Objectives

1. **Understanding Digital Circuit Simulation**

- Simulate and analyze Verilog-based digital circuits using Xilinx software to verify their functionality.
- Perform behavioral and timing simulations to understand circuit behavior.

2. **Exploring FPGA Implementation**

- Synthesize Verilog designs using Xilinx Vivado/ISE and generate FPGA bitstreams for hardware testing.
- Analyze the resource utilization and performance of implemented circuits.

3. **Cadence Simulation and Analysis**

- Execute at least five experiments using Cadence software to simulate digital circuits at the transistor level.
- Compare functional and timing results between Xilinx and Cadence simulations.

4. **Comparative Study of Tools**

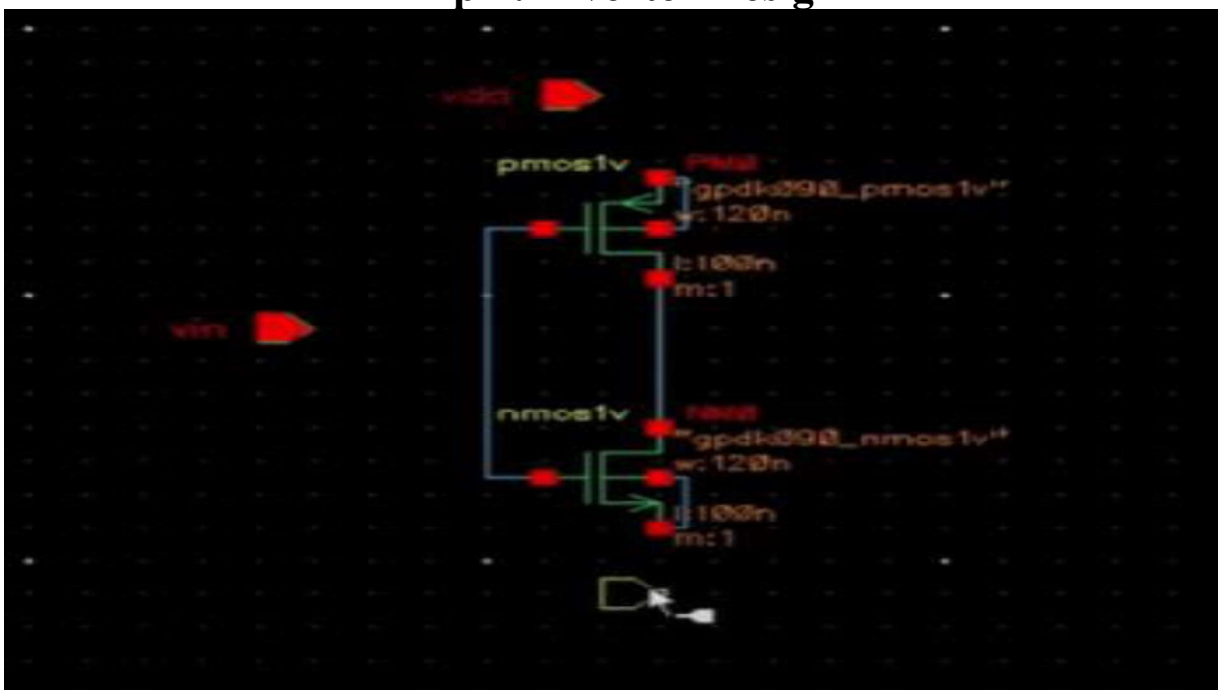
- Evaluate the differences between Xilinx (FPGA-based design) and Cadence (ASIC-focused design) methodologies.
- Understand how synthesis and simulation differ for FPGA vs. ASIC implementations.

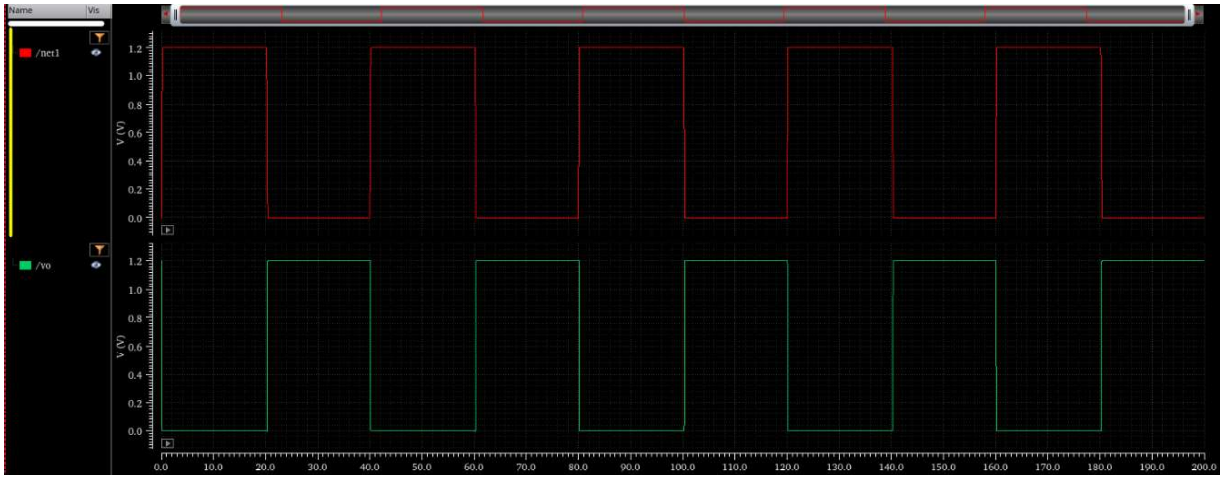
5. **Technical Documentation**

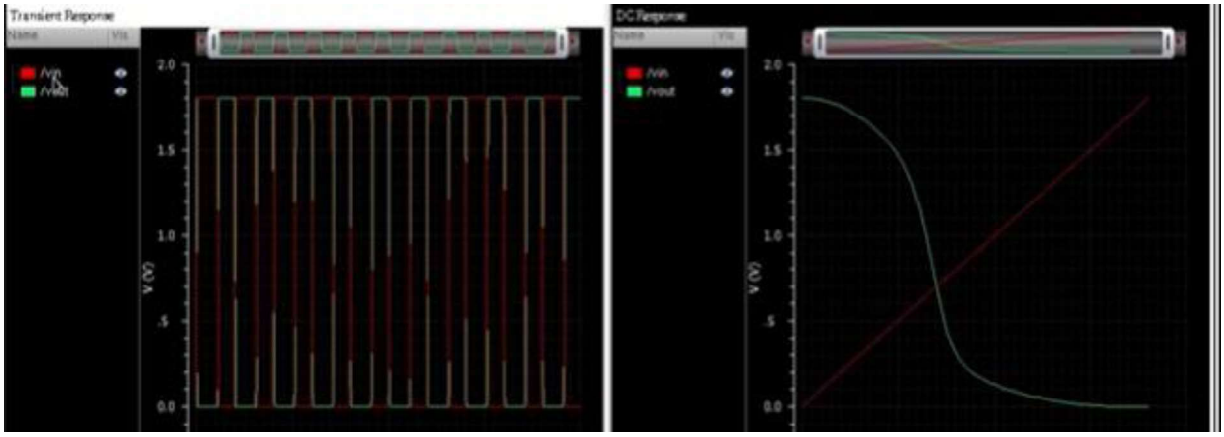
- Prepare a structured report documenting the simulation setup, methodology, results, and observations for both Xilinx and Cadence.
- Include screenshots, waveforms, and analysis of circuit performance.

