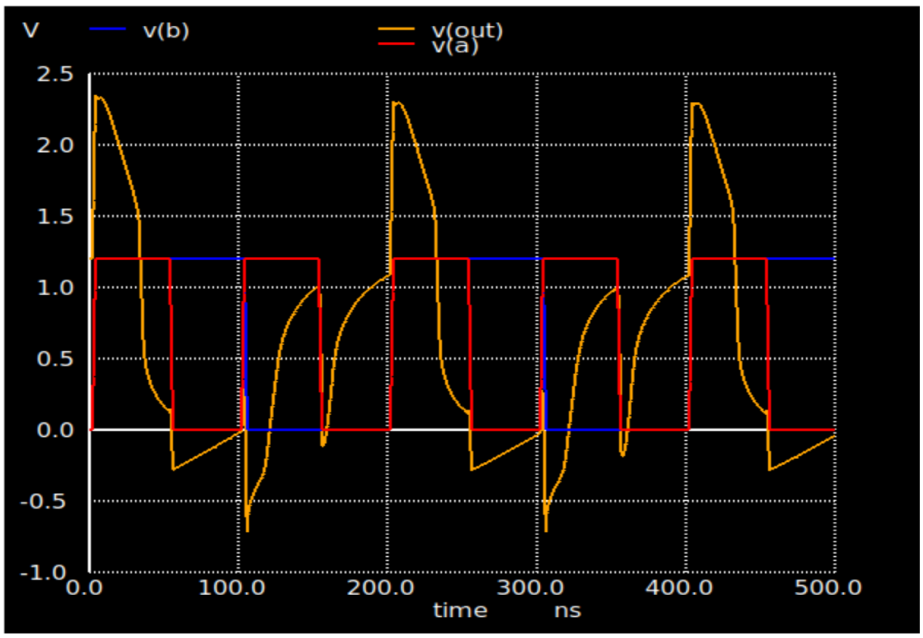


Figure 4: Graph showing the behavior of NAND Gate

3.7 Results of NAND Simulation

The NAND gate, a fundamental universal gate, was effectively designed and simulated. The simulation results, as shown in the graph, demonstrate the gate's characteristic functionality, where the output is low only when both inputs are high, validating its correct operation.

4 NOR Gate

4.1 Aim

To investigate the NOR gate, another universal gate, and its application in creating basic as well as complex logic circuits.

4.2 Theory

The NOR gate combines an OR gate with an Inverter. Its output is true only when all inputs are false. Like the NAND gate, it is also a universal gate and can form any other logic gate.

4.3 Truth Table

A	B	Output
0	0	1
0	1	0
1	0	0
1	1	0

4.4 Schematic of NOR

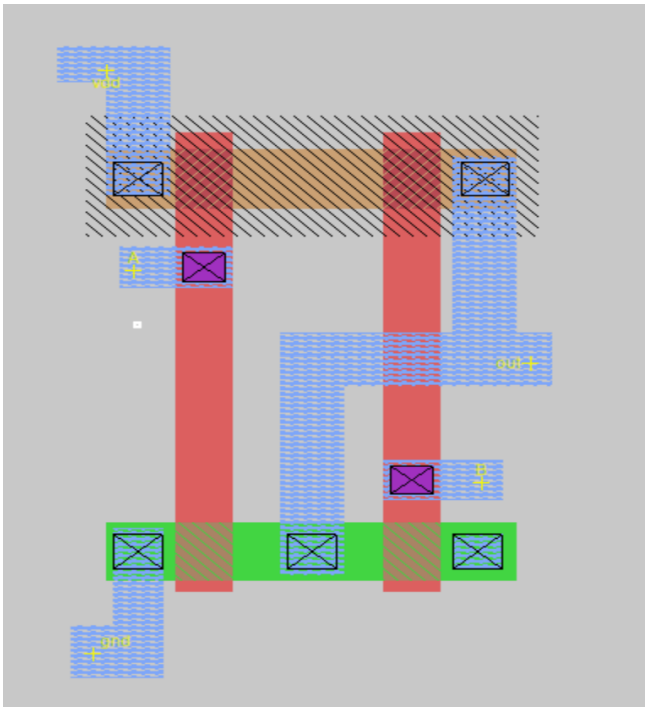


Figure 5: Schematic of NOR Gate

4.5 SPICE Code for NOR

```
* SPICE3 file created from nor.ext - technology: scmos

.include techfile130.txt
.option scale=1u
.subckt nor Vdd A B Out Gnd
M1000 a_44_n44# B out Gnd nmos w=10 l=8
+  ad=110 pd=42 as=220 ps=64
M1001 out A gnd Gnd nmos w=10 l=8
+  ad=0 pd=0 as=100 ps=40
M1002 out B a_14_21# w_n7_16# pmos w=10 l=8
+  ad=110 pd=42 as=220 ps=64
M1003 a_14_21# A vdd w_n7_16# pmos w=10 l=8
+  ad=0 pd=0 as=100 ps=40
C0 w_n7_16# A 5.70fF
C1 w_n7_16# B 5.70fF
C2 out B 2.16fF
C3 w_n7_16# vdd 2.54fF
```

```
C4 out w_n7_16# 2.12fF
C5 gnd Gnd 4.98fF
C6 out Gnd 26.65fF
C7 vdd Gnd 7.05fF
C8 B Gnd 27.23fF
C9 A Gnd 25.40fF
.ends
```

4.6 Graph of NOR

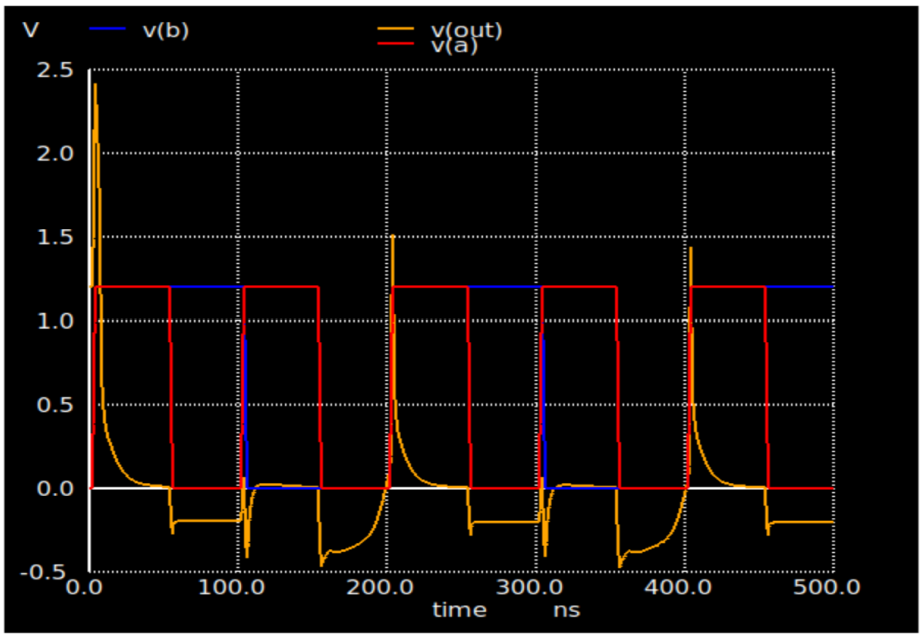


Figure 6: Graph showing the behavior of NOR Gate

4.7 Results of NOR Simulation

The NOR gate simulation was successfully carried out. The results indicate that the gate performs as expected, outputting a high signal only when both inputs are low. This aligns with the theoretical understanding of the NOR gate's operation.

5 XOR Gate

5.1 Aim

To elucidate the functioning of the XOR gate, a key component in arithmetic and comparison operations in digital electronics.

5.2 Theory

The XOR (Exclusive OR) gate outputs true only when the inputs differ. It is pivotal in digital arithmetic operations, such as addition, where it is used for calculating sums.