Cadence

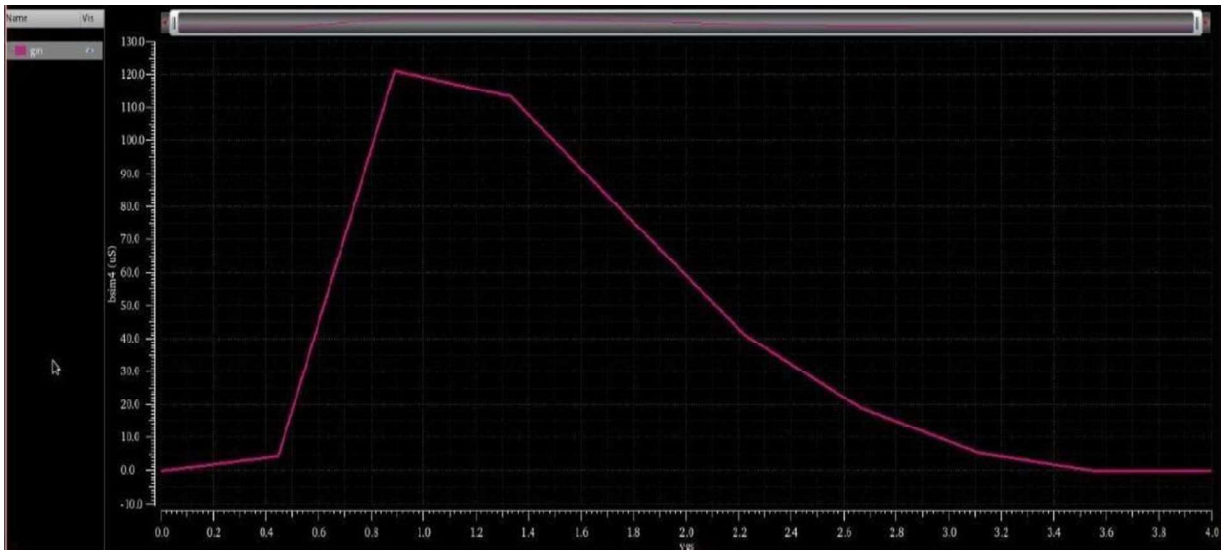
Exp 1: Inverter Design

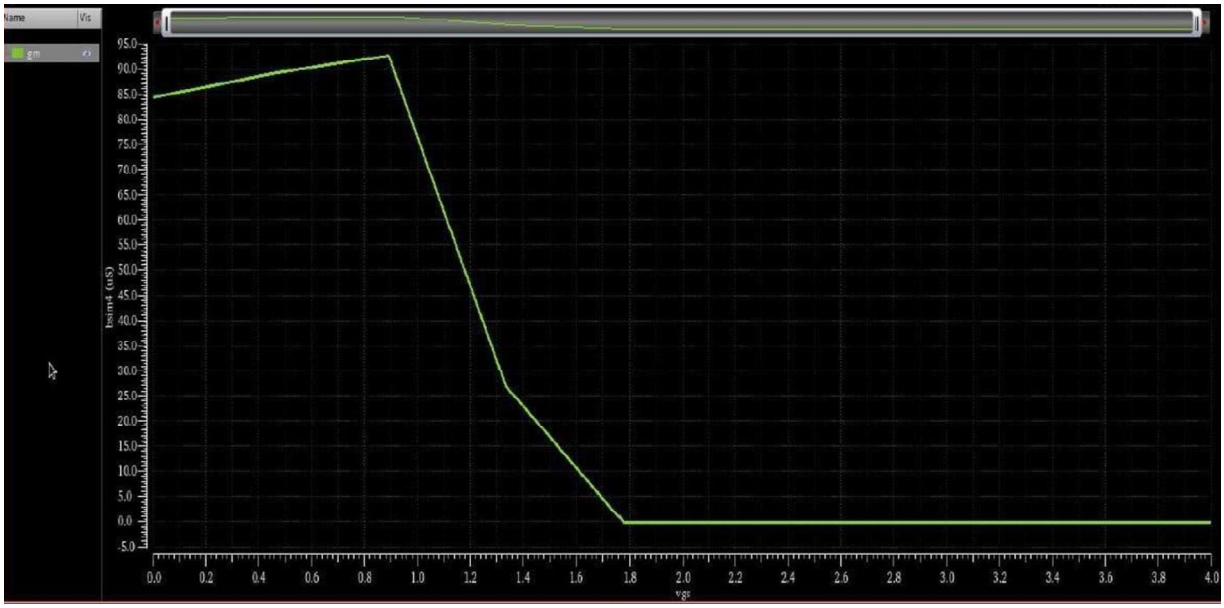


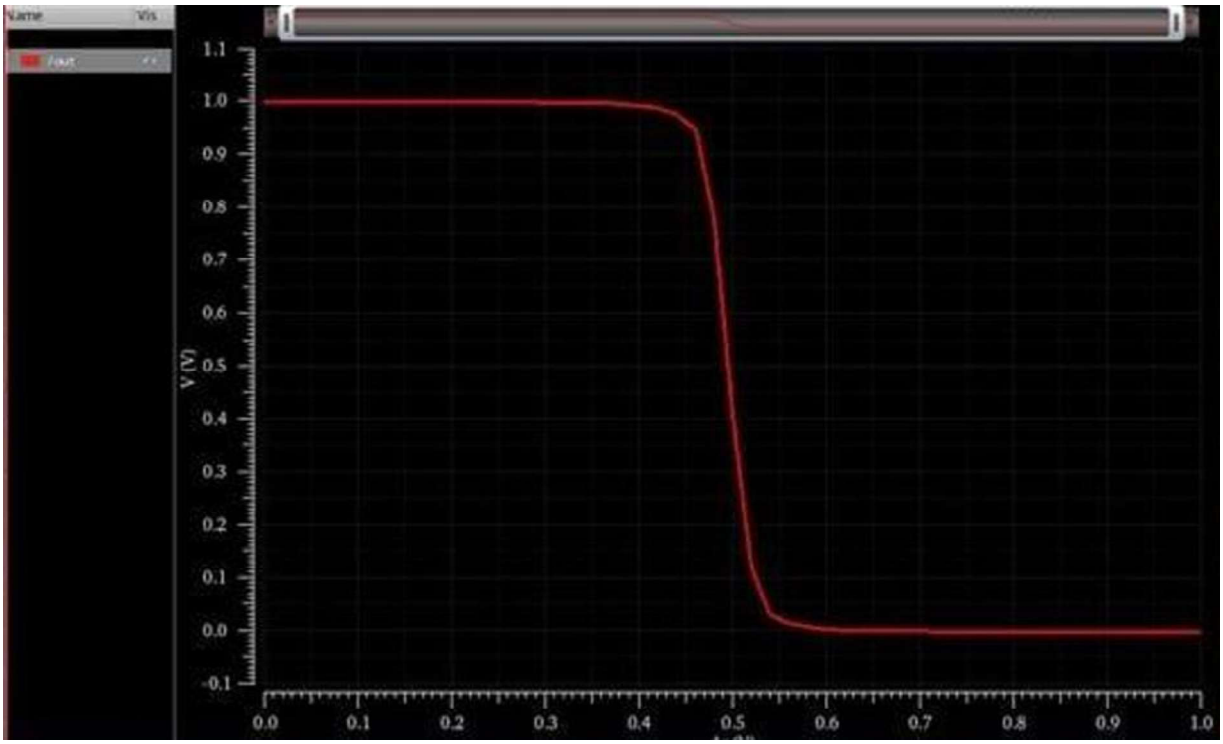




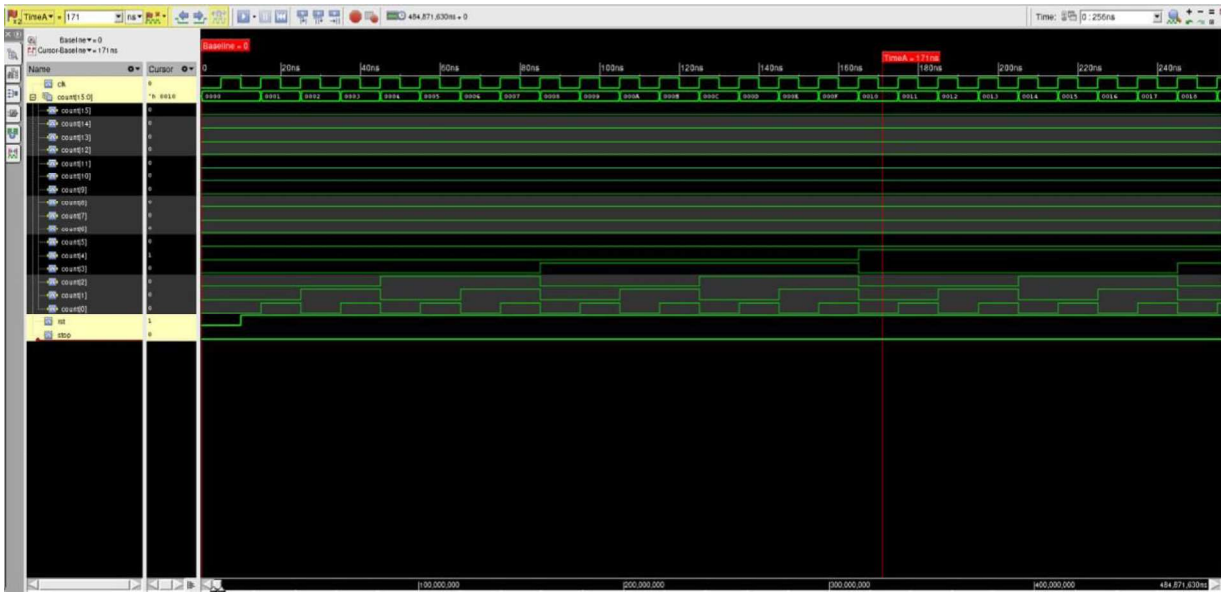
Exp 2:CMOS INVERTER CHARACTERISTCS

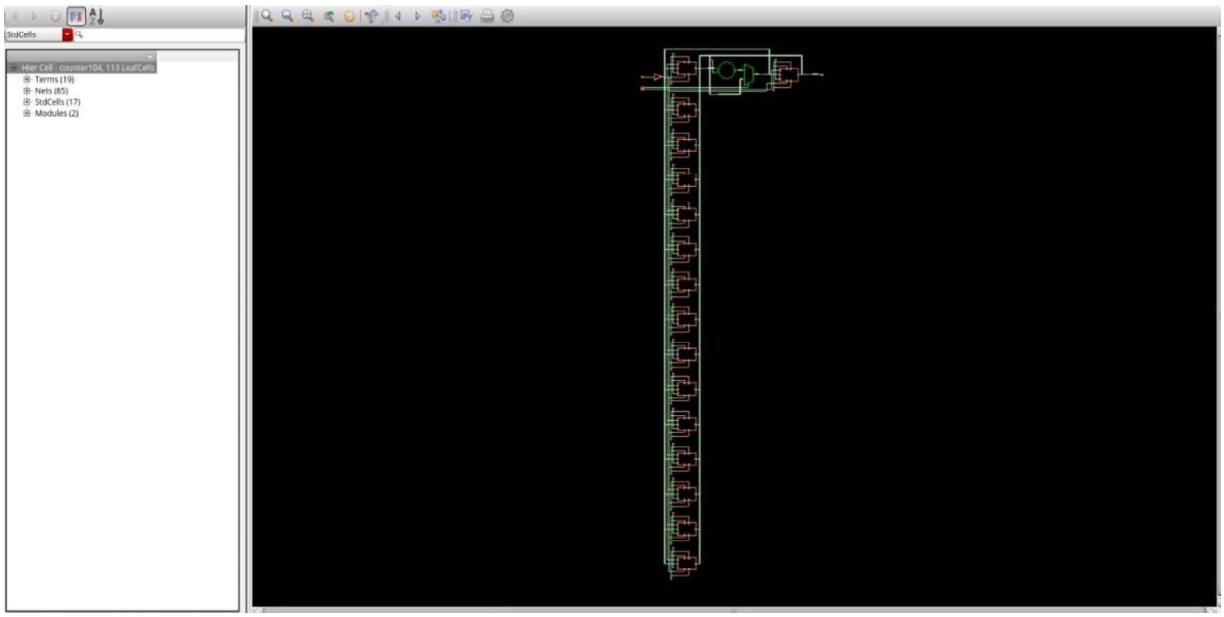




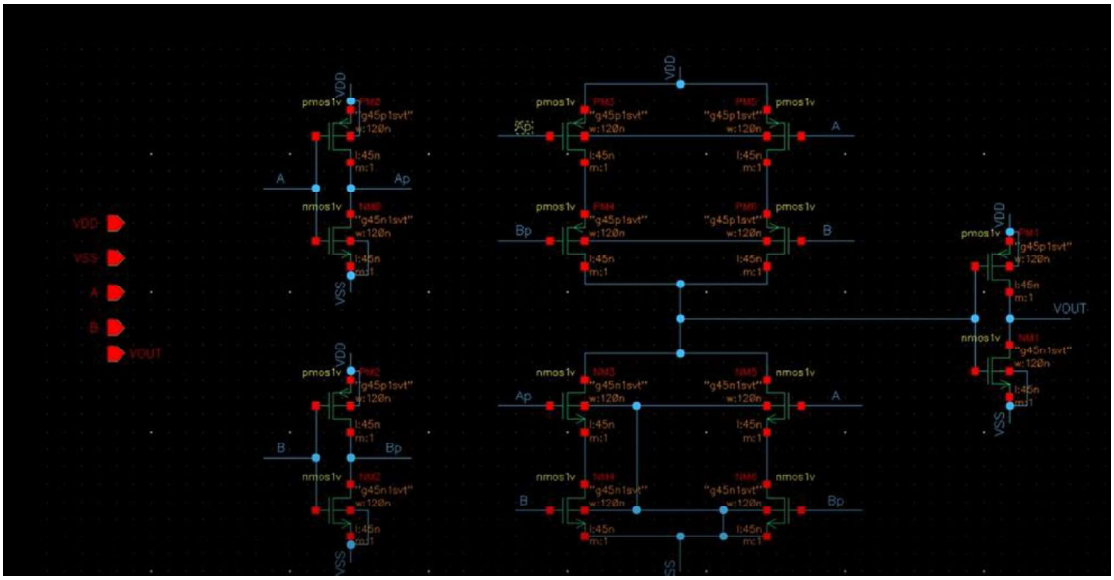

Name
Vin
Vout
1.1
1.0
0.9
0.8
0.7
0.6
0.5
0.4
0.3
0.2
0.1
0.0
-0.1
V (V)
0.0
0.1
0.2
0.3
0.4
0.5
0.6
0.7
0.8
0.9
1.0

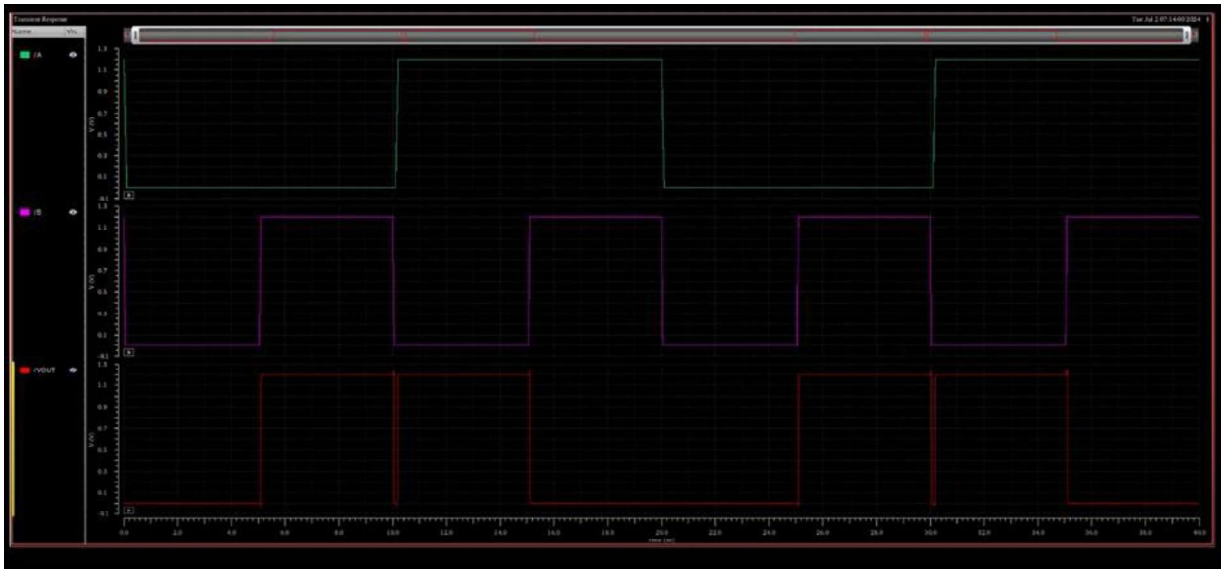
8-bit Counter



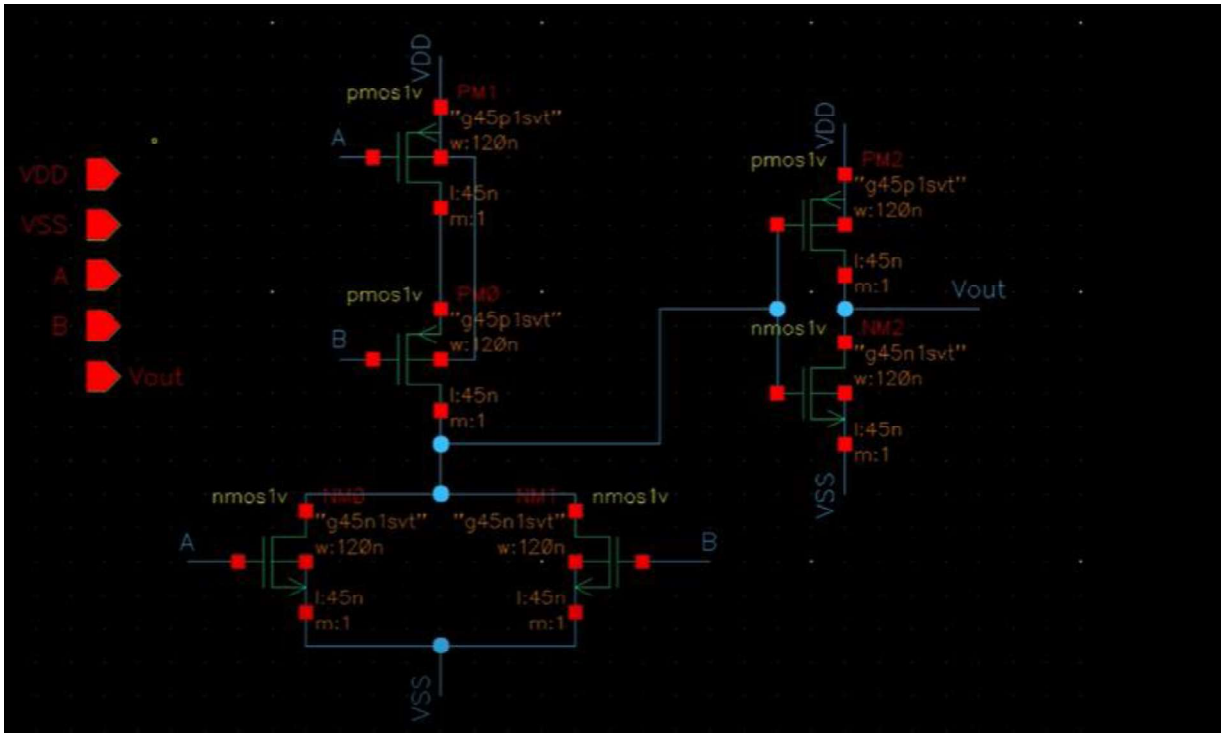


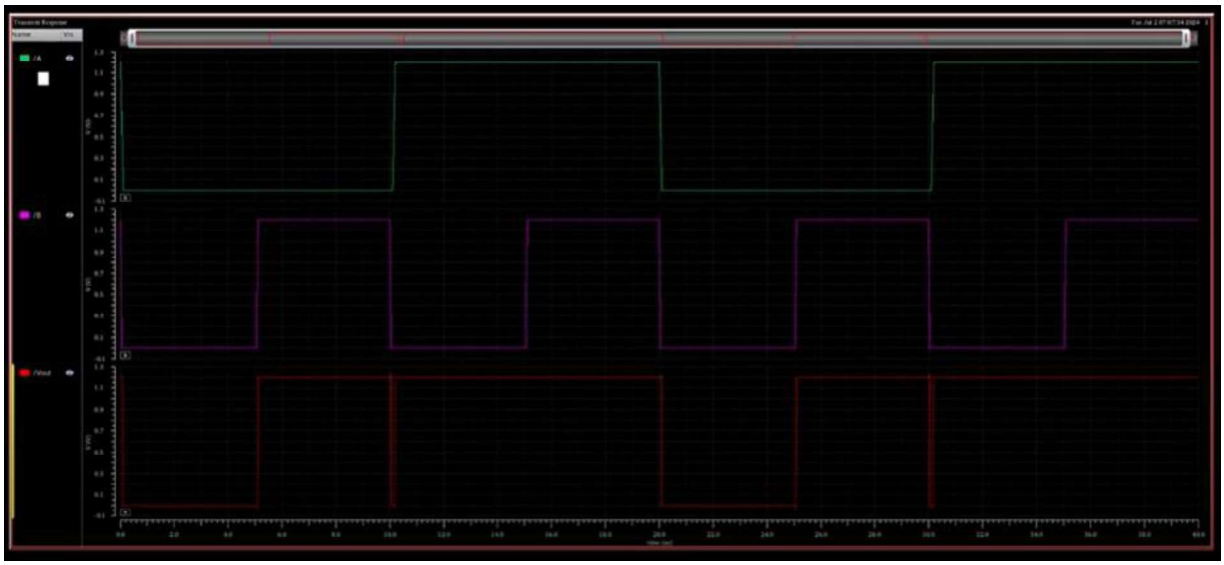
XOR gate



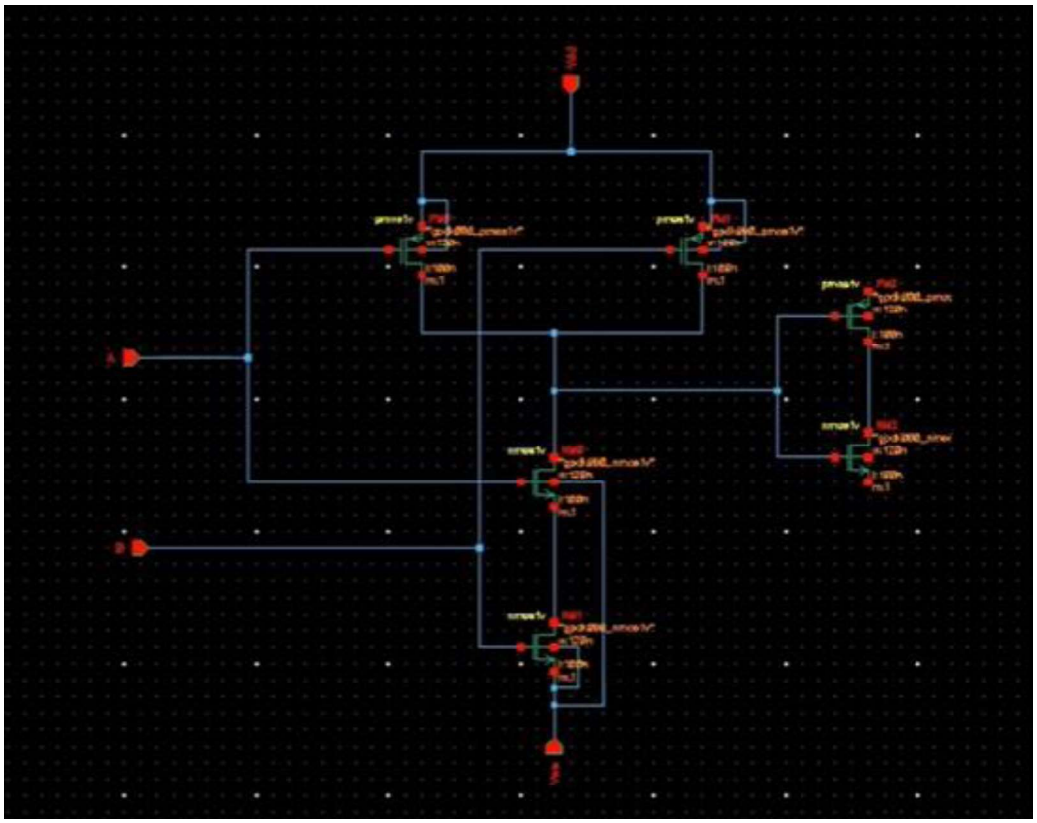


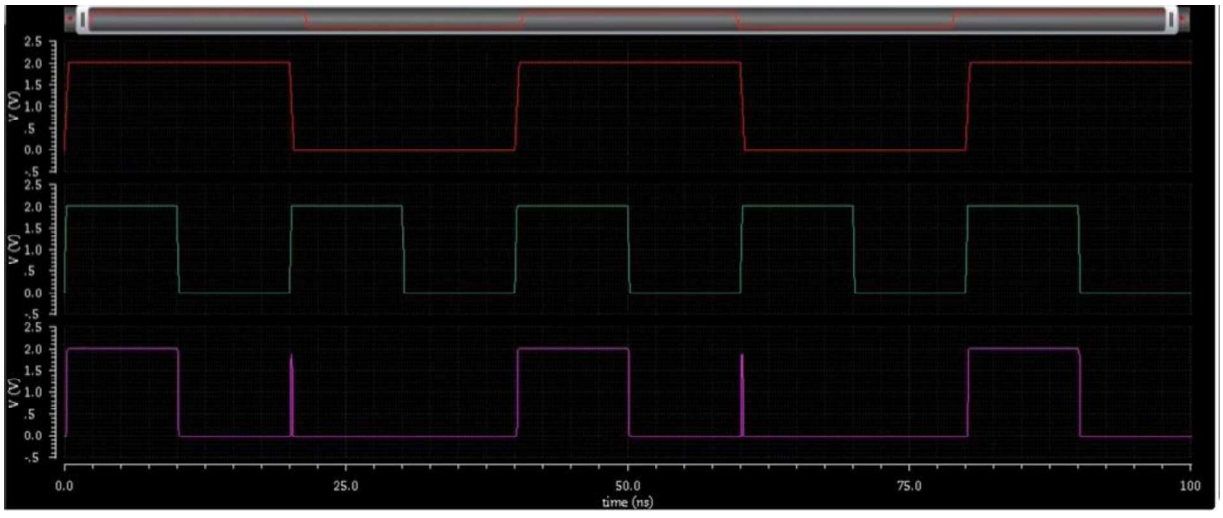
OR gate


VDD
VSS
A
B
Vout
pmos1v
PM1
"g45p1svt"
w:120n
l:45n
m:1
PM0
PM2
nmos1v
NM0
NM1
NM2
"g45n1svt"

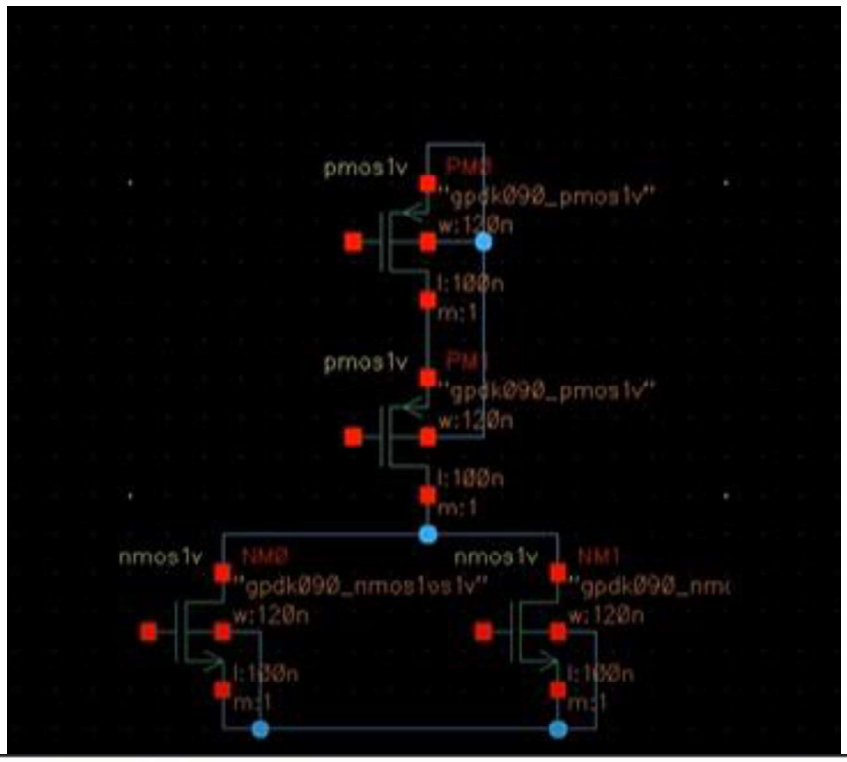


AND gate



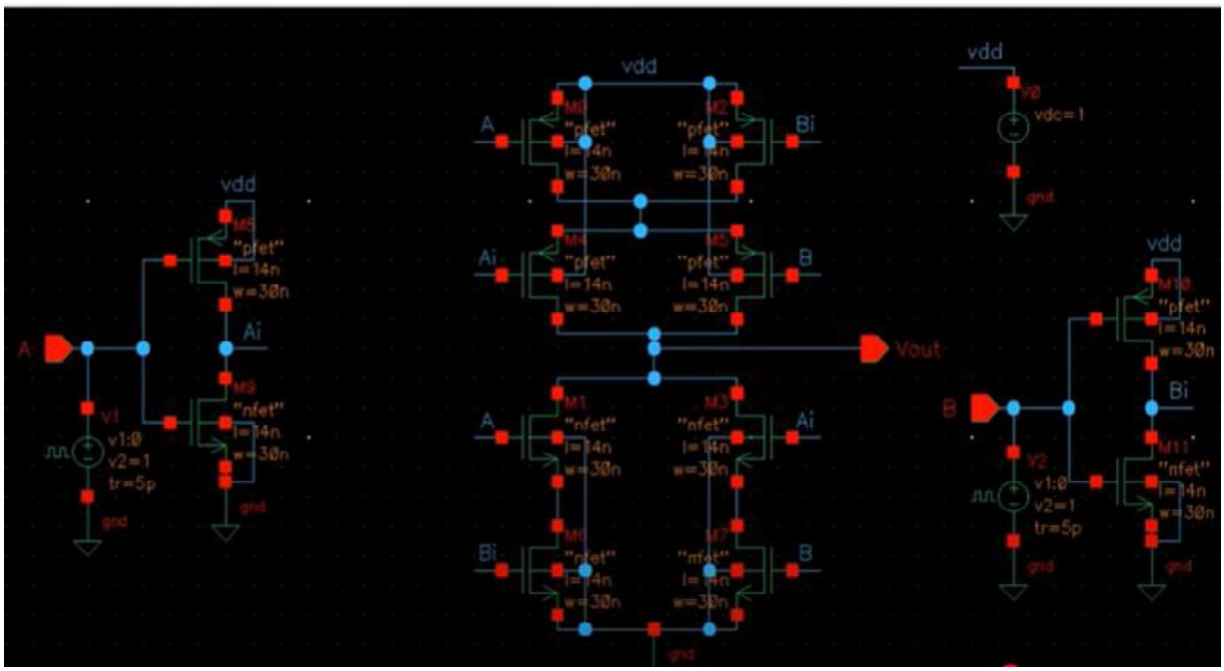


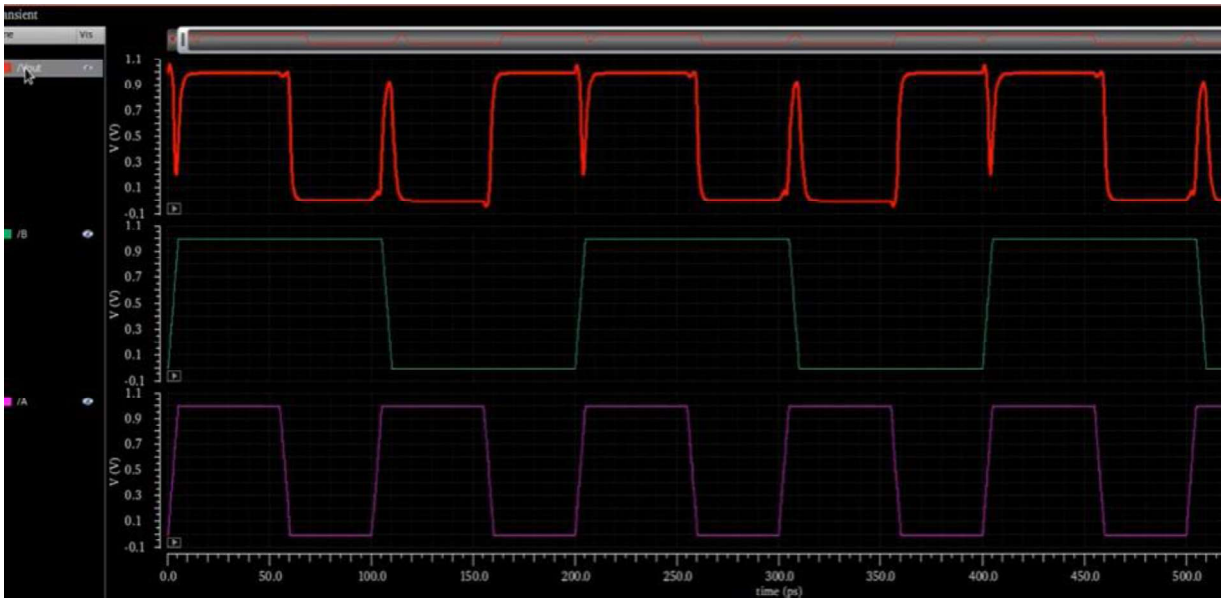
NOR gate


pmos1v
PM0
"gpdk090_pmos1v"
w:120n
l:100n
m:1
pmos1v
PM1
"gpdk090_pmos1v"
w:120n
l:100n
m:1
nmos1v
NM0
"gpdk090_nmos1v"
w:120n
l:100n
m:1
nmos1v
NM1
w:120n
l:100n
m:1