5.3 Truth Table

A	B	Output
0	0	0
0	1	1
1	0	1
1	1	0

5.4 Schematic of XOR

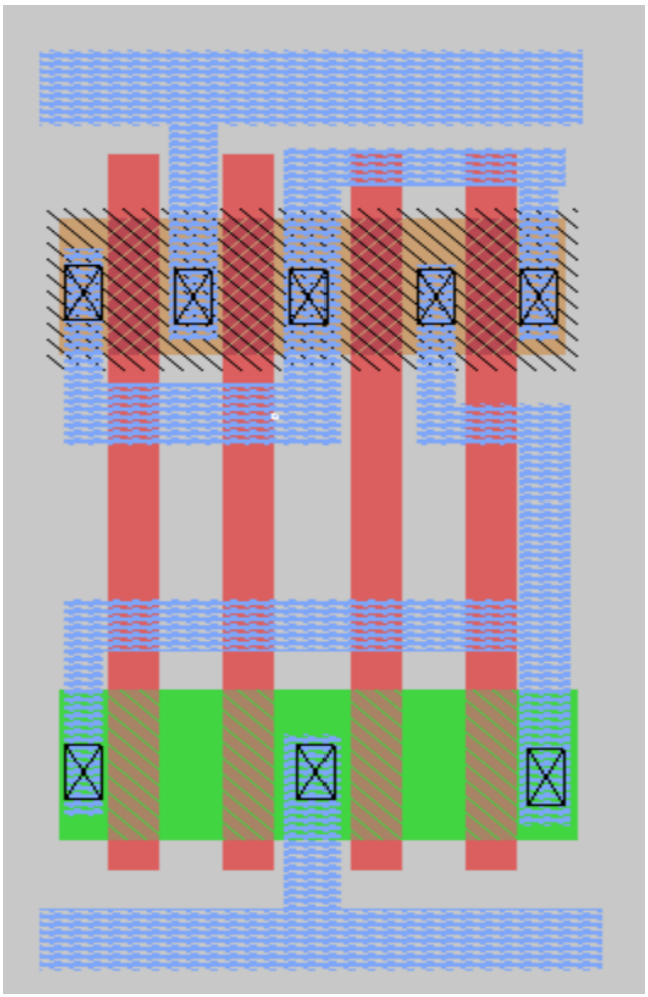


Figure 7: Schematic of XOR Gate

5.5 SPICE Code for XOR

```
* SPICE3 file created from xor.ext - technology: scmos

.option scale=1u
.include techfile130.txt
.subckt XOR vdd A B Abar Bbar out gnd
M1000 out Abar a_21_n84# Gnd nmos w=30 l=8
+  ad=540 pd=156 as=330 ps=82
M1001 a_n35_13# B vdd w_n37_10# pmos w=27 l=8
```

```
+  ad=783 pd=220 as=297 ps=76
M1002 vdd A a_n35_13# w_n37_10# pmos w=27 l=8
+  ad=0 pd=0 as=0 ps=0
M1003 a_n19_n84# A out Gnd nmos w=30 l=8
+  ad=330 pd=82 as=0 ps=0
M1004 a_21_n84# Bbar gnd Gnd nmos w=30 l=8
+  ad=0 pd=0 as=390 ps=86
M1005 out Bbar a_n35_13# w_n37_10# pmos w=27 l=8
+  ad=297 pd=76 as=0 ps=0
M1006 gnd B a_n19_n84# Gnd nmos w=30 l=8
+  ad=0 pd=0 as=0 ps=0
M1007 a_n35_13# Abar out w_n37_10# pmos w=27 l=8
+  ad=0 pd=0 as=0 ps=0
C0 out B 2.40fF
C1 w_n37_10# Bbar 3.64fF
C2 a_n35_13# B 2.88fF
C3 out Bbar 2.40fF
C4 gnd Bbar 2.88fF
C5 Abar w_n37_10# 3.64fF
C6 A w_n37_10# 3.64fF
C7 B w_n37_10# 3.64fF
C8 Abar out 4.32fF
C9 A out 2.40fF
C10 A a_n35_13# 2.88fF
C11 a_n35_13# w_n37_10# 3.52fF
C12 gnd Abar 2.88fF
C13 gnd Gnd 45.97fF
C14 out Gnd 32.62fF
C15 vdd Gnd 11.42fF
C16 a_n35_13# Gnd 7.61fF
C17 Abar Gnd 62.48fF
C18 Bbar Gnd 62.95fF
C19 B Gnd 21.24fF
C20 A Gnd 18.14fF
.ends
```

5.6 Graph of XOR

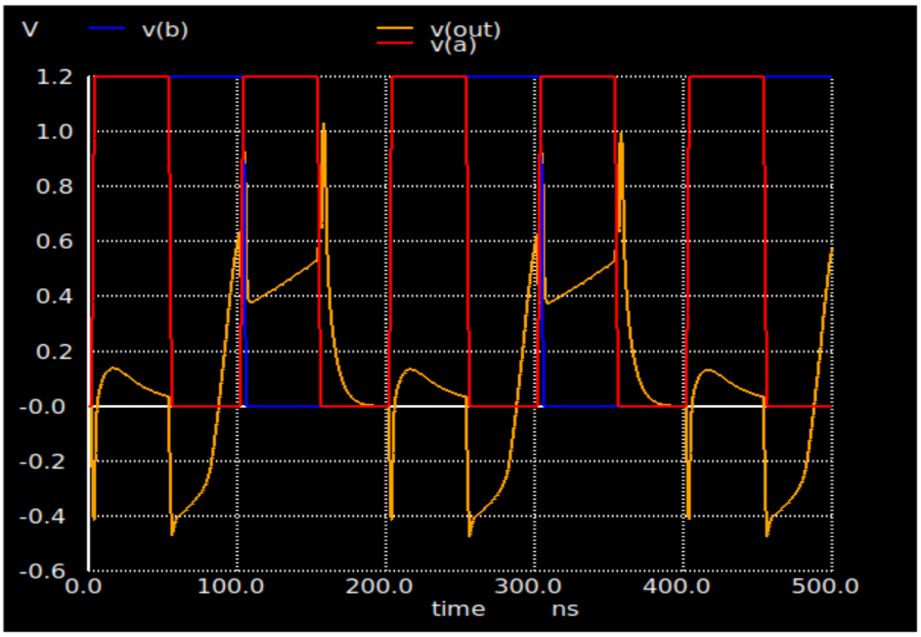


Figure 8: Graph showing the behavior of XOR Gate

5.7 Results of XOR Simulation

The XOR gate was designed and simulated, showcasing its unique behavior where the output is high only when the inputs are different. The simulation graphs clearly depict this exclusive behavior, confirming the gate's intended functionality.

6 D Flip-Flop

6.1 Aim

To understand the D Flip-Flop, a fundamental storage element in digital electronics, and its role in data storage and transfer.

6.2 Theory

The D Flip-Flop is a memory element that stores one bit of data. It captures the input value on a clock edge and maintains it until the next clock event. This makes it essential for synchronization and data storage in digital systems.

6.3 Truth Table

D (Data)	Clock	Q (Output)
0	↑	0
1	↑	1

Note: ↑ indicates a rising edge of the clock signal.