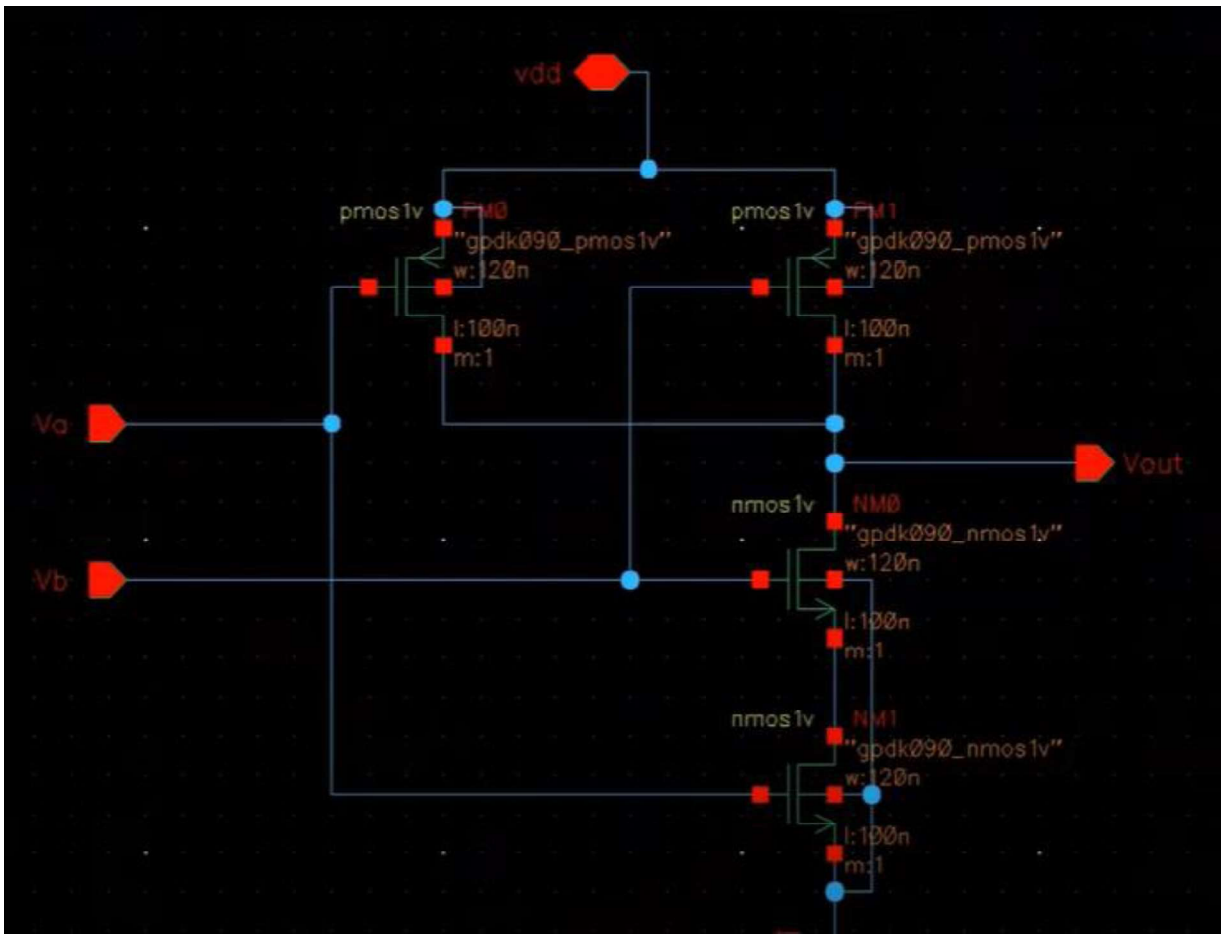

Transient Response
/Vout
/Va
/Vb
V1
1.0V
1.0V
0.0
25.0
50.0
75.0

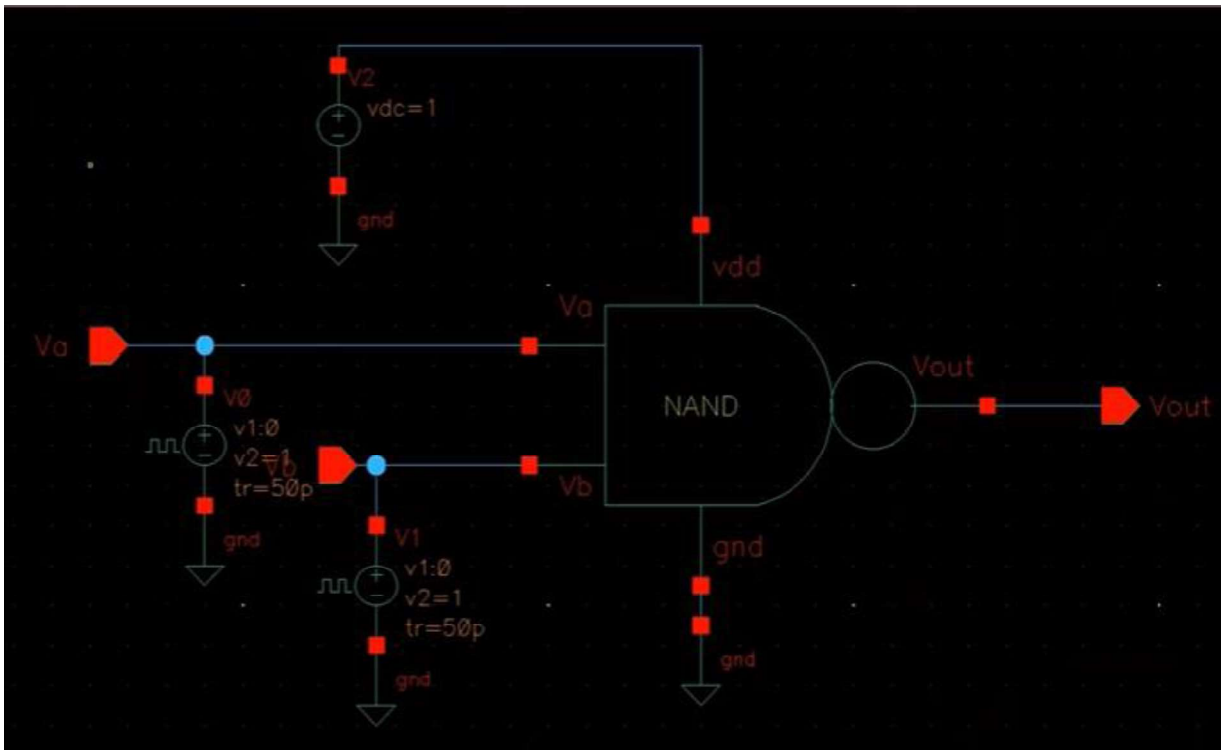
XNOR gate

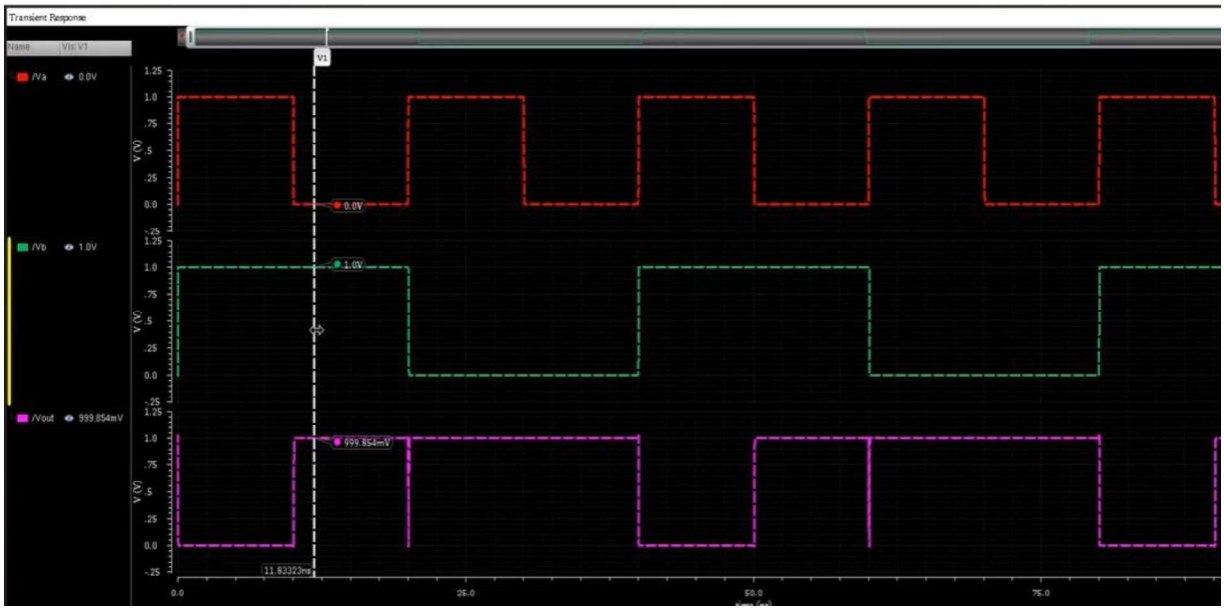

vdd
A
Ai
B
Bi
Vout
"pfet"
"nfet"
l=14n
w=30n
V1
v1:0
v2=1
tr=5p
V2
vdc=1
gnd


/Vout
/B
/A
V (V)
time (ps)
0.0
50.0
100.0
150.0
200.0
250.0
300.0
350.0
400.0
450.0
500.0

NAND gate


vdd
pmos1v
PM0
"gpdk090_pmos1v"
w:120n
l:100n
m:1
pmos1v
PM1
"gpdk090_pmos1v"
w:120n
l:100n
m:1
Va
Vout
nmos1v
NM0
"gpdk090_nmos1v"
w:120n
l:100n
m:1
Vb
nmos1v
NM1
"gpdk090_nmos1v"
w:120n
l:100n
m:1


V2
vdc=1
gnd
vdd
Va
Va
V0
v1:0
v2=1
tr=50p
Vb
NAND
Vout
Vout
Vb
gnd
V1
v1:0
v2=1
tr=50p
gnd
gnd
gnd



1. AND Gate Design Using Cadence

Description:

An AND gate is a fundamental logic gate that produces a HIGH output only when all inputs are HIGH. It is widely used in arithmetic operations, multiplexers, and control logic.

Implementation:

- Can be constructed using NAND, NOR, XOR, XNOR, and Inverter gates by applying Boolean logic principles.
- In CMOS design, the AND gate is implemented using a combination of PMOS and NMOS transistors to ensure low power consumption and efficient switching.

Summary:

In Cadence Virtuoso, the AND gate is designed using CMOS technology. The schematic design, layout creation, and SPICE simulations are used to analyze parameters such as power consumption, propagation delay, and transistor efficiency for ASIC applications.

2. NAND Gate Design Using Cadence

Description:

A NAND gate outputs LOW only when all inputs are HIGH; otherwise, the output remains HIGH. It is classified as a universal gate because it can be used to implement any other logic function.

Implementation:

- In CMOS design, the NAND gate consists of NMOS transistors in a series configuration and PMOS transistors in parallel.
- Used to design complex logic circuits, including adders, multiplexers, and memory elements.

Summary:

In Cadence Virtuoso, the NAND gate is designed using CMOS logic. The schematic, layout, and simulation processes help optimize its performance in terms of speed, power efficiency, and area.

3. NOR Gate Design Using Cadence

Description:

A NOR gate produces a HIGH output only when all inputs are LOW; otherwise, the output remains LOW. It is also a universal gate, allowing the construction of any logic circuit.

Implementation:

- Constructed using PMOS transistors in parallel and NMOS transistors in series.
- Used in memory circuits, control logic, and arithmetic operations.

Summary:

In Cadence Virtuoso, the NOR gate is implemented using CMOS technology. The schematic and layout verification ensure optimized performance with minimal power consumption and delay.

4. XOR Gate Design Using Cadence

Description:

An XOR gate produces a HIGH output when an odd number of inputs are HIGH. It plays a crucial role in arithmetic circuits, parity generators, and data comparison systems.

Implementation:

- Designed using pass-transistor logic (PTL) or complementary CMOS technology.
- Commonly used in half-adders, full-adders, and error detection systems.

Summary:

In Cadence Virtuoso, the XOR gate is implemented using CMOS logic. The schematic, layout, and simulation steps focus on minimizing power consumption and optimizing the transistor count for efficient circuit design.

5. XNOR Gate Design Using Cadence

Description:

An XNOR gate produces a HIGH output when an even number of inputs are HIGH. It is widely used in equality comparison circuits and arithmetic operations.

Implementation:

- Can be designed using CMOS logic or pass transistor logic (PTL).
- Used in parity checkers, error detection systems, and logic comparators.

Summary:

In Cadence Virtuoso, the XNOR gate is designed to optimize power and area efficiency. The schematic and layout simulations ensure functionality while maintaining low power consumption.

6. Inverter (NOT Gate) Design Using Cadence

Description:

An Inverter (NOT gate) is the simplest logic gate that reverses its input state. It is a fundamental building block in all digital circuits.

Implementation:

- Implemented using a PMOS and NMOS transistor pair.
- Used in buffer circuits, signal conditioning, and logic level conversion.

Summary:

In Cadence Virtuoso, the inverter is designed and simulated to verify switching characteristics, propagation delay, and power efficiency, making it essential for ASIC and FPGA design.

7. 8-Bit Counter Design Using Cadence

Description:

An 8-bit counter is a sequential circuit that counts from 0 to 255 in binary, commonly used in digital systems for counting operations, frequency division, and event tracking.

Implementation:

- Built using a series of flip-flops (D or T) with a common clock signal.
- Includes a reset input to initialize the count.
- Can be classified into:
    - Synchronous Counter: All flip-flops share the same clock, reducing delay.
    - Asynchronous Counter (Ripple Counter): Flip-flops are clocked sequentially, introducing propagation delay.

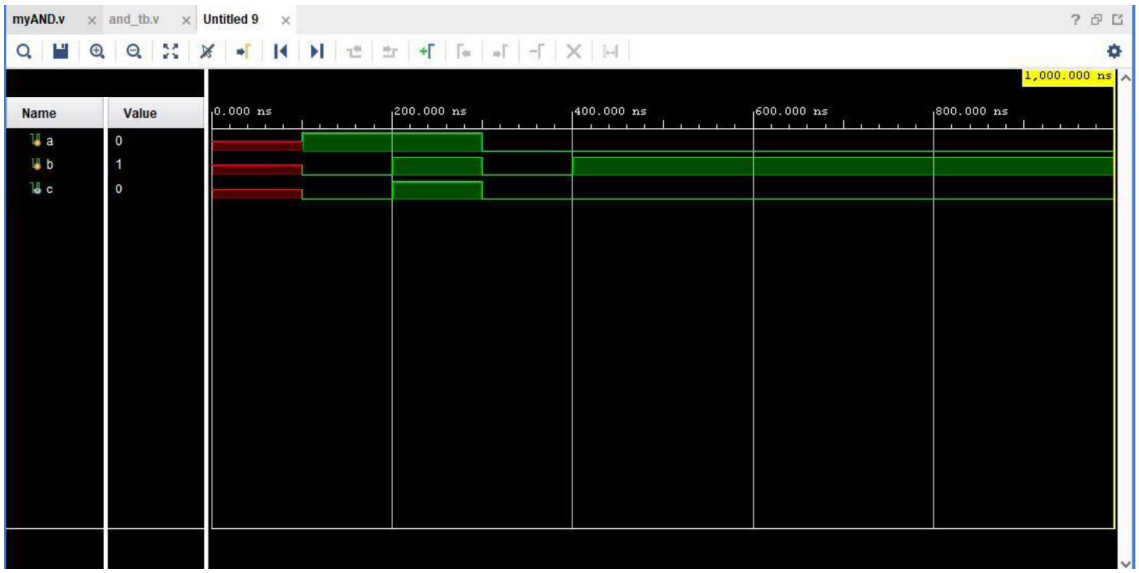
Verilog

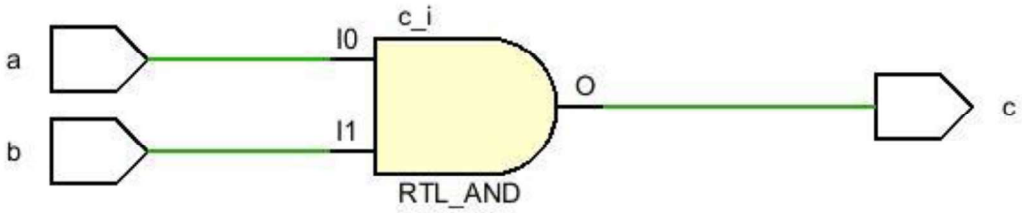
1. AND gate

```
module AND_Gate(
    input A,
    input B,
    output Y
);
    assign Y = A & B;
endmodule
```

testbench:

```
module AND_Gate_tb;
    reg A, B;
    wire Y;

    AND_Gate uut (
        .A(A),
        .B(B),
        .Y(Y)
    );

    initial begin
        $monitor("A = %b, B = %b, Y = %b", A, B, Y);
        A = 0; B = 0; #10;
        A = 0; B = 1; #10;
        A = 1; B = 0; #10;
        A = 1; B = 1; #10;
        $stop;
    end
endmodule
```





Description:

An AND gate is a fundamental logic gate that outputs HIGH (1) only if all its inputs are HIGH (1). It follows the Boolean expression:

$Y = A \cdot B$

where AAA and BBB are inputs, and YYY is the output.