6.4 Schematic of D Flip-Flop

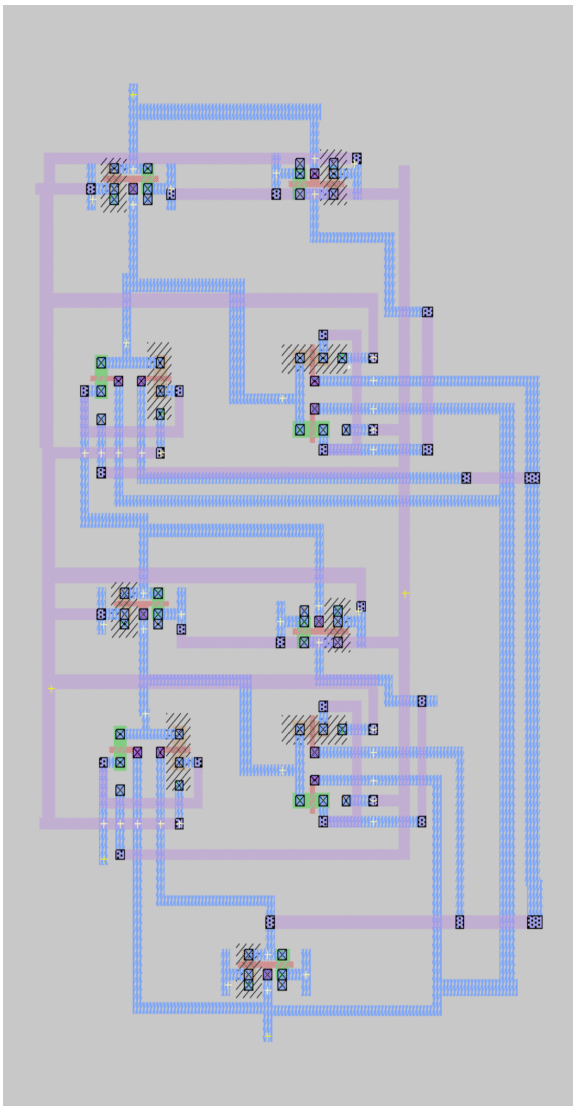


Figure 9: Schematic of D Flip-Flop

6.5 SPICE Code for D Flip-Flop

```
* SPICE3 file created from dff.ext - technology: scmos

.option scale=1u
.include techfile130.txt
```

```
.subckt DFF vdd vin phi out gnd
M1000 inv_3/in inv_0/in vdd vdd pmos w=6 l=2
+  ad=72 pd=48 as=120 ps=88
M1001 inv_3/in inv_0/in gnd Gnd nmos w=6 l=2
+  ad=72 pd=48 as=120 ps=88
M1002 out inv_1/in vdd vdd pmos w=6 l=2
+  ad=30 pd=22 as=0 ps=0
M1003 out inv_1/in gnd Gnd nmos w=6 l=2
+  ad=30 pd=22 as=0 ps=0
M1004 inv_0/in phi vin Gnd nmos w=6 l=2
+  ad=96 pd=56 as=42 ps=26
M1005 inv_0/in inv_4/out vin vdd pmos w=6 l=2
+  ad=96 pd=56 as=42 ps=26
M1006 inv_2/out out vdd vdd pmos w=6 l=2
+  ad=72 pd=48 as=0 ps=0
M1007 inv_2/out out gnd Gnd nmos w=6 l=2
+  ad=72 pd=48 as=0 ps=0
M1008 inv_1/in phi inv_3/in Gnd nmos w=6 l=2
+  ad=96 pd=56 as=0 ps=0
M1009 inv_1/in inv_4/out inv_3/in vdd pmos w=6 l=2
+  ad=96 pd=56 as=0 ps=0
M1010 inv_3/out inv_3/in vdd vdd pmos w=6 l=2
+  ad=72 pd=48 as=0 ps=0
M1011 inv_3/out inv_3/in gnd Gnd nmos w=6 l=2
+  ad=72 pd=48 as=0 ps=0
M1012 inv_1/in phi inv_2/out Gnd nmos w=6 l=2
+  ad=0 pd=0 as=0 ps=0
M1013 inv_1/in inv_4/out inv_2/out vdd pmos w=6 l=2
+  ad=0 pd=0 as=0 ps=0
M1014 inv_4/out phi inv_4/vdd inv_4/vdd pmos w=6 l=2
+  ad=30 pd=22 as=30 ps=22
M1015 inv_4/out phi inv_4/gnd Gnd nmos w=6 l=2
+  ad=30 pd=22 as=30 ps=22
M1016 inv_0/in phi inv_3/out Gnd nmos w=6 l=2
+  ad=0 pd=0 as=0 ps=0
M1017 inv_0/in inv_4/out inv_3/out vdd pmos w=6 l=2
+  ad=0 pd=0 as=0 ps=0
C0 gnd inv_4/out 14.94fF
C1 inv_2/out gnd 6.30fF
C2 vdd gnd 4.42fF
C3 vdd out 3.67fF
C4 inv_3/out gnd 6.30fF
C5 vdd inv_4/out 7.48fF
C6 inv_1/in vdd 5.27fF
C7 inv_3/in vdd 6.21fF
C8 vdd inv_3/out 7.06fF
C9 phi gnd 4.14fF
C10 vdd inv_0/in 12.74fF
C11 phi inv_4/out 3.73fF
C12 inv_4/out Gnd 49.10fF
```

```
C13 inv_0/in Gnd 42.13fF
C14 gnd Gnd 24.51fF
C15 inv_3/out Gnd 32.75fF
C16 phi Gnd 239.84fF
C17 vdd Gnd 29.23fF
C18 inv_4/gnd Gnd 2.44fF
C19 inv_4/vdd Gnd 3.95fF
C20 inv_1/in Gnd 28.12fF
C21 inv_2/out Gnd 40.34fF
C22 inv_3/in Gnd 22.52fF
C23 out Gnd 39.92fF
C24 vin Gnd 12.70fF
.ends
```

6.6 Graph of D Flip-Flop

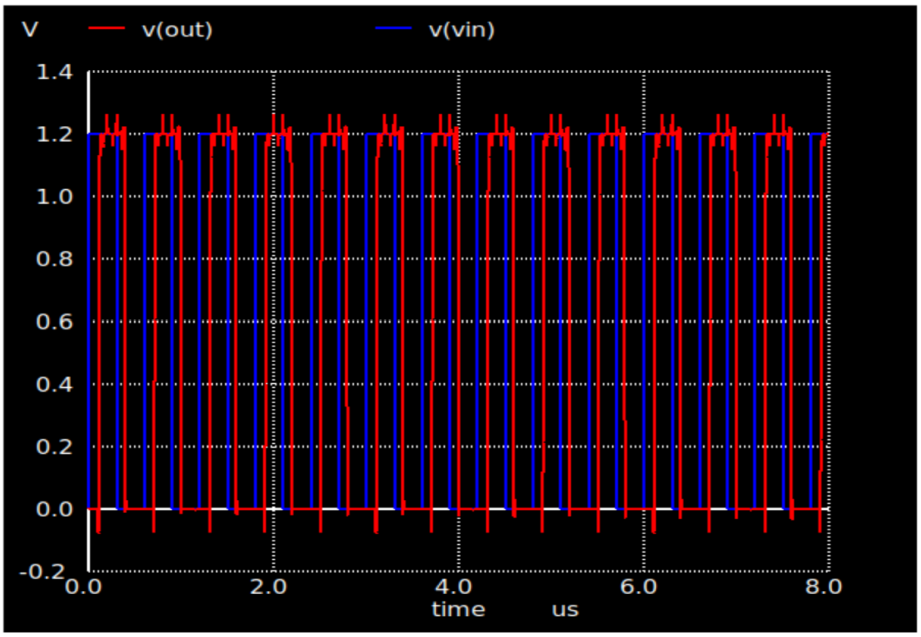


Figure 10: Graph showing the behavior of D Flip-Flop

6.7 Results of D Flip-Flop Simulation

The D Flip-Flop, an essential component in digital memory and storage, was simulated successfully. The simulation results demonstrate its capability to store a bit of data, changing the output only at the rising edge of the clock signal, as expected from its design.

7 Inverter

7.1 Theory

An inverter is one of the fundamental building blocks in digital and analog electronic circuits. It is a simple electronic device that performs the logical NOT operation, which means it takes an input signal and produces the complement of that signal at its output. Inverters are widely used in digital logic circuits, amplifiers, and other electronic applications. ...

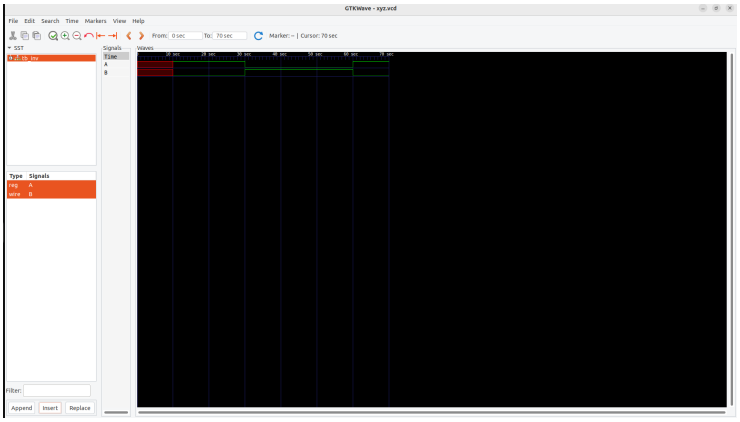
7.2 Verilog Code

```
module inv(input A, output B);
        assign B =~A;
endmodule
```

7.3 Testbench

```
`include "inv.v"
module tb_inv();
        reg A;
        wire B;
        inv inv1(A,B);

        initial
                begin
                        $dumpfile("xyz.vcd");
                        $dumpvars;
                        #10 A <= 1'b1;
                        #20 A <= 1'b0;
                        #30 A <= 1'b1;
                        #10 $finish;
                end
endmodule
```

7.4 GTKWave Output



8 AND Gate

8.1 Theory

An AND gate is a fundamental digital logic gate that performs the logical AND operation on two or more input signals. It takes multiple binary inputs (usually two or more) and produces a single binary output based on the AND logic function. The AND gate is widely used in digital electronics and plays a critical role in Boolean algebra and digital circuit design. ...

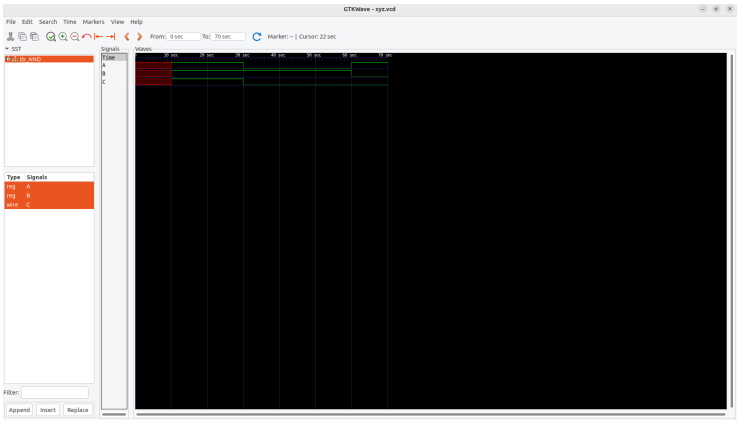
8.2 Verilog Code

```
module AND(input A,B, output C);
        assign C = A&B;
endmodule
```

8.3 Testbench

```
`include "and.v"
module tb_AND();
        reg A;
        reg B;
        wire C;
        AND and1(A,B,C);

        initial
                begin
                        $dumpfile("xyz.vcd");
                        $dumpvars;
                        #10 A <= 1'b1;
                            B <= 1'b1;
                        #20 A <= 1'b0;
                            B <= 1'b1;
                        #30 A <= 1'b1;
                            B <= 1'b0;
                        #10 $finish;
                end
endmodule
```

8.4 GTKWave Output



9 Decoder

9.1 Theory

A decoder is a digital circuit that converts binary information from the 'n' coded inputs into a maximum of 2^n unique outputs. They are used for a variety of tasks in digital circuits, including data demultiplexing, memory address decoding, and binary-to-numeric decoding.

9.2 Behavioral Design

Behavioral design in Verilog abstracts the circuit logic at a higher level, focusing on the functionality rather than the specific hardware implementation. This approach is more akin to traditional programming.

9.2.1 Verilog Code

```
module DECODER(input a,b,e ,output reg [3:0] d);

 always @(a or b or e) begin
          case ({e,a,b})
               3'b000: d = 4'b0000;
               3'b001: d = 4'b0000;
               3'b010: d = 4'b0000; ‘
               3'b011: d = 4'b0000;
               3'b100: d = 4'b0001;
               3'b101: d = 4'b0010;
               3'b110: d = 4'b0100;
               3'b111: d = 4'b1000;
            endcase
         end
endmodule
```

9.3 Structural Design

Structural design in Verilog involves constructing the circuit using interconnected components like gates and flip-flops. This approach closely represents the actual hardware layout, focusing on how different parts are connected to form the overall circuit.

9.3.1 Verilog Code

```
module DECODER(input a,b,e ,output [3:0] d);
          wire a_bar,b_bar;
          not n1(a_bar,a);
          not n2(b_bar,b);
```

```
    and #2 a1(d[0] , a_bar,b_bar,e);//,a2(d[1] , a_bar,b,e),a3(d[2] , a,b_b
    and #2 a2(d[1] , a_bar,b,e);
    and #2 a3(d[2] , a,b_bar,e);
    and #2 a4(d[3] , a,b,e);

endmodule
```

9.4 Dataflow Design

Dataflow design in Verilog represents the circuit in terms of the flow of data between operations. It uses continuous assignments to describe the relationship between inputs and outputs, often resulting in a more compact and readable code that directly maps to the intended logic operations.

9.4.1 Verilog Code

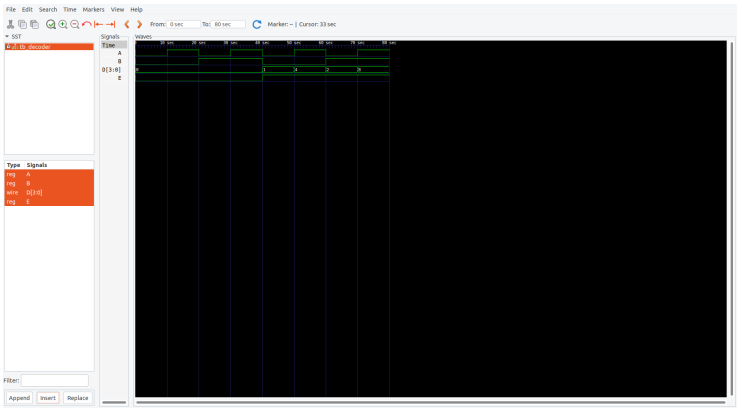
```
module DECODER(input a,b,e ,output [3:0] d);
      assign d[0] = ~a & ~b & e;
      assign d[1] = ~a & b & e;
      assign d[2] = a & ~b & e;
      assign d[3] = a & b & e;

endmodule
```

9.4.2 Testbench

```
module tb_decoder();
    reg A,B,E;
    wire [3:0]D;
    DECODER dec1(A,B,E , D);
    initial
        begin
            $dumpfile("xyz.vcd");
            $dumpvars;
                E<=1'b0;
                A <= 1'b0;
                B <= 1'b0;
            #10 E<=1'b0;
                A <= 1'b1;
                B <= 1'b0;
            #10 E<=1'b0;
                A <= 1'b0;
                B <= 1'b1;
            #10 E<=1'b0;
                A <= 1'b1;
```

```
        B <= 1'b1;
    #10 E<=1'b1;
        A <= 1'b0;
        B <= 1'b0;
    #10 E<=1'b1;
        A <= 1'b1;
        B <= 1'b0;
    #10 E<=1'b1;
        A <= 1'b0;
        B <= 1'b1;
    #10 E<=1'b1;
        A <= 1'b1;
        B <= 1'b1;

    #10 $finish;
    end
    always @(A or B or E)
$monitor(At TIME(in ns) = %t, A = %D B = %d D= %d,
$time ,A,B,D)
endmodule
```

9.4.3 GTKWave Output



10 Full Adder in Verilog

10.1 Theory

A full adder is a fundamental digital circuit that computes the sum of three bits, typically two input bits and a carry-in bit. It is a key element in arithmetic logic units (ALUs) within CPUs and is essential for performing binary addition.

10.2 Behavioral Design

Behavioral design in Verilog abstracts the circuit logic at a higher level, focusing on the functionality rather than the specific hardware implementation. This approach is more akin to traditional programming.

10.2.1 Verilog Code

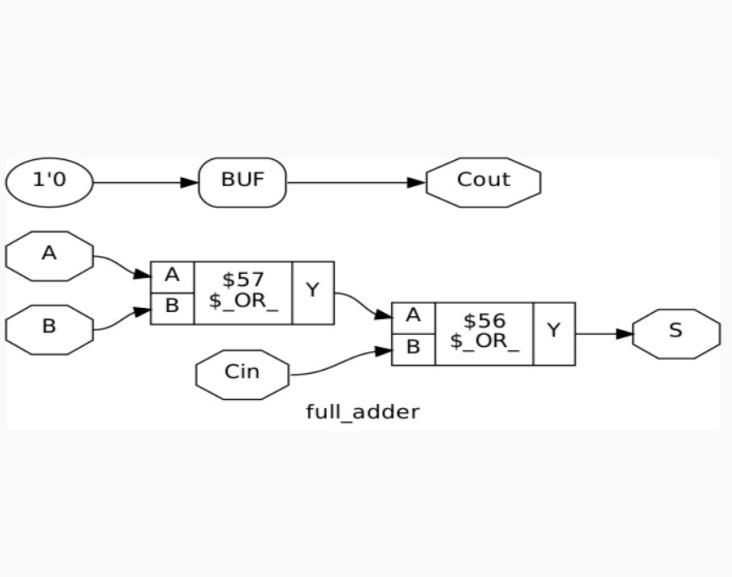
```
‘timescale 1ns / 1ps
module full_adder( A, B, Cin, S, Cout);

 input wire A, B, Cin;
 output reg S, Cout;


  always @(A or B or Cin)
  begin
   case (A | B | Cin)
     3'b000: begin S = 0; Cout = 0; end
     3'b001: begin S = 1; Cout = 0; end
     3'b010: begin S = 1; Cout = 0; end
     3'b011: begin S = 0; Cout = 1; end
     3'b100: begin S = 1; Cout = 0; end
     3'b101: begin S = 0; Cout = 1; end
     3'b110: begin S = 0; Cout = 1; end
     3'b111: begin S = 1; Cout = 1; end
   endcase
  end
endmodule
```

10.2.2 Yosys Synthesis Verilog Code

```
/* Generated by Yosys 0.9 (git sha1 1979e0b) */

(* top = 1 *)
(* src = "adder_beh.v:4" *)
```

```
module full_adder(A, B, Cin, S, Cout);
  (* src = "adder_beh.v:6" *)
  wire _0_;
  (* src = "adder_beh.v:6" *)
  wire _1_;
  (* src = "adder_beh.v:6" *)
  wire _2_;
  (* src = "adder_beh.v:7" *)
  wire _3_;
  wire _4_;
  wire _5_;
  (* src = "adder_beh.v:12" *)
  wire _6_;
  (* src = "adder_beh.v:6" *)
  input A;
  (* src = "adder_beh.v:6" *)
  input B;
  (* src = "adder_beh.v:6" *)
  input Cin;
  (* src = "adder_beh.v:7" *)
  output Cout;
  (* src = "adder_beh.v:7" *)
  output S;
  INVX1 _7_ (
    .A(_1_),
    .Y(_4_)
  );
  NOR2X1 _8_ (
    .A(_2_),
    .B(_0_),
    .Y(_5_)
  );
  NAND2X1 _9_ (
    .A(_4_),
    .B(_5_),
    .Y(_3_)
  );
  assign Cout = 1'h0;
  assign _2_ = Cin;
  assign S = _3_;
  assign _0_ = A;
  assign _1_ = B;
endmodule
```

10.2.3 Yosys Synthesised Design



10.3 Structural Design

Structural design in Verilog involves constructing the circuit using interconnected components like gates and flip-flops. This approach closely represents the actual hardware layout, focusing on how different parts are connected to form the overall circuit.