Summary:

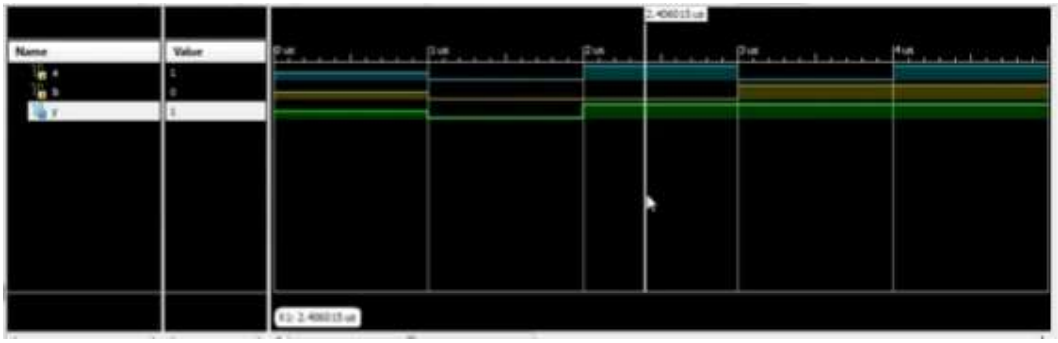
In Xilinx, an AND gate can be implemented using Verilog or VHDL in an FPGA design. It can also be instantiated using built-in logic primitives in Xilinx Vivado or ISE. The design can be tested using behavioral simulation to verify its truth table. When implemented on an FPGA, the AND gate can be mapped to LUTs (Look-Up Tables) within the logic fabric of the device.

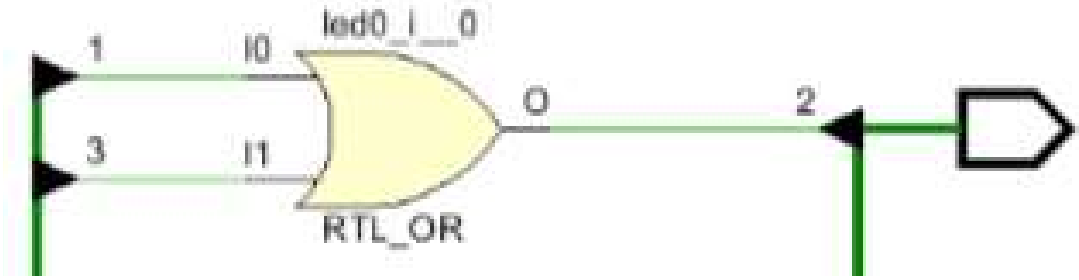
2. OR gate

```
Module or_gate(input a,b; output y;);
Assign y=a|b;
endmodule
```

testbench:

```
module tb_or_gate;
    reg A, B;
    wire Y;

    or_gate uut (
        .A(A),
        .B(B),
        .Y(Y)
    );

    initial begin
        A = 0; B = 0;
        #10;
        A = 0; B = 1;
        #10;
        A = 1; B = 0;
        #10;
        A = 1; B = 1;
        #10;
        $finish;
    end

    initial begin
        $monitor("A = %b, B = %b, Y = %b", A, B, Y);
    end
endmodule
```





Description:

An OR gate is a basic logic gate that outputs HIGH (1) if at least one of its inputs is HIGH (1). It follows the Boolean expression:

Y=A+B

where AAA and BBB are inputs, and YYY is the output.

Summary:

In Xilinx, an OR gate can be implemented using Verilog or VHDL in an FPGA design. It can also be instantiated using built-in logic primitives in Xilinx Vivado or ISE. The design can be tested using behavioral simulation to verify its functionality. When implemented on an FPGA, the OR gate is synthesized using LUTs (Look-Up Tables), optimizing resource usage based on the target FPGA architecture.

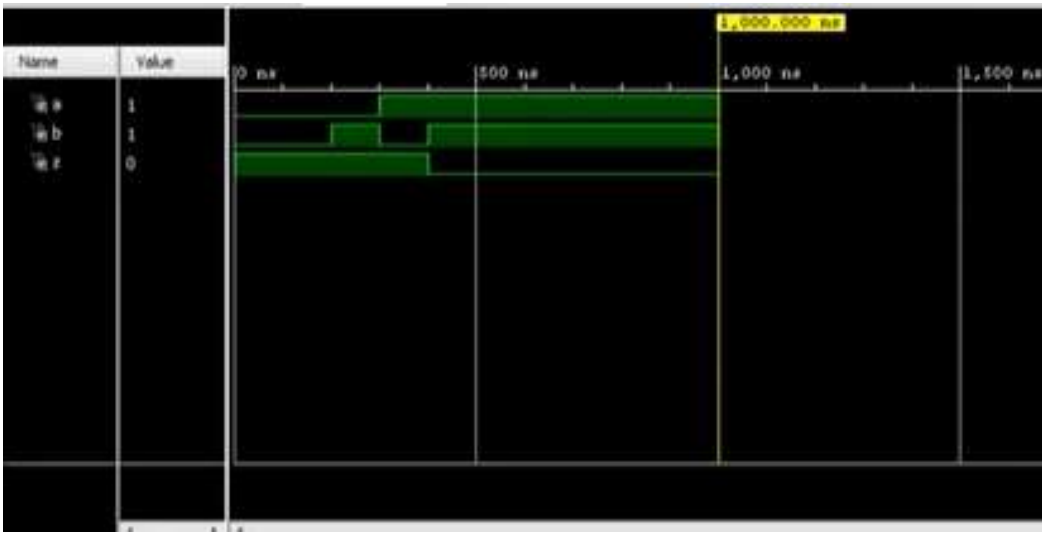
3. NAND gate

```
module nand_gate(
    input a,
    input b,
    output y
);
    assign Y = ~(A & B);
endmodule
```

testbench:

```
module tb_nand_gate;
    reg A, B;
    wire Y;

    nand_gate uut (
        .A(A),
        .B(B),
        .Y(Y)
    );

    initial begin
        A = 0; B = 0;
        #10;
        A = 0; B = 1;
        #10;
        A = 1; B = 0;
        #10;
        A = 1; B = 1;
        #10;
        $finish;
    end

    initial begin
        $monitor("A = %b, B = %b, Y = %b", A, B, Y);
    end
```

endmodule



Description:

A NAND (NOT AND) gate is a universal logic gate that outputs LOW (0) only when all its inputs are HIGH (1). It follows the Boolean expression:

Y=(A·B)’

where AAA and BBB are inputs, and YYY is the output.

Summary:

In Xilinx, a NAND gate can be implemented using Verilog or VHDL in an FPGA design. It can also be instantiated using built-in logic primitives in Xilinx Vivado or ISE. The design can be tested using behavioral simulation to verify its truth table. When implemented on an FPGA, the NAND gate is mapped to LUTs (Look-Up Tables) within the FPGA fabric, utilizing minimal resources due to its fundamental nature.

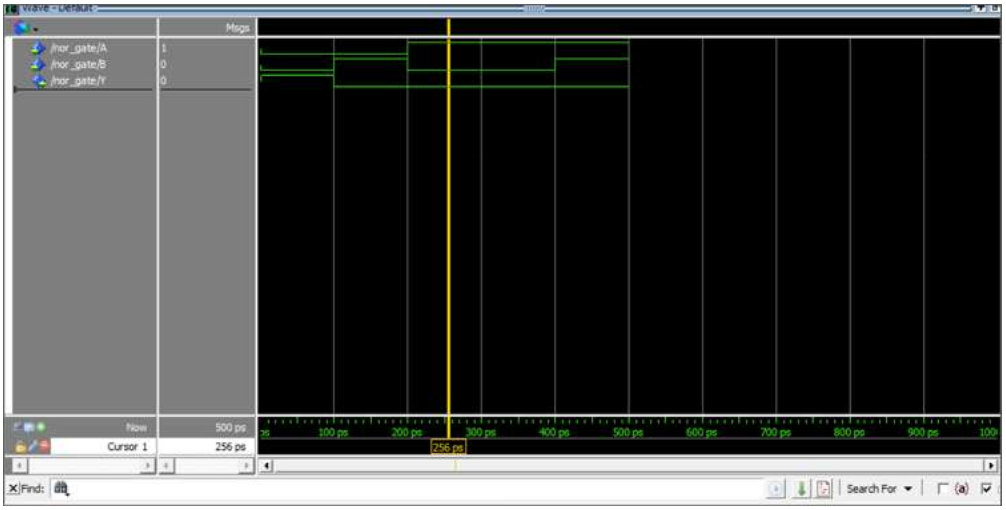
4. NOR gate

Code:

```
module myAND(
   input a,b,
   output c
   );
   assign c = ~(a|b);
endmodule
```

Testbench:

```
module or_tb();
reg a;
reg b;
wire c;
myAND module_u_test(a,b,c);
initial begin

#100;
a=1;b=0;#100;
a=1;b=1;#100;
a=0;b=0;#100;
a=0;b=1;#100;


end

endmodule
```



Description:

A NOR (NOT OR) gate is a fundamental logic gate that outputs HIGH (1) only when all its inputs are LOW (0). It follows the Boolean expression:

Y= A+B

where AAA and BBB are inputs, and YYY is the output.

Summary:

In Xilinx, a NOR gate can be implemented using Verilog or VHDL in an FPGA design. It can also be instantiated using built-in logic primitives in Xilinx Vivado or ISE. The design can be tested using behavioral simulation to verify its functionality. When implemented on an FPGA, the NOR gate is synthesized using LUTs (Look-Up Tables), optimizing resource utilization based on the target FPGA architecture.

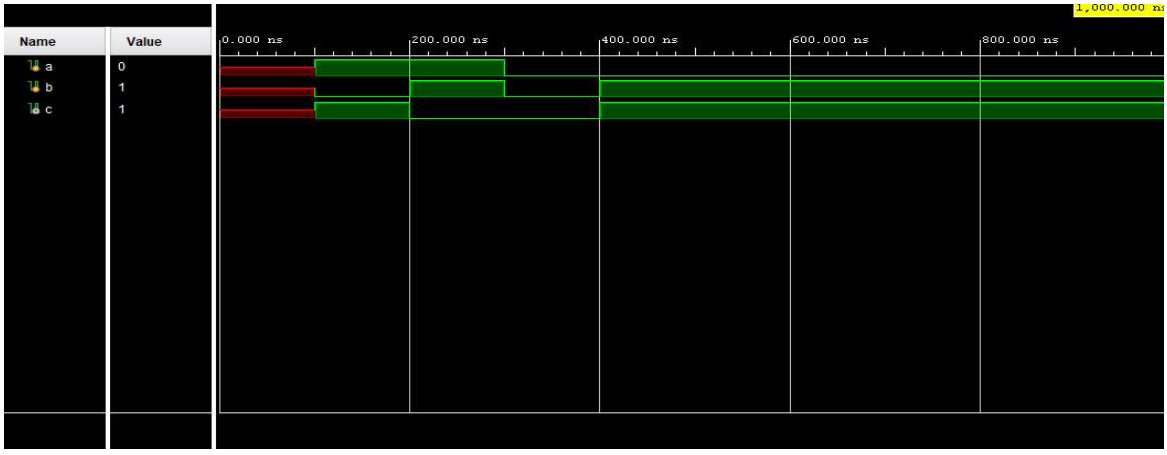
5. XOR gate

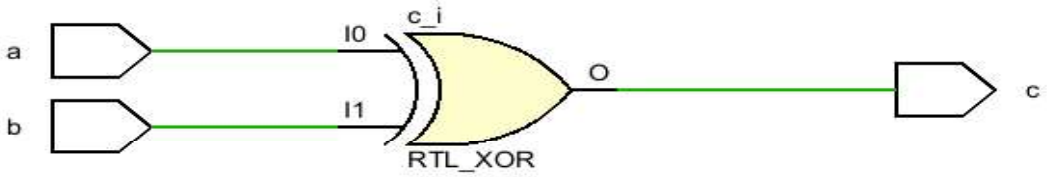
Code:

```
module myAND(
   input a,b,
   output c
   );
   assign c = a^b;
endmodule
```

Testbench:

```
module or_tb();
reg a;
reg b;
wire c;
myAND module_u_test(a,b,c);
initial begin

#100;
a=1;b=0;#100;
a=1;b=1;#100;
a=0;b=0;#100;
a=0;b=1;#100;


end

endmodule
```





Adders:

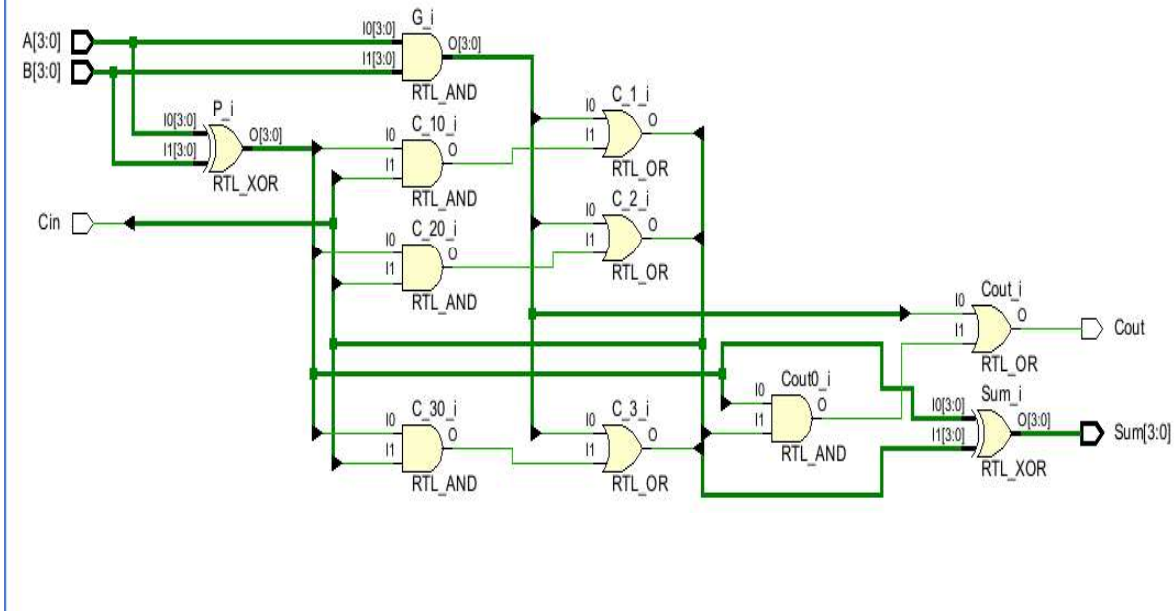
1. Carry Lookahead adder:

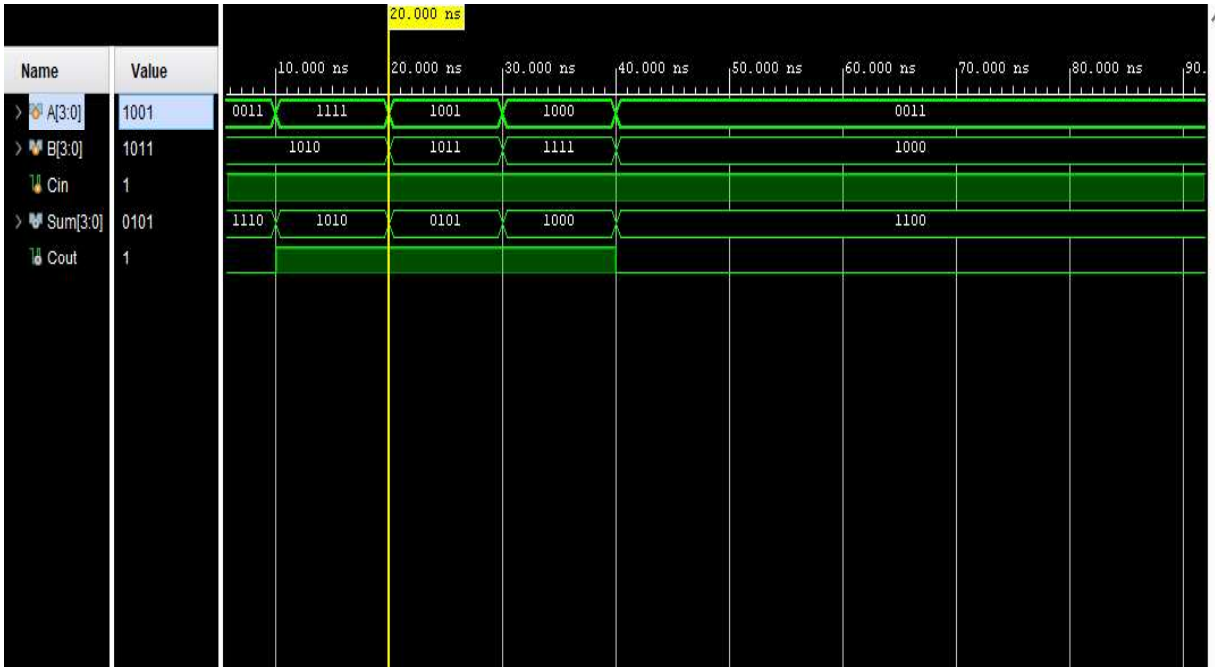
Code:

```
module carry_lookahead(
   input [3:0]A,B,
   input Cin,
   output [3:0]Sum,
   output Cout
   );
   wire [3:0]G,P;
   wire [3:0]C;
   assign G=A&B;
   assign P=A^B;
   assign C[0]=Cin;
   assign C[1]=(G[0]|P[0]&C[0]);
   assign C[2]=(G[1]|P[1]&C[1]);
   assign C[3]=(G[2]|P[2]&C[2]);
   assign Cout=(G[3]|P[3]&C[3]);
   assign Sum=P^C;

endmodule
```

Testbench:

```
module CLA_tb;
reg [3:0]A,B;
reg Cin;
wire [3:0]Sum;
wire Cout;
carry_lookahead uut(
.A(A),
.B(B),
.Cin(Cin),
.Sum(Sum),
.Cout(Cout)
);
initial begin
A=4'b0011;B=4'b1010;Cin=1;#10;
A=4'b1111;B=4'b1010;Cin=1;#10;
A=4'b1001;B=4'b1011;Cin=1;#10;
A=4'b1000;B=4'b1111;Cin=1;#10;
A=4'b0011;B=4'b1000;Cin=1;#10;

end
endmodule
```





Description:

A Carry Look-Ahead Adder (CLA) is a high-speed adder designed to reduce the propagation delay caused by carry generation in ripple-carry adders. It calculates carry bits in parallel using generate (GGG) and propagate (PPP) logic, following these equations:

- **Propagate:** Pi= Ai⊕Bi
- **Generate:** Gi=Ai·Bi
- **Carry:** Ci+1=Gi+(Pi·Ci)C

Summary:

In Xilinx, a CLA can be implemented using Verilog or VHDL in an FPGA. The architecture consists of logic blocks that compute generate and propagate signals, enabling faster carry computation compared to a ripple-carry adder. It can be synthesized and optimized using Xilinx Vivado or ISE tools. The CLA design improves speed significantly, making it suitable for high-performance arithmetic operations in digital circuits and processors.

2. Carry skip adder

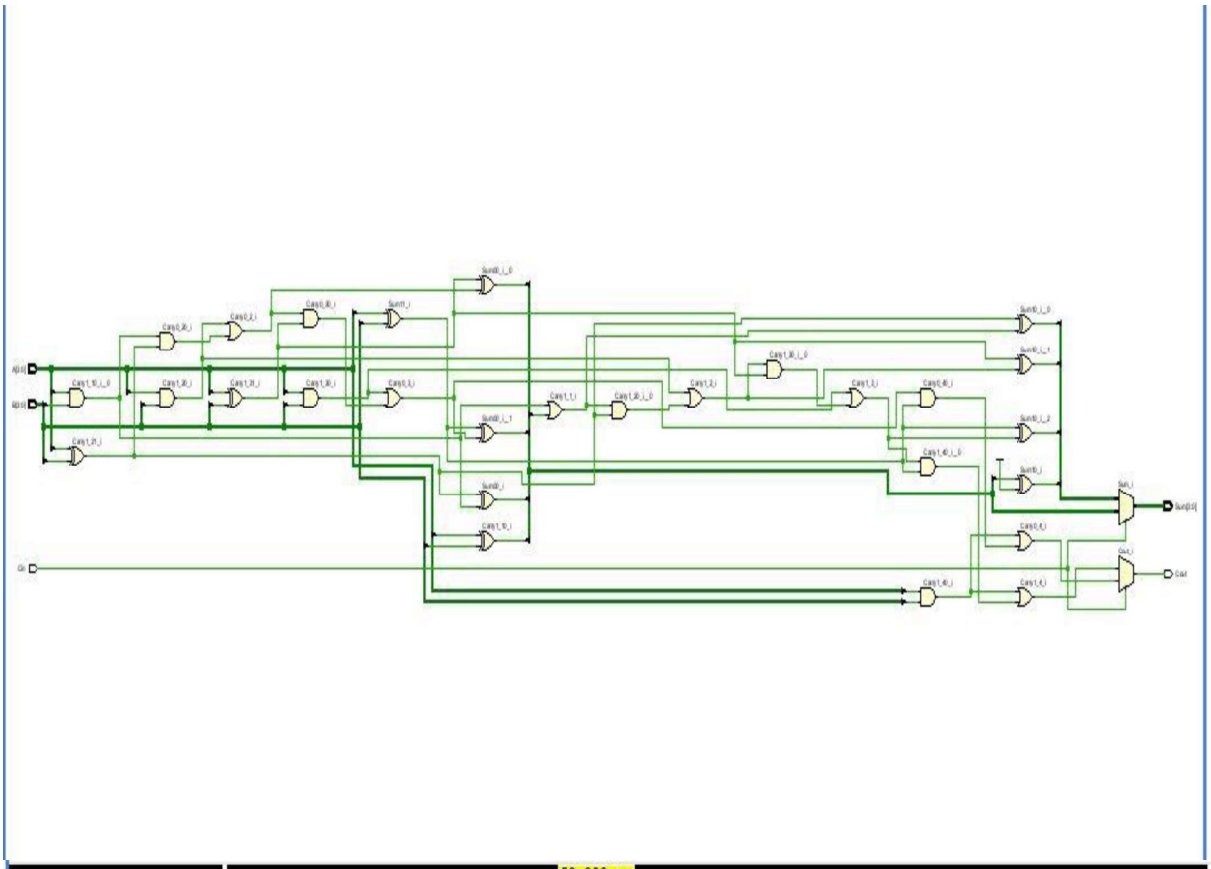
Code:

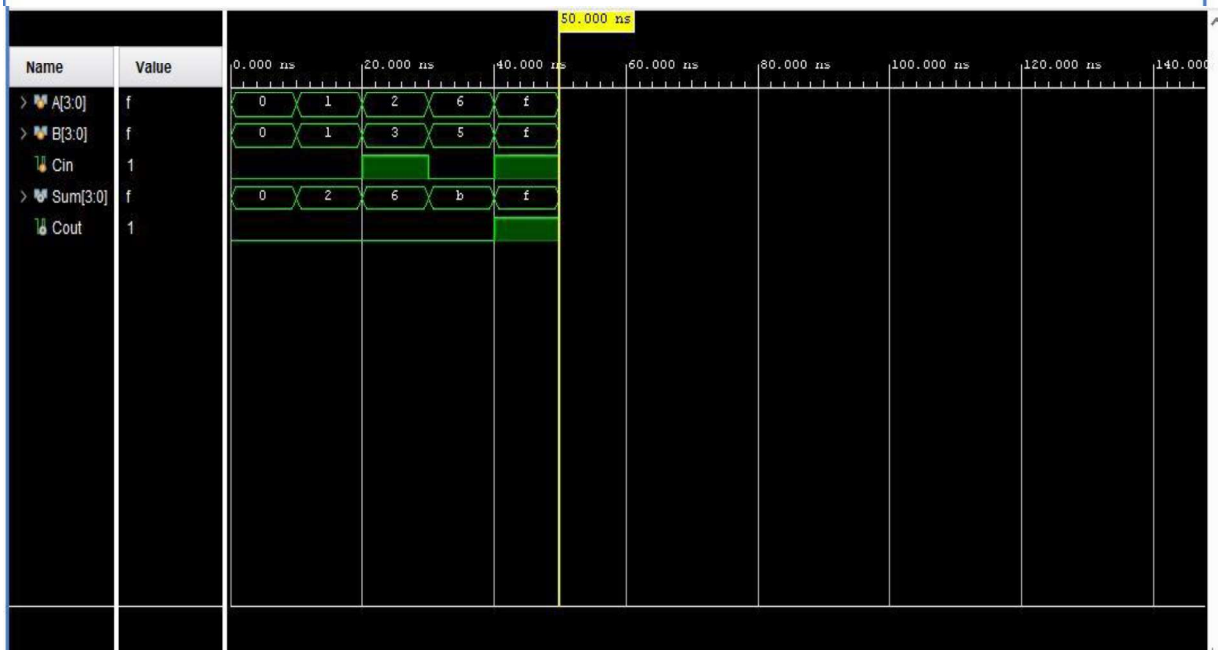
```
module CarrySelectAdder4Bit (
   input  [3:0] A, B,
   input  Cin,
   output [3:0] Sum,
   output Cout
);

   wire [3:0] Sum0, Sum1;
   wire [4:0] Carry0, Carry1;

   assign Carry0[0] = 0;
   assign Carry1[0] = 1;

   genvar i;
   generate
      for (i = 0; i < 4; i = i + 1) begin : bit_slice
         assign Sum0[i] = A[i] ^ B[i] ^ Carry0[i];
         assign Carry0[i+1] = (A[i] & B[i]) | (Carry0[i] & (A[i] ^ B[i]));

         assign Sum1[i] = A[i] ^ B[i] ^ Carry1[i];
         assign Carry1[i+1] = (A[i] & B[i]) | (Carry1[i] & (A[i] ^ B[i]));
      end
   endgenerate

   assign Sum = Cin ? Sum1 : Sum0;
   assign Cout = Cin ? Carry1[4] : Carry0[4];

endmodule
```

Testbench:

```
module CarrySelectAdder4Bit_tb;
   reg [3:0] A, B;
   reg Cin;
   wire [3:0] Sum;
   wire Cout;

   // Instantiate the CarrySelectAdder4Bit module
   CarrySelectAdder4Bit uut (
      .A(A),
      .B(B),
      .Cin(Cin),
      .Sum(Sum),
      .Cout(Cout)
   );

   initial begin
      // Monitor the values
      $monitor("Time=%0t A=%b B=%b Cin=%b | Sum=%b Cout=%b", $time, A, B, Cin, Sum,
Cout);

      // Apply test cases
      A = 4'b0000; B = 4'b0000; Cin = 0; #10;
      A = 4'b0001; B = 4'b0001; Cin = 0; #10;
      A = 4'b0010; B = 4'b0011; Cin = 1; #10;
```

```
        A = 4'b0110; B = 4'b0101; Cin = 0; #10;
        A = 4'b1111; B = 4'b1111; Cin = 1; #10;

        // End simulation
        $finish;
    end
endmodule
```





Description:

A Carry Skip Adder (CSA) is an optimized adder designed to reduce carry propagation delay in large bit-width additions. It divides the adder into blocks, allowing the carry to "skip" over groups of bits when possible, reducing the worst-case delay.

Summary:

In Xilinx, a Carry Skip Adder can be implemented using Verilog or VHDL in an FPGA. It is designed by grouping full adders into blocks and incorporating skip logic to bypass unnecessary carry computations. This reduces propagation delay compared to a ripple-carry adder. Using Xilinx Vivado or ISE, the design can be synthesized and tested for performance improvements. CSA provides a balance between speed and hardware complexity, making it a practical choice for arithmetic circuits in FPGA-based systems.

3. **Carry save adder**

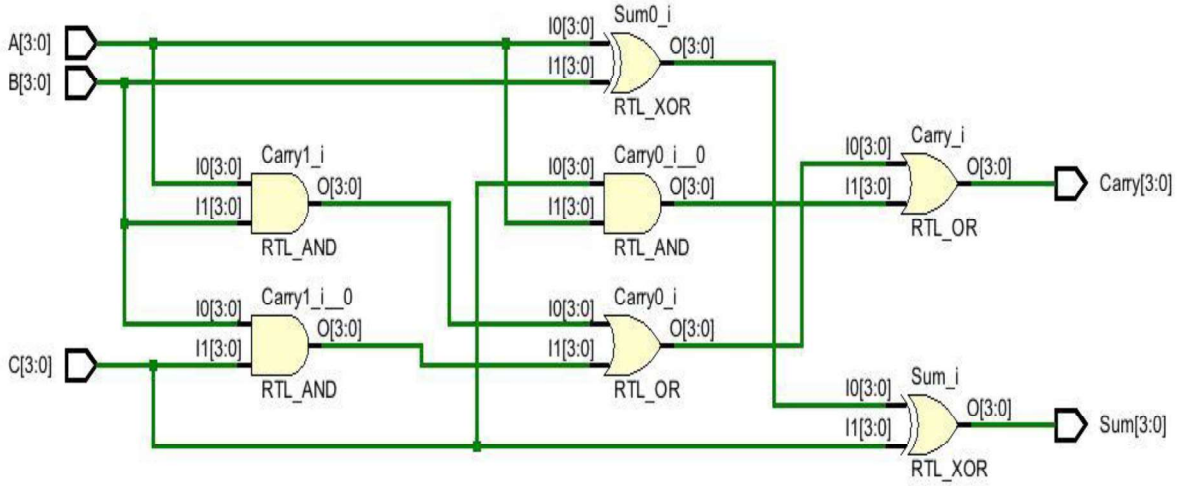
Code:

```
module CarrySaveAdder4Bit (
   input  [3:0] A, B, C,
   output [3:0] Sum,
   output [3:0] Carry
);

   assign Sum   = A ^ B ^ C;  // Sum without carry propagation
   assign Carry = (A & B) | (B & C) | (C & A); // Carry bits

endmodule
```

Testbench:

```
module CarrySaveAdder4Bit_tb;
   reg [3:0] A, B, C;
   wire [3:0] Sum, Carry;

   // Instantiate the CarrySaveAdder4Bit module
   CarrySaveAdder4Bit uut (
      .A(A),
      .B(B),
      .C(C),
      .Sum(Sum),
      .Carry(Carry)
   );

   initial begin
      // Monitor the values
      $monitor("Time=%0t A=%b B=%b C=%b | Sum=%b Carry=%b", $time, A, B, C, Sum, Carry);

      // Apply test cases
      A = 4'b0000; B = 4'b0000; C = 4'b0000; #10;
      A = 4'b0001; B = 4'b0001; C = 4'b0001; #10;
      A = 4'b0010; B = 4'b0011; C = 4'b0001; #10;
      A = 4'b0110; B = 4'b0101; C = 4'b0011; #10;
      A = 4'b1111; B = 4'b1111; C = 4'b1111; #10;

      // End simulation
      $finish;
   End
```

endmodule



Description:

A Carry Save Adder (CSA) is a type of adder used to efficiently add multiple binary numbers, commonly used in multipliers and digital signal processing. Instead of propagating the carry immediately, it stores the carry separately and passes it to the next stage, reducing overall delay.