10.3.1 Verilog Code

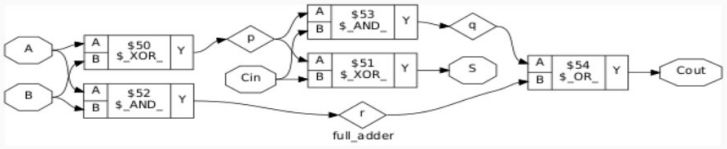
```
‘timescale 1ns / 1ps
module full_adder( A, B, Cin, S, Cout);

 input A, B, Cin;
 output S, Cout;
 wire p,q,r;

 xor(p,A,B);
 xor(S,p,Cin);
 and(r,A,B);
 and(q,p,Cin);
 or(Cout,q,r);

endmodule
```

10.3.2 Yosys Synthesis Verilog Code

```
/* Generated by Yosys 0.9 (git sha1 1979e0b) */

(* top =  1  *)
(* src = "adder_struct.v:2" *)
module full_adder(A, B, Cin, S, Cout);
  (* src = "adder_struct.v:4" *)
  wire _00_;
  (* src = "adder_struct.v:4" *)
  wire _01_;
  (* src = "adder_struct.v:4" *)
  wire _02_;
  (* src = "adder_struct.v:5" *)
  wire _03_;
  (* src = "adder_struct.v:5" *)
  wire _04_;
  wire _05_;
  wire _06_;
  wire _07_;
  (* src = "adder_struct.v:4" *)
  input A;
  (* src = "adder_struct.v:4" *)
  input B;
  (* src = "adder_struct.v:4" *)
  input Cin;
  (* src = "adder_struct.v:5" *)
  output Cout;
  (* src = "adder_struct.v:5" *)
  output S;
  (* src = "adder_struct.v:6" *)
  wire p;
  (* src = "adder_struct.v:6" *)
  wire q;
  (* src = "adder_struct.v:6" *)
  wire r;
  INVX1 _08_ (
    .A(_02_),
    .Y(_05_)
  );
  NAND2X1 _09_ (
    .A(_00_),
    .B(_01_),
    .Y(_06_)
  );
  XNOR2X1 _10_ (
```

```
    .A(_00_),
    .B(_01_),
    .Y(_07_)
  );
  XNOR2X1 _11_ (
    .A(_02_),
    .B(_07_),
    .Y(_04_)
  );
  OAI21X1 _12_ (
    .A(_05_),
    .B(_07_),
    .C(_06_),
    .Y(_03_)
  );
  assign _00_ = A;
  assign _01_ = B;
  assign _02_ = Cin;
  assign S = _04_;
  assign Cout = _03_;
endmodule
```

10.3.3 Yosys Synthesised Design



10.4 Dataflow Design

Dataflow design in Verilog represents the circuit in terms of the flow of data between operations. It uses continuous assignments to describe the relationship between inputs and outputs, often resulting in a more compact and readable code that directly maps to the intended logic operations.

10.4.1 Verilog Code

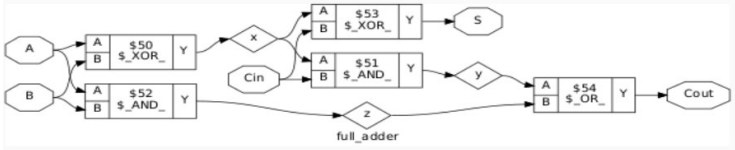
```
`timescale 1ns / 1ps
module full_adder( A, B, Cin, S, Cout);

 input A, B, Cin;
 output S, Cout;
 wire x,y,z;

assign x = A ^ B ;        // even one single expression can do.
assign y = x & Cin ;
assign z = A & B ;
assign S = x ^ Cin ;
assign Cout = y | z ;

endmodule
```

10.4.2 Yosys Synthesis Verilog Code

```
/* Generated by Yosys 0.9 (git sha1 1979e0b) */

(* top =  1  *)
(* src = "adder_dataflow.v:4" *)
module full_adder(A, B, Cin, S, Cout);
  (* src = "adder_dataflow.v:6" *)
  wire _00_;
  (* src = "adder_dataflow.v:6" *)
  wire _01_;
  (* src = "adder_dataflow.v:6" *)
  wire _02_;
  (* src = "adder_dataflow.v:7" *)
  wire _03_;
  (* src = "adder_dataflow.v:7" *)
  wire _04_;
  wire _05_;
  wire _06_;
  wire _07_;
  (* src = "adder_dataflow.v:6" *)
```

```
  input A;
  (* src = "adder_dataflow.v:6" *)
  input B;
  (* src = "adder_dataflow.v:6" *)
  input Cin;
  (* src = "adder_dataflow.v:7" *)
  output Cout;
  (* src = "adder_dataflow.v:7" *)
  output S;
  (* src = "adder_dataflow.v:8" *)
  wire x;
  (* src = "adder_dataflow.v:8" *)
  wire y;
  (* src = "adder_dataflow.v:8" *)
  wire z;
  INVX1 _08_ (
    .A(_02_),
    .Y(_05_)
  );
  NAND2X1 _09_ (
    .A(_00_),
    .B(_01_),
    .Y(_06_)
  );
  XNOR2X1 _10_ (
    .A(_00_),
    .B(_01_),
    .Y(_07_)
  );
  XNOR2X1 _11_ (
    .A(_02_),
    .B(_07_),
    .Y(_04_)
  );
  OAI21X1 _12_ (
    .A(_05_),
    .B(_07_),
    .C(_06_),
    .Y(_03_)
  );
  assign _00_ = A;
  assign _01_ = B;
  assign _02_ = Cin;
  assign S = _04_;
  assign Cout = _03_;
endmodule
```

10.4.3 Yosys Synthesised Design

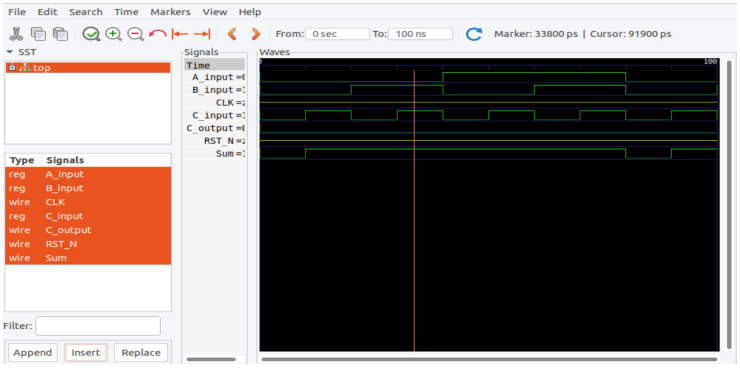


10.5 Testbench

```
‘timescale 1ns/1ps
‘include "adder_dataflow.v"
‘include "osu018_stdcells.v"

module top(input CLK, RST_N);
  //declare testbench variables
  reg  A_input, B_input, C_input;
  wire Sum, C_output;

  full_adder instantiation(.A(A_input), .B(B_input),
  .Cin(C_input), .S(Sum), .Cout(C_output));

  initial
    begin
      $dumpfile("xyz.vcd");
      $dumpvars;

      //set stimulus to test the code
      A_input=0;
      B_input=0;
      C_input=0;
      #100 $finish;
```

```
    end

always #40 A_input=~A_input;
always #20 B_input=~B_input;
always #10 C_input=~C_input;

  always @(A_input or B_input or C_input)
      $monitor("At TIME(in ns)=%t, A=%d B=%d C=%d Sum=%d
------Carry=%d", $time, A_input, B_input, C_input, Sum, C_output);

endmodule
```

10.6 GTKWave Output



11 Full Adder Using Half Adders

11.1 Theory

A full adder can also be constructed using two half adders and an OR gate. This approach demonstrates the modularity of digital circuits, where complex units can be built from simpler components.

...

11.2 Verilog Code

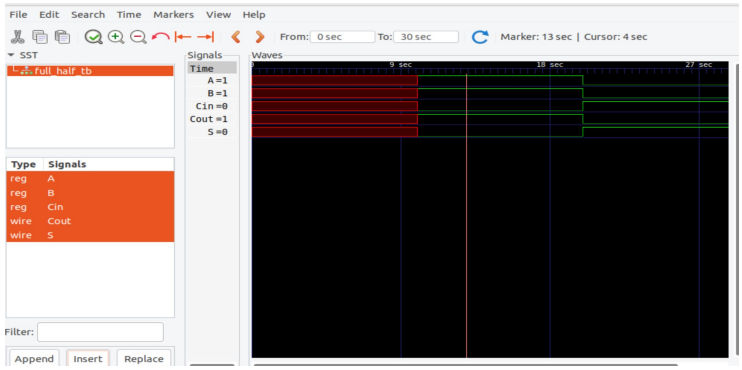
```
module HA(
    input A,B,
    output X,Y
);

xor(X,A,B);
and(Y,A,B);

endmodule


module FA(
    input A,B,Cin,
    output S, Cout
);

wire w1,w2,w3;

HA ha1(A,B,w1,w2);
HA ha2(w1,Cin,S,w3);
or(Cout, w3,w2);

endmodule
```

11.3 Testbench

```
`include "full_half.v"

module full_half_tb;
reg A,B,Cin;
wire S, Cout;
FA dut(A,B,Cin,S,Cout);
initial
begin
```

```
      $dumpfile("full_adder.vcd");
      $dumpvars(1,full_half_tb);
#10
A<=1;B<=1;Cin<=0;
#10
A<=0;B<=0;Cin<=1;
#10
$finish();
end

endmodule
```