The carry bits are shifted and added in the next stage using a conventional adder, typically a Ripple Carry or Carry Look-Ahead Adder.

Summary:

In Xilinx, a Carry Save Adder can be implemented using Verilog or VHDL for FPGA-based applications. The design involves multiple full adders operating in parallel without immediate carry propagation, making it significantly faster than a Ripple Carry Adder for multi-operand addition. Using Xilinx Vivado or ISE, the CSA can be synthesized and optimized for applications like multiplication and digital arithmetic circuits, where high-speed addition of multiple numbers is required.

4. Carry skip adders

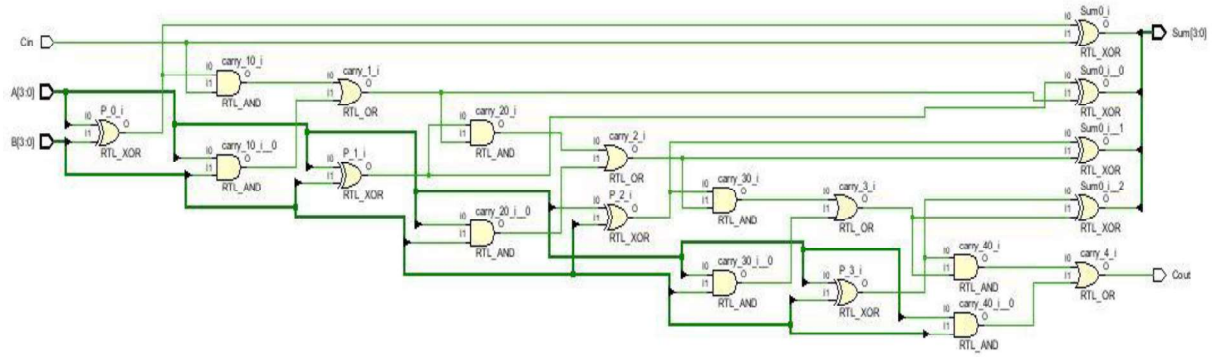
Code:

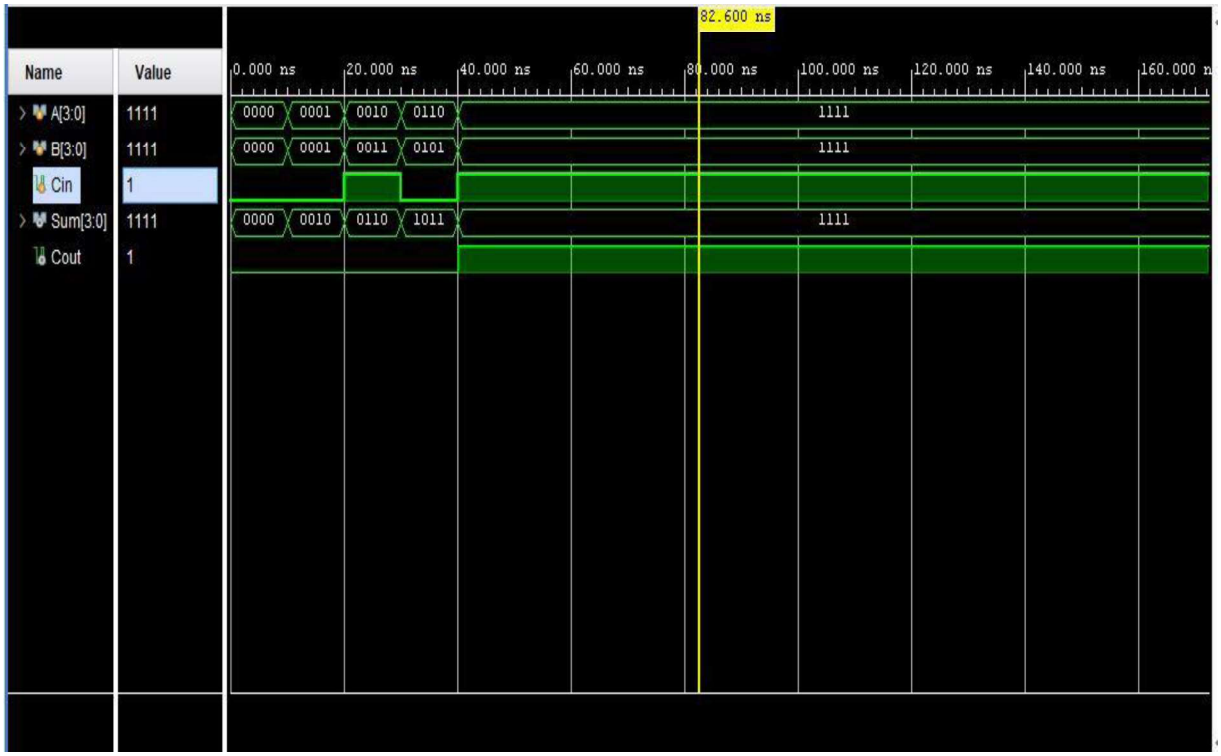
```
module carry_skip_adder(
   input  [3:0] A, B,
   input  Cin,
   output [3:0] Sum,
   output Cout
);

   wire [4:0] carry;
   wire [3:0] P;

   assign carry[0] = Cin;

   genvar i;
   generate
      for (i = 0; i < 4; i = i + 1) begin : bit_slice
         assign P[i] = A[i] ^ B[i];
         assign Sum[i] = P[i] ^ carry[i];
         assign carry[i+1] = (P[i] & carry[i]) | (A[i] & B[i]);
      end
   endgenerate

   assign Cout = carry[4];

endmodule
```

Testbench:

```
module carry_skip_tb;
   reg [3:0] A, B;
   reg Cin;
   wire [3:0] Sum;
   wire Cout;

   // Instantiate the CarrySkipAdder4Bit module
   carry_skip_adder uut (
      .A(A),
      .B(B),
      .Cin(Cin),
      .Sum(Sum),
      .Cout(Cout)
   );

   initial begin
      // Monitor the values
      $monitor("Time=%0t A=%b B=%b Cin=%b | Sum=%b Cout=%b", $time, A, B, Cin, Sum,
Cout);

      // Apply test cases
      A = 4'b0000; B = 4'b0000; Cin = 0; #10;
      A = 4'b0001; B = 4'b0001; Cin = 0; #10;
      A = 4'b0010; B = 4'b0011; Cin = 1; #10;
      A = 4'b0110; B = 4'b0101; Cin = 0; #10;
      A = 4'b1111; B = 4'b1111; Cin = 1; #10;

      // End simulation

   end
```

endmodule





Description:

A Carry Skip Adder (CSA) is an optimized adder designed to reduce carry propagation delay in large bit-width additions. It divides the adder into blocks, allowing the carry to "skip" over groups of bits when possible, reducing the worst-case delay.

Summary:

In Xilinx, a Carry Skip Adder can be implemented using Verilog or VHDL in an FPGA. It is designed by grouping full adders into blocks and incorporating skip logic to bypass unnecessary carry computations. This reduces propagation delay compared to a ripple-carry adder. Using Xilinx Vivado or ISE, the design can be synthesized and tested for performance improvements. CSA provides a balance between speed and hardware complexity, making it a practical choice for arithmetic circuits in FPGA-based systems.

5. Inverter

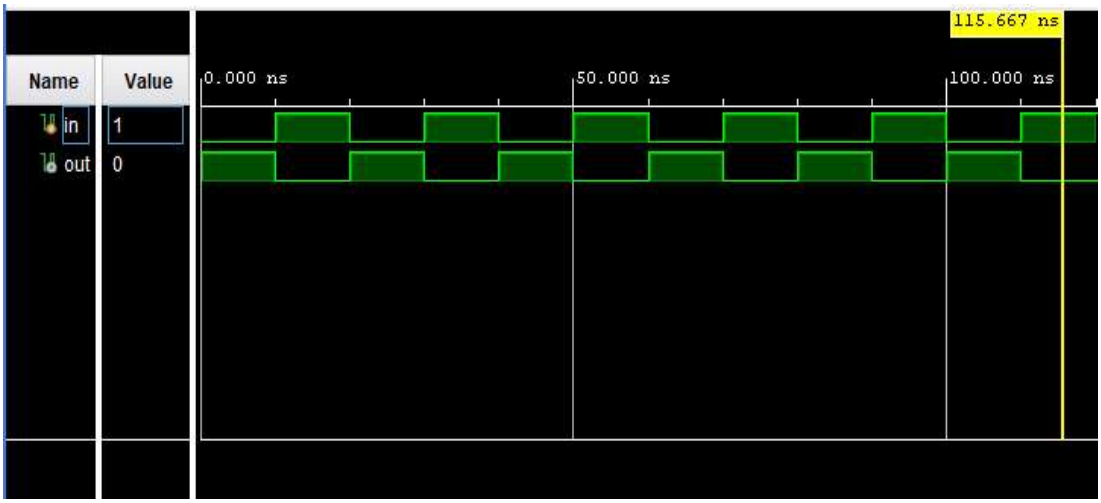
Code:

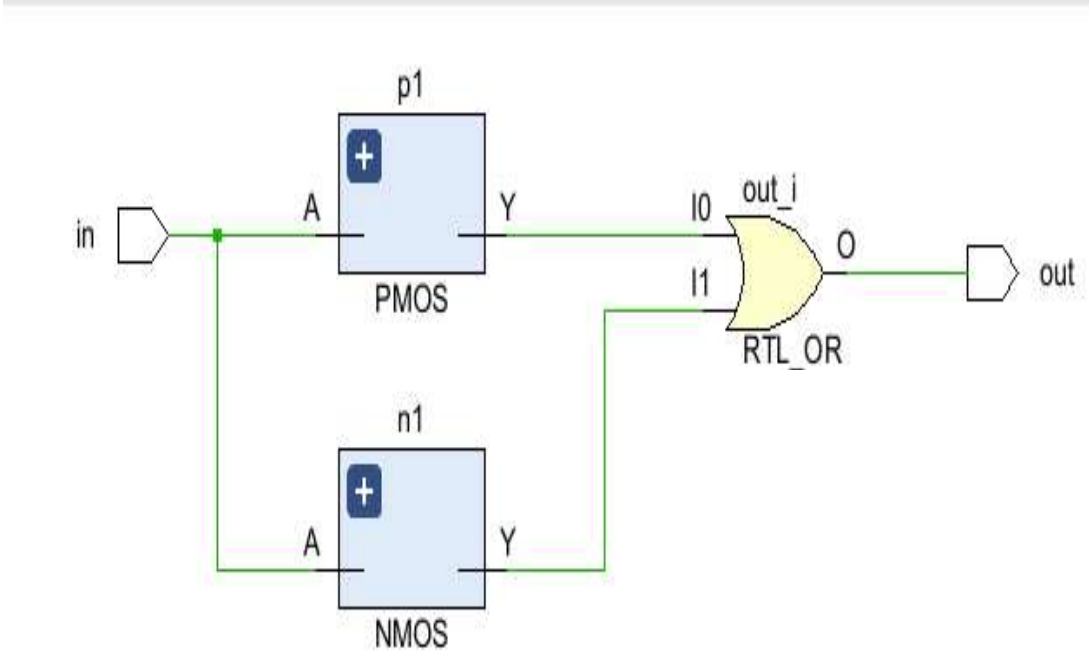
```
module PMOS(
   input wire A,
   output wire Y
   );
   assign Y=~A;
endmodule
module NMOS(
   input wire A,
   output wire Y
   );
   assign Y=~A;
endmodule
module CMOS_INV(
   input  wire in,
   output wire out
   );
   wire nmos_out,pmos_out;
   PMOS p1(.A(in),.Y(pmos_out));
   NMOS n1(.A(in),.Y(nmos_out));
   assign out=pmos_out|nmos_out;
endmodule
```

Testbench:

```
module CMOS_TB;
reg in;
wire out;
CMOS_INV uut(
.in(in),
.out(out)
);
initial begin
in=0;
#10;
in=1;
#10;
in=0;
#10;
in=1;
#10;
in=0;
#10;
in=1;
#10;
in=0;
#10;
in=1;
#10;
in=0;
#10;
in=1;
#10;
in=0;
#10;
in=1;
#10;
$finish;
```

```
end
endmodule
```





Description:

A CMOS (Complementary Metal-Oxide-Semiconductor) inverter is a fundamental logic circuit used to invert the input signal. It consists of:

- **A PMOS transistor (pull-up network)**, which conducts when the input is LOW (0) and connects the output to VDD (HIGH).
- **An NMOS transistor (pull-down network)**, which conducts when the input is HIGH (1) and connects the output to GND (LOW).

The operation follows:

- If **Input = 0**, PMOS turns ON, NMOS turns OFF → **Output = 1**
- If **Input = 1**, NMOS turns ON, PMOS turns OFF → **Output = 0**

Summary:

In Xilinx, a CMOS inverter can be implemented using Verilog or VHDL by defining a NOT gate (`Y = ~A`). The synthesized design is mapped onto FPGA logic elements such as LUTs, rather than physical CMOS transistors. However, for ASIC design, CMOS inverters are implemented at the transistor level using Xilinx Vivado tools with HDL-based structural descriptions. CMOS inverters are widely used in digital circuits, including buffers, logic gates, and clock distribution networks.