11.4 GTKWave Output



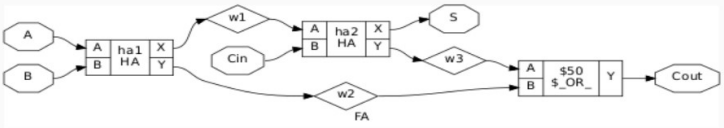
11.5 Yosys Synthesis Verilog Code

```
/* Generated by Yosys 0.9 (git sha1 1979e0b) */

(* top = 1 *)
(* src = "full_half.v:12" *)
module FA(A, B, Cin, S, Cout);
  (* src = "full_half.v:14" *)
  wire _0_;
  (* src = "full_half.v:17" *)
  wire _1_;
  (* src = "full_half.v:17" *)
  wire _2_;
  (* src = "full_half.v:13" *)
  input A;
  (* src = "full_half.v:13" *)
  input B;
  (* src = "full_half.v:13" *)
```

```
  input Cin;
  (* src = "full_half.v:14" *)
  output Cout;
  (* src = "full_half.v:14" *)
  output S;
  (* src = "full_half.v:17" *)
  wire w1;
  (* src = "full_half.v:17" *)
  wire w2;
  (* src = "full_half.v:17" *)
  wire w3;
  OR2X1 _3_ (
    .A(_2_),
    .B(_1_),
    .Y(_0_)
  );
  (* module_not_derived = 32'd1 *)
  (* src = "full_half.v:19" *)
  HA ha1 (
    .A(A),
    .B(B),
    .X(w1),
    .Y(w2)
  );
  (* module_not_derived = 32'd1 *)
  (* src = "full_half.v:20" *)
  HA ha2 (
    .A(w1),
    .B(Cin),
    .X(S),
    .Y(w3)
  );
  assign _2_ = w3;
  assign _1_ = w2;
  assign Cout = _0_;
endmodule

(* src = "full_half.v:1" *)
module HA(A, B, X, Y);
  (* src = "full_half.v:2" *)
  wire _0_;
  (* src = "full_half.v:2" *)
  wire _1_;
  (* src = "full_half.v:3" *)
  wire _2_;
  (* src = "full_half.v:3" *)
```

```
  wire _3_;
  (* src = "full_half.v:2" *)
  input A;
  (* src = "full_half.v:2" *)
  input B;
  (* src = "full_half.v:3" *)
  output X;
  (* src = "full_half.v:3" *)
  output Y;
  AND2X1 _4_ (
    .A(_0_),
    .B(_1_),
    .Y(_3_)
  );
  XOR2X1 _5_ (
    .A(_0_),
    .B(_1_),
    .Y(_2_)
  );
  assign _0_ = A;
  assign _1_ = B;
  assign Y = _3_;
  assign X = _2_;
endmodule
```

11.6 Yosys Synthesised Design



12 Synchronous RAM

12.1 Theory

Memory blocks are essential for data storage in digital systems. Synchronous RAM (Random Access Memory) is a type of memory that synchronizes with the system clock, ensuring consistent data access times.

12.2 Verilog Code

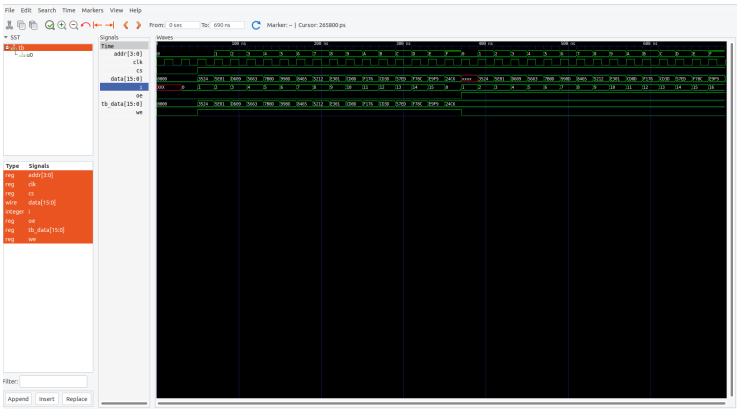
```
module single_port_sync_ram
 # (parameter ADDR_WIDTH = 4,
    parameter DATA_WIDTH = 16,
    parameter DEPTH = 16
   )

  ( input clk, cs, we, oe,
                input [ADDR_WIDTH-1:0]  addr,
                inout [DATA_WIDTH-1:0]  data
  );

  reg [DATA_WIDTH-1:0]  tmp_data;
  reg [DATA_WIDTH-1:0]  mem[DEPTH];

  always @ (posedge clk) begin
    if (cs & we)
      mem[addr] <= data;
  end

  always @ (posedge clk) begin
    if (cs & !we)
        tmp_data <= mem[addr];
  end

  assign data = cs & oe & !we ? tmp_data : 'hz;
endmodule
```

12.3 Testbench

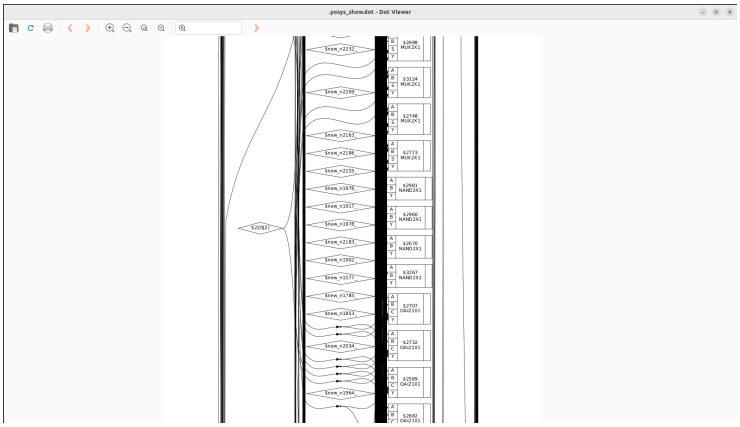
```
`timescale 1ns / 1ps
`include "single_port_sync_ram.v"

module tb(input CLK, RST_N);
  parameter ADDR_WIDTH = 4;
  parameter DATA_WIDTH = 16;
```

```
  parameter DEPTH = 16;

  reg clk;
  reg cs;
  reg we;
  reg oe;
  reg [ADDR_WIDTH-1:0] addr;
  wire [DATA_WIDTH-1:0] data;
  reg [DATA_WIDTH-1:0] tb_data;
  integer i;

  single_port_sync_ram #(.DATA_WIDTH(DATA_WIDTH)) u0 (.clk(clk),
  .addr(addr), .data(data), .cs(cs), .we(we), .oe(oe) );

  always #10 clk = ~clk;
  assign data = !oe ? tb_data : 'hz;

  initial
  begin
        $dumpfile("ram.vcd");
        $dumpvars;
        {clk, cs, we, addr, tb_data, oe} <= 0;

repeat (5)
            @(posedge clk);


for (i = 0; i < 2**ADDR_WIDTH; i= i+1)
            begin
                repeat (1)
                        @(posedge clk) addr <= i; we <= 1; cs <=1;
                        oe <= 0; tb_data <=$random;
            end

for (i = 0; i < 2**ADDR_WIDTH; i= i+1)
            begin
                repeat (1)
                        @(posedge clk) addr <= i; we <= 0; cs <= 1;
                        oe <= 1;
            end
        #20 $finish;
  end
endmodule
```

12.4 GTKWave Output



12.5 Yosys Synthesised Design



13 Melay State Machine

13.1 Theory

The Melay State Machine is a finite state machine where the output is determined by both the current state and the current inputs. It is commonly used in sequential logic design.

13.2 Verilog Code

```
‘timescale 1ns/1ps

module melay(
input  in, rst,clk,
output out

);
//state assignment
parameter ST_zero = 1'b0;
parameter ST_one = 1'b1;
reg out;
//assign registers
reg state, next_state;
//logic of fsm
always @(posedge clk )  //state updatation

    if(rst)begin
       state <= ST_zero;
    end
    else begin
        state <= next_state;
    end

always @( in, state)  //state computation and yput
  case (state)
    ST_zero:begin
        if(in)begin
            out <= 1'b1;
            next_state=ST_one;

        end
        else begin
            out <= 1'b0;
            next_state=state;
            end
    end
```

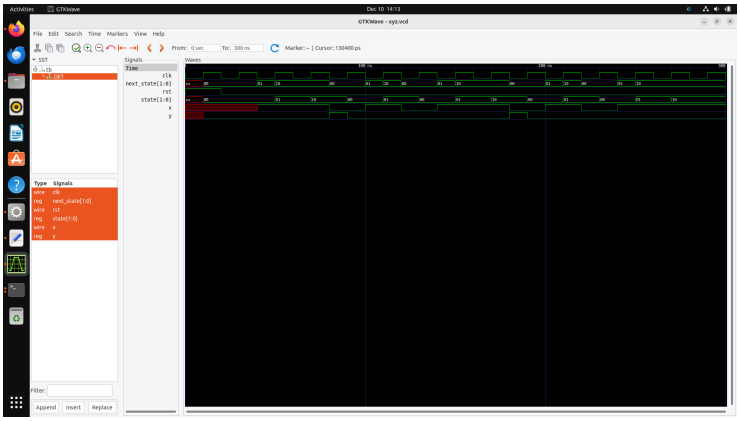
```
    ST_one: begin
        if(in)begin
            out <= 1'b1;
            next_state= state;
        end
        else begin
            out <= 1'b0;
            next_state = ST_zero;
        end
    end
endcase
endmodule
```

13.3 Testbench

```
`timescale 1ns/1ps

`include "melay.v"

module tb_melay();
reg in,rst;
wire out;
reg clk;
melay dut(in,rst,clk,out);

always begin
    clk<=0;
    #10;
    clk<=1;
    #10;
end

initial begin
    $dumpfile("melay.vcd");
    $dumpvars;

    rst <= 1;
    #20 rst<=0;
    #20 in<=0;
    #20 in<=1;
    #20 in<=0;
    #20 in<=1;
    #20 in<=0;
    #20 $finish;
end
```

endmodule

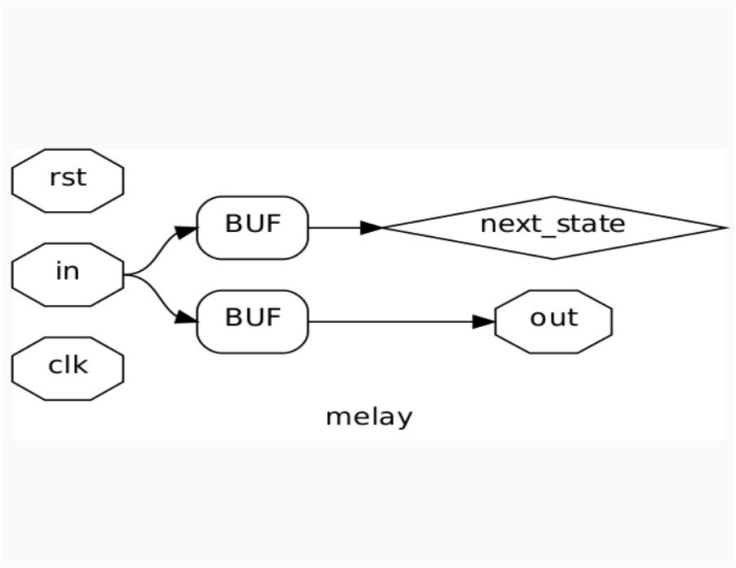
13.4 GTKWave Output



13.5 Yosys Synthesis Verilog Code

```
/* Generated by Yosys 0.9 (git sha1 1979e0b) */

(* top =  1  *)
(* src = "melay.v:3" *)
module melay(in, rst, clk, out);
  (* src = "melay.v:4" *)
  input clk;
  (* src = "melay.v:4" *)
  input in;
  (* src = "melay.v:13" *)
  wire next_state;
  (* src = "melay.v:5" *)
  output out;
  (* src = "melay.v:4" *)
  input rst;
  assign next_state = in;
  assign out = in;
endmodule
```

13.6 Yosys Synthesised Design



14 Moore State Machine

14.1 Theory

In contrast to the Mealy machine, a Moore State Machine's output is determined solely by its current state, making its output less sensitive to changes in input.

14.2 Verilog Code

```
module seq_det_moore(input clk, rst, x, output y);

reg y;
reg [1:0] state, next_state;
parameter a = 2'b00, b=2'b01, c=2'b10, d=2'b11;

always @(posedge clk)
        begin
                if(rst) state <=a;
                else state <= next_state;
        end

always @(x,state)

        case(state)
        a:
                begin
                        if(x) next_state = b;
                        else next_state = a;
                end
        b:
                begin
                        if(x) next_state = c;
                        else next_state = a;
                end
        c:
                begin
                        if(x) next_state=c;
                        else next_state =d;
                end
        d:
                begin
                        if(x) next_state =b;
                        else next_state =a;
                end
        endcase
always @(state)
```

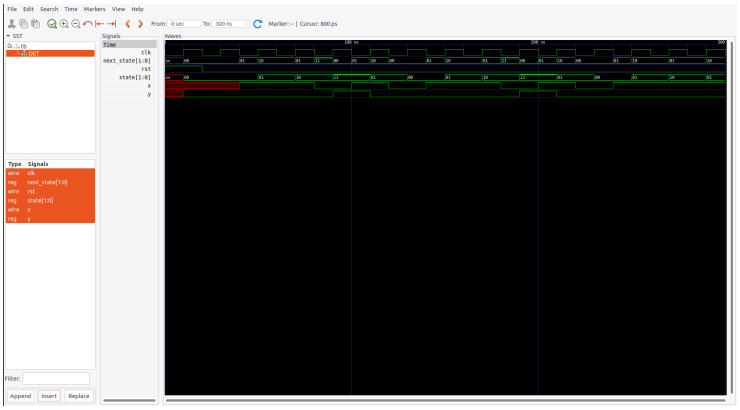
```
        case(state)
                a: y=0;
                b: y=0;
                c: y=0;
                d: y=1;
        endcase
endmodule
```

14.3 Testbench

```
`timescale 1ns / 1ps
`include "seq_det_moore.v"

module tb();
reg x,clk,rst;
wire y;
seq_det_moore DET(clk,rst,x,y);

always
        begin
                clk<=0;
                #10;
                clk<=1;
                #10;
        end
initial
        begin
                $dumpfile("moore.vcd");
                $dumpvars;
                #300 $finish;
        end
initial
        begin
                rst<=1;
                #20 rst <= 0;
                #20 x<=1;
                #20 x<=1;
                #20 x<=0;
                #20 x<=1;
                #20 x<=0;
                #20 x<=1;
                #20 x<=1;
                #20 x<=1;
                #20 x<=0;
```

```
                #20  x<=1;
                #20  x<=1;
                #20  x<=0;
            end
endmodule
```

14.4 GTKWave Output



14.5 Yosys Synthesis Verilog Code

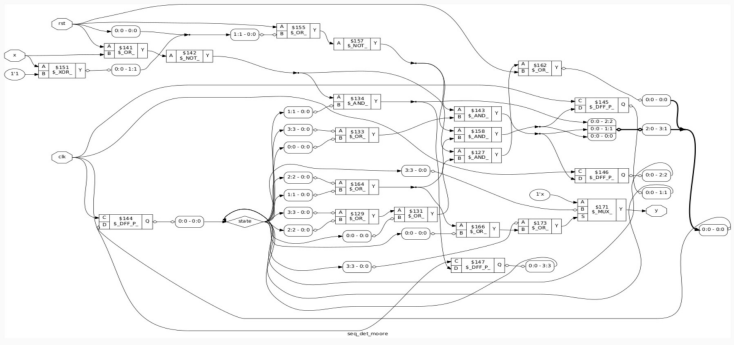
```
/* Generated by Yosys 0.9 (git sha1 1979e0b) */

(* top =  1  *)
(* src = "moore.v:1" *)
module seq_det_moore(clk, rst, x, y);
  wire _00_;
  wire _01_;
  wire _02_;
  wire _03_;
  wire _04_;
  wire _05_;
  wire _06_;
  wire _07_;
  wire _08_;
  wire _09_;
  (* src = "moore.v:1" *)
  wire _10_;
  wire _11_;
  wire _12_;
  wire _13_;
```

```
wire _14_;
(* src = "moore.v:1" *)
wire _15_;
(* src = "moore.v:1" *)
wire _16_;
wire _17_;
wire _18_;
wire _19_;
wire _20_;
wire _21_;
wire _22_;
wire _23_;
wire [3:0] _24_;
wire _25_;
wire _26_;
wire _27_;
wire _28_;
wire _29_;
wire [1:0] _30_;
wire _31_;
(* src = "moore.v:43|moore.v:39|<techmap.v>:445" *)
wire _32_;
(* src = "moore.v:1" *)
input clk;
(* src = "moore.v:1" *)
input rst;
(* onehot = 32'd1 *)
wire [3:0] state;
(* src = "moore.v:1" *)
input x;
(* src = "moore.v:1" *)
output y;
INVX1 _33_ (
  .A(_10_),
  .Y(_04_)
);
INVX1 _34_ (
  .A(_12_),
  .Y(_05_)
);
INVX1 _35_ (
  .A(_15_),
  .Y(_06_)
);
NOR2X1 _36_ (
  .A(_11_),
```

```
    .B(_14_),
    .Y(_07_)
);
NOR2X1 _37_ (
    .A(_10_),
    .B(_13_),
    .Y(_08_)
);
AOI22X1 _38_ (
    .A(_04_),
    .B(_15_),
    .C(_07_),
    .D(_08_),
    .Y(_03_)
);
NOR3X1 _39_ (
    .A(_10_),
    .B(_05_),
    .C(_15_),
    .Y(_02_)
);
NOR3X1 _40_ (
    .A(_10_),
    .B(_06_),
    .C(_07_),
    .Y(_01_)
);
NOR2X1 _41_ (
    .A(_13_),
    .B(_12_),
    .Y(_09_)
);
NOR3X1 _42_ (
    .A(_10_),
    .B(_06_),
    .C(_09_),
    .Y(_00_)
);
BUFX2 _43_ (
    .A(_14_),
    .Y(_16_)
);
DFFPOSX1 _44_ (
    .CLK(clk),
    .D(_24_[0]),
    .Q(state[0])
```

```
  );
  DFFPOSX1 _45_ (
    .CLK(clk),
    .D(_21_),
    .Q(state[1])
  );
  DFFPOSX1 _46_ (
    .CLK(clk),
    .D(_22_),
    .Q(state[2])
  );
  DFFPOSX1 _47_ (
    .CLK(clk),
    .D(_23_),
    .Q(state[3])
  );
  assign _30_[0] = rst;
  assign _24_[3:1] = { _23_, _22_, _21_ };
  assign _11_ = state[0];
  assign _14_ = state[3];
  assign _10_ = rst;
  assign _24_[0] = _03_;
  assign _13_ = state[2];
  assign _12_ = state[1];
  assign _15_ = x;
  assign y = _16_;
  assign _23_ = _02_;
  assign _22_ = _01_;
  assign _21_ = _00_;
endmodule
```

14.6 Yosys Synthesised Design



15 Processor Design

15.1 Theory

Designing a processor in Verilog involves creating a central processing unit (CPU) at a hardware description level. It encompasses the integration of various components like ALUs, registers, and control units.

15.2 Verilog Code

```
module processor(input clk, rst, inout [7:0] data,
output r, w, output [7:0] PC);

reg [7:0] IR, R0, R1, R2, PC;
reg [7:0] tmp_data;
initial PC = 8'h00;
initial R0 = 8'h20;
initial R1 = 8'h15;
initial R2 = 8'h00;
initial IR = 8'h00;
reg [2:0] state, next_state;
reg r,w;
initial r = 0;
initial w = 0;
parameter
fetch = 3'b000, decode = 3'b001, execute_add = 3'b010,
execute_sub = 3'b011, store = 3'b100;

always @(posedge clk)
        begin
                if(rst) state = fetch;
                else state = next_state;
        end

always @(state)

        case(state)
        fetch:
                begin
                        w <= 0;
                        r <= 1;
                        next_state = decode;
                end
        decode:
                begin
```

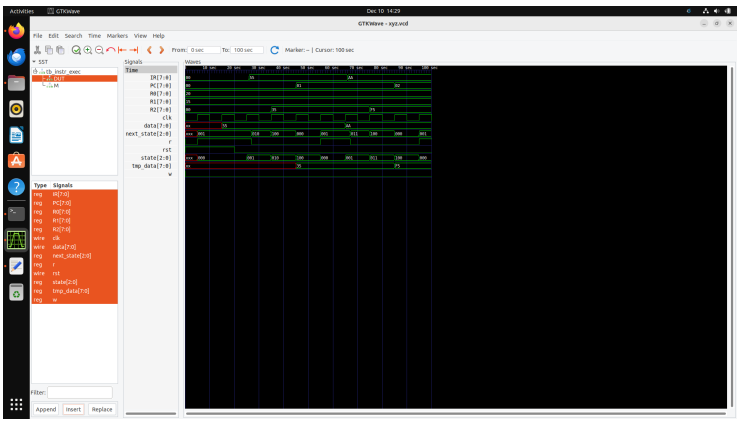
```
                        #1
                        IR <= data;
                        #1
                        if(IR == 8'h55) next_state = execute_add;
                        else if(IR == 8'hAA) next_state = execute_sub;
                        else next_state = decode;
                        r <= 0;
                end
        execute_add:
                begin
                        R2 <= R1 + R0;
                        next_state = store;
                end
        execute_sub:
                begin
                        R2 <= R1 - R0;
                        next_state = store;
                end
        store:
                begin
                        tmp_data <= R2;
                        w <= 0;
            instr gets corrupt
                        next_state = fetch;
                        PC <= PC + 1;
                end
        endcase
assign data = !r & w ? tmp_data : 'hz;
endmodule
```

15.3 Testbench

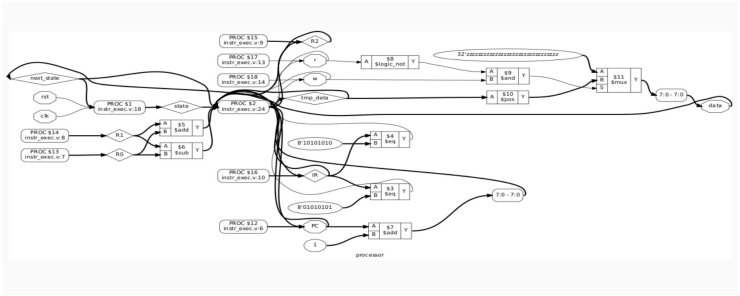
```
`timescale 1ns / 1ps
`include "instr_exec.v"

module memory(input clk,r, w, input [7:0] PC, inout [7:0] data);
reg [7:0] mem[2:0];
reg [7:0] tmp_data;
initial mem[0] = 8'h55;
initial mem[1] = 8'hAA;
initial mem[2] = 8'h00;
  always @ (posedge clk) begin
    if (w)
      mem[PC] <= data;
  end
```

```
  always @ (posedge clk) begin
    if (r)
        tmp_data <= mem[PC];
  end
  assign data = tmp_data;
endmodule


module tb_instr_exec(input CLK, RST_N);

reg clk, rst;
wire r,w;
wire [7:0] PC;
wire [7:0] data;
memory M(clk,r,w,PC,data);
processor DUT(clk,rst,data,r,w,PC);

always
        begin
                clk <= 0;
                #5;
                clk <= 1;
                #5;
        end

initial
        begin
                $dumpfile("xyz.vcd");
                $dumpvars;
                #100 $finish;
        end

initial
        begin
            rst <= 1;
            #20 rst <= 0;

        end

endmodule
```

15.4 GTKWave Output



15.5 Yosys Synthesised Design



15.6 Q flow

15.6.1 Yosys

Role: Yosys is used for RTL synthesis. Description: Yosys is an open-source synthesis tool that translates RTL (Register-Transfer Level) descriptions written in hardware description languages (HDLs) like Verilog into gate-level netlists. It is a crucial tool for converting the high-level description of a digital design into a form that can be physically implemented.

15.6.2 Graywolf

Role: Graywolf is used for placement optimization. Description: Graywolf is an open-source placement tool that helps optimize the placement of cells on the chip layout. It takes into account various constraints and objectives, such as minimizing wirelength, meeting timing requirements, and adhering to floorplan specifications.

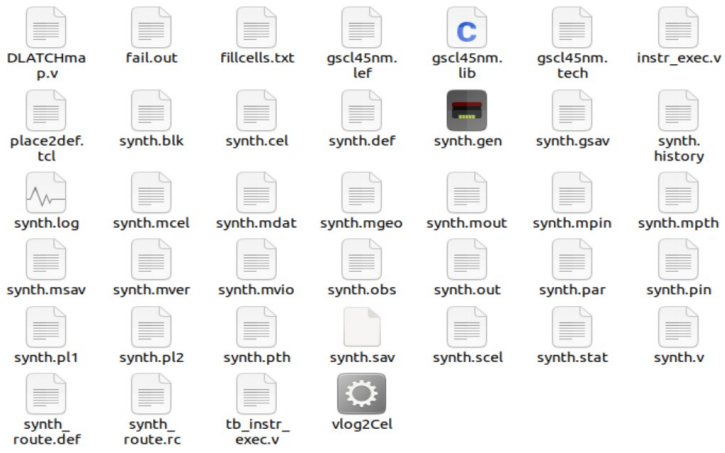
15.6.3 Qrouter

Role: Qrouter is used for automatic routing. Description: Qrouter is an open-source router that is part of the Qflow toolchain. It is used to automatically route the interconnections between the placed cells on the chip, following the defined design rules. It helps optimize the routing to meet area, power, and timing constraints.

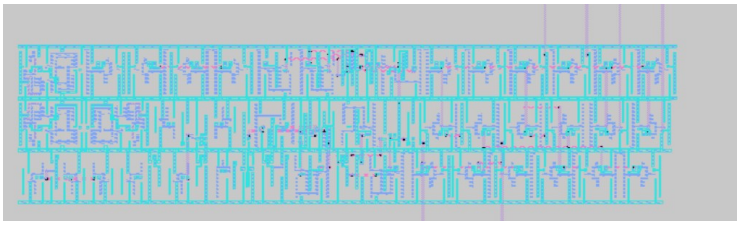
15.6.4 Magic

Role: Magic is used for layout viewing and editing. Description: Magic is an open-source layout editor and viewer. It allows designers to view, edit, and manipulate the physical layout of the chip. Magic is often used in the physical design stage for tasks like floorplanning, placement, and routing.

15.7 Synthesis Files



15.8 Placement Design



16 Conclusion

This report has comprehensively covered the design, simulation, and analysis of a wide range of digital components and systems using Verilog. Beginning with the fundamental logic gates - Inverter, NAND, NOR, XOR - and the D Flip-Flop, the report delved into their operational principles and truth tables, emphasizing their crucial roles in digital circuitry. It further explored the implementation of a Full Adder using various design methodologies, including behavioral, structural, and dataflow approaches, as well as its construction from Half Adders. Additionally, the report presented detailed insights into Memory Blocks and Synchronous RAM, illustrating the intricacies of data storage mechanisms. The concepts of Mealy and Moore State Machines were also elaborated, showcasing their respective logic and applications in sequential circuit design. Finally, the design and simulation of a Processor were discussed, culminating in a synthesis and physical design using tools like Yosys, Graywolf, and QRouter. Each of these segments was accompanied by Verilog code, testbenches, and simulation outputs, providing a holistic understanding of VLSI design and its practical applications in modern electronics.