6. Array Multiplier

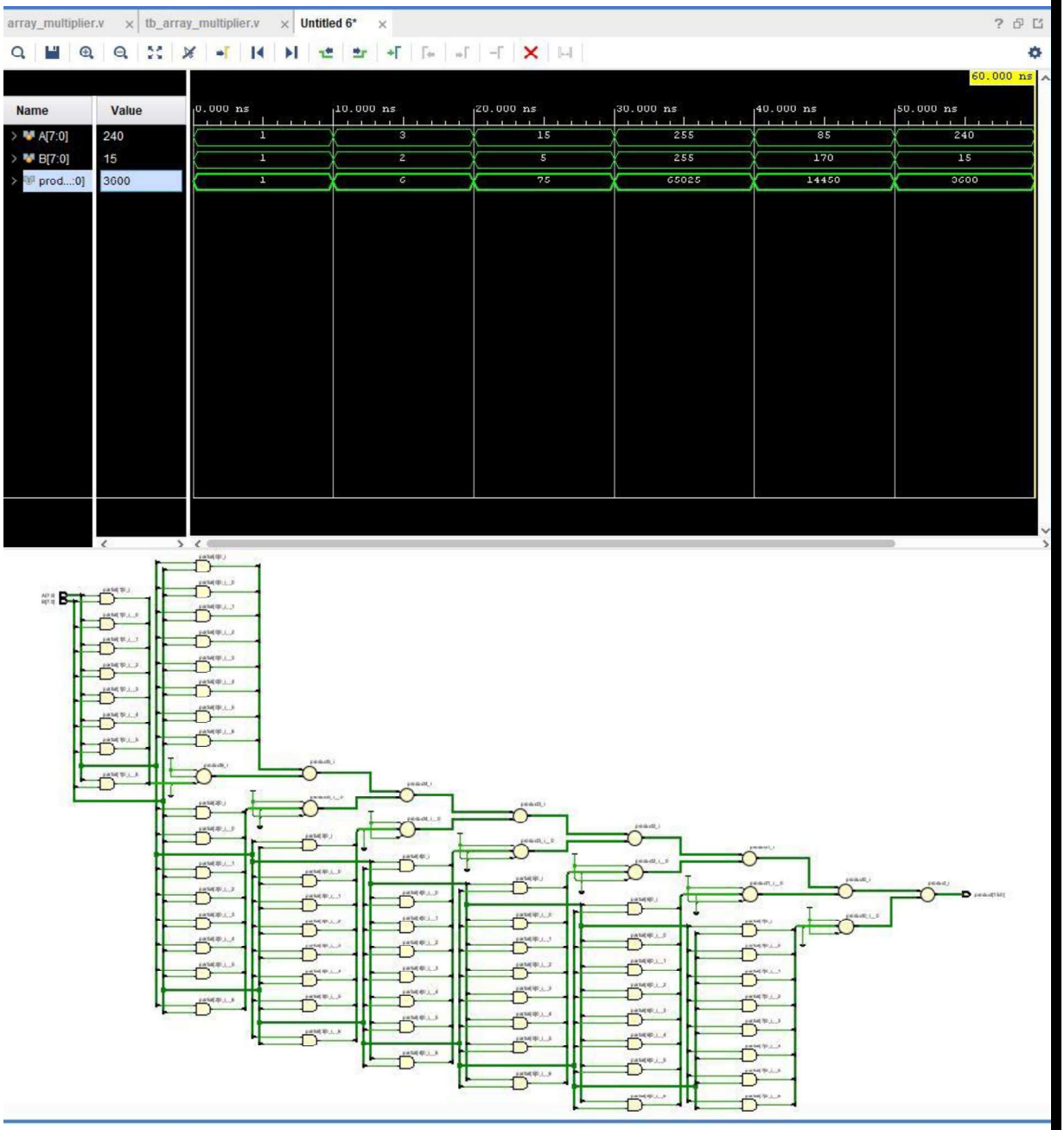
Code:

```
module array_multiplier (
    input [7:0] A,
    input [7:0] B,
    output [15:0] product
);

    wire [7:0] partial [7:0];

    genvar i, j;
    generate
        for (i = 0; i < 8; i = i + 1) begin :
            for (j = 0; j < 8; j = j + 1) begin :
                assign partial[i][j] = A[i] & B[j];
            end
        end
    endgenerate

    assign product = (partial[0]) +
                     (partial[1] << 1) +
                     (partial[2] << 2) +
                     (partial[3] << 3) +
                     (partial[4] << 4) +
                     (partial[5] << 5) +
                     (partial[6] << 6) +
                     (partial[7] << 7);

endmodule
```

Testbench:

```
module tb_array_multiplier;

    reg [7:0] A, B;
    wire [15:0] product;

    array_multiplier uut (
        .A(A),
        .B(B),
        .product(product)
    );
```

```
initial begin
    $display("A         B         | Product");
    $display("------------------------------");

    A = 8'b00000001; B = 8'b00000001;
    #10; $display("%b   %b   | %b", A, B, product);

    A = 8'b00000011; B = 8'b00000010;
    #10; $display("%b   %b   | %b", A, B, product);

    A = 8'b00001111; B = 8'b00000101;
    #10; $display("%b   %b   | %b", A, B, product);

    A = 8'b11111111; B = 8'b11111111;
    #10; $display("%b   %b   | %b", A, B, product);

    A = 8'b01010101; B = 8'b10101010;
    #10; $display("%b   %b   | %b", A, B, product);

    A = 8'b11110000; B = 8'b00001111;
    #10; $display("%b   %b   | %b", A, B, product);

    $finish;
end
```



Description:

A **4-bit Array Multiplier** is a combinational circuit used for multiplying two 4-bit binary numbers. It follows the traditional paper-and-pencil multiplication method, where each bit of one operand is multiplied with every bit of the other operand, and the partial products are summed accordingly.

If the two 4-bit numbers are:

A=A3A2A1A0A B=B3B2B1B0B

then the partial products are calculated using AND gates:

P=A.B

The partial products are then added using **half adders (HA)** and **full adders (FA)**.

Summary:

In Xilinx, a 4-bit Array Multiplier can be implemented using **Verilog or VHDL** and synthesized onto an FPGA using **Vivado or ISE**. The design consists of:

- **AND gates** to generate partial products
- **Half Adders and Full Adders** to sum the partial products efficiently
- A structured arrangement to minimize delay and optimize performance

Since it is a combinational circuit, the multiplication is performed in a single clock cycle, making it fast but resource-intensive for larger bit-widths. This design is widely used in digital arithmetic units such as ALUs and DSP applications.

7. Barrel shifter

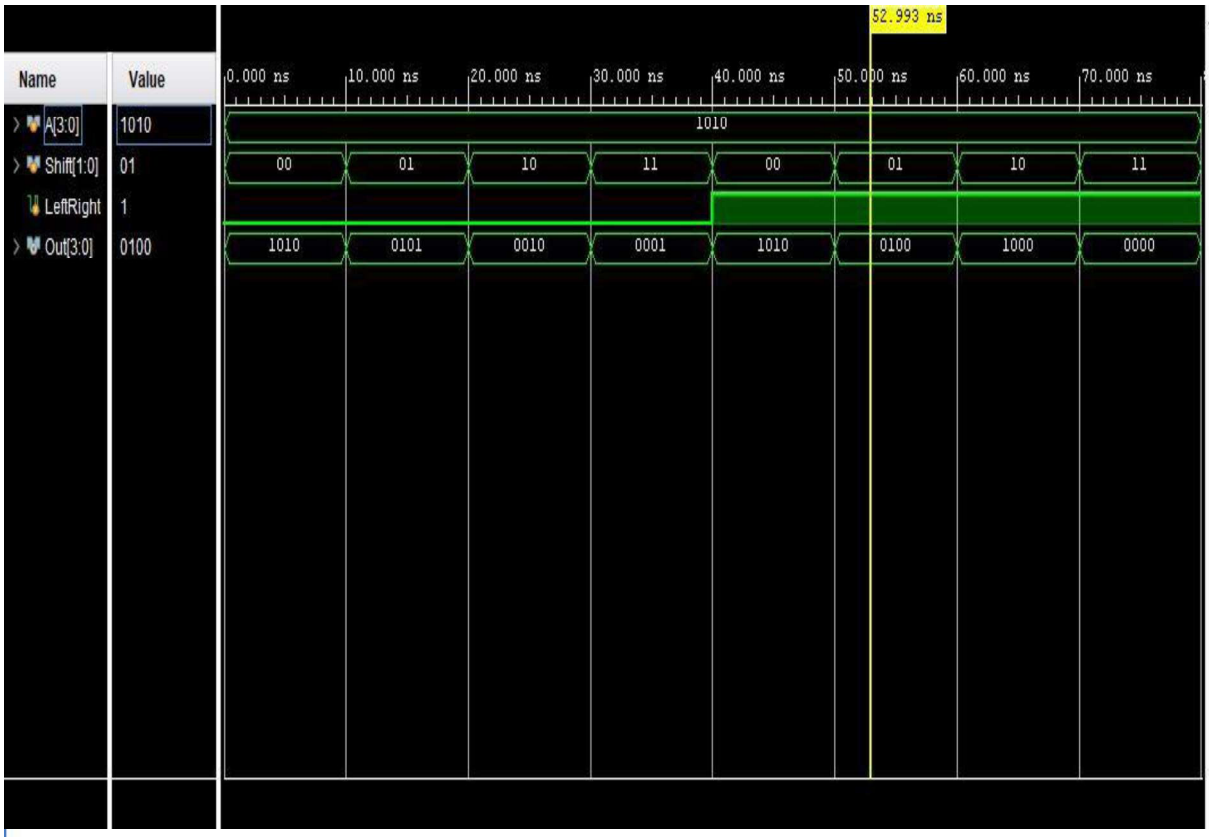
Code:

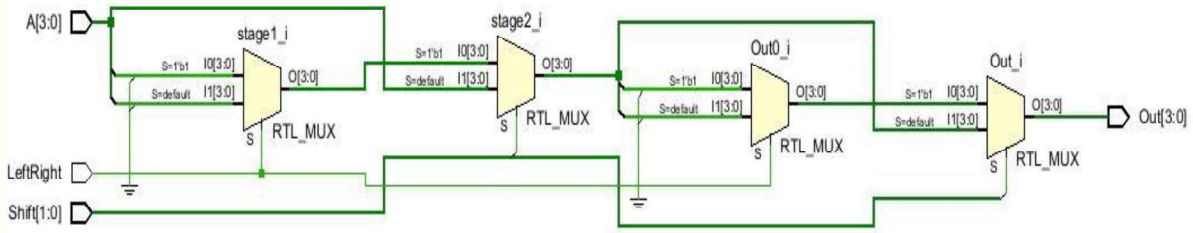
```
module BarrelShifter4Bit (
    input [3:0] A,
    input [1:0] Shift,
    input LeftRight,
    output [3:0] Out
);

    wire [3:0] stage1, stage2;

    assign stage1 = (LeftRight) ? {A[2:0], 1'b0} : {1'b0, A[3:1]};

    assign stage2 = (Shift[0]) ? stage1 : A;
    assign Out = (Shift[1]) ? ((LeftRight) ? {stage2[1:0], 2'b00} : {2'b00, stage2[3:2]}) : stage2;

endmodule
```

Testbench:

```
module BarrelShifter4Bit_tb;
    reg [3:0] A;
    reg [1:0] Shift;
    reg LeftRight;
    wire [3:0] Out;

    BarrelShifter4Bit uut (
        .A(A),
        .Shift(Shift),
        .LeftRight(LeftRight),
        .Out(Out)
    );
```

```
    initial begin

        $monitor("Time=%0t A=%b Shift=%b LeftRight=%b | Out=%b", $time, A, Shift, LeftRight, Out);

        // Apply test cases
        A = 4'b1010; Shift = 2'b00; LeftRight = 0; #10;
        A = 4'b1010; Shift = 2'b01; LeftRight = 0; #10;
        A = 4'b1010; Shift = 2'b10; LeftRight = 0; #10;
        A = 4'b1010; Shift = 2'b11; LeftRight = 0; #10;
        A = 4'b1010; Shift = 2'b00; LeftRight = 1; #10;
        A = 4'b1010; Shift = 2'b01; LeftRight = 1; #10;
        A = 4'b1010; Shift = 2'b10; LeftRight = 1; #10;
        A = 4'b1010; Shift = 2'b11; LeftRight = 1; #10;

        // End simulation
        $finish;
    end
endmodule
```


52.993 ns
Name
Value
0.000 ns
10.000 ns
20.000 ns
30.000 ns
40.000 ns
50.000 ns
60.000 ns
70.000 ns
A[3:0]
1010
Shift[1:0]
01
LeftRight
1
Out[3:0]
0100
1010
00
01
10
11
00
01
10
11
1010
0101
0010
0001
1010
0100
1000
0000


A[3:0]
stage1_i
stage2_i
Out0_i
Out_i
S=1'b1
S=default
I0[3:0]
I1[3:0]
O[3:0]
S
RTL_MUX
Out[3:0]
LeftRight
Shift[1:0]

Description:

A **4-bit Barrel Shifter** is a combinational circuit used for shifting data left or right by a specified number of positions in a single clock cycle. It uses multiplexers to perform shifts efficiently without requiring multiple clock cycles, unlike sequential shifters.

A 4-bit barrel shifter can perform:

- **Logical Left Shift (LSL)**
- **Logical Right Shift (LSR)**
- **Rotate Left (ROL)**
- **Rotate Right (ROR)**

The shift amount is controlled by a **2-bit select input** (`S1 S0`):

- `00` → No Shift
- `01` → Shift by 1 bit
- `10` → Shift by 2 bits
- `11` → Shift by 3 bits

Summary:

In **Xilinx**, a 4-bit Barrel Shifter can be implemented using **Verilog or VHDL** and synthesized onto an **FPGA using Vivado or ISE**. The design is based on **multiplexers** that dynamically select the output based on the shift amount. Barrel shifters are widely used in ALUs, DSP applications, and floating-point arithmetic operations due to their high-speed shifting capability.

8. FIFO AND LIFO

Code:

```
module lifo #(parameter WIDTH = 8, DEPTH = 8)(
    input clk,              // Clock signal
    input reset,            // Active-high reset
    input push,             // Push enable
    input pop,              // Pop enable
    input [WIDTH-1:0] data_in, // Data input
    output reg [WIDTH-1:0] data_out, // Data output
    output reg full,        // Stack full flag
    output reg empty        // Stack empty flag
);

    reg [WIDTH-1:0] stack [DEPTH-1:0]; // Stack memory
    reg [2:0] sp = 0;                  // Stack pointer

    always @(posedge clk or posedge reset) begin
        if (reset) begin
            sp <= 0;
            full <= 0;
            empty <= 1;
        end else begin
            // Push Operation
            if (push && !full) begin
                stack[sp] <= data_in;
                sp <= sp + 1;
            end

            // Pop Operation
            if (pop && !empty) begin
                sp <= sp - 1;
                data_out <= stack[sp - 1];
            end
```

```
            // Update full and empty flags
            full  <= (sp == DEPTH);
            empty <= (sp == 0);
        end
    end
endmodule

module fifo #(parameter WIDTH = 8, DEPTH = 8)(
    input clk,              // Clock signal
    input reset,            // Active-high reset
    input wr_en,            // Write enable
    input rd_en,            // Read enable
    input [WIDTH-1:0] data_in, // Data input
    output reg [WIDTH-1:0] data_out, // Data output
    output reg full,        // FIFO full flag
    output reg empty        // FIFO empty flag
);

    reg [WIDTH-1:0] memory [DEPTH-1:0]; // FIFO memory storage
    reg [2:0] wr_ptr = 0, rd_ptr = 0;   // Write and Read pointers
    reg [3:0] count = 0;                // Number of elements in FIFO

    always @(posedge clk or posedge reset) begin
        if (reset) begin
            wr_ptr <= 0;
            rd_ptr <= 0;
            count  <= 0;
            full   <= 0;
            empty  <= 1;
        end else begin
            // Write Operation
            if (wr_en && !full) begin
                memory[wr_ptr] <= data_in;
                wr_ptr <= (wr_ptr + 1) % DEPTH;
```

```
            count <= count + 1;
        end

        // Read Operation
        if (rd_en && !empty) begin
            data_out <= memory[rd_ptr];
            rd_ptr <= (rd_ptr + 1) % DEPTH;
            count <= count - 1;
        end

        // Update full and empty flags
        full  <= (count == DEPTH);
        empty <= (count == 0);
    end
  end
endmodule
```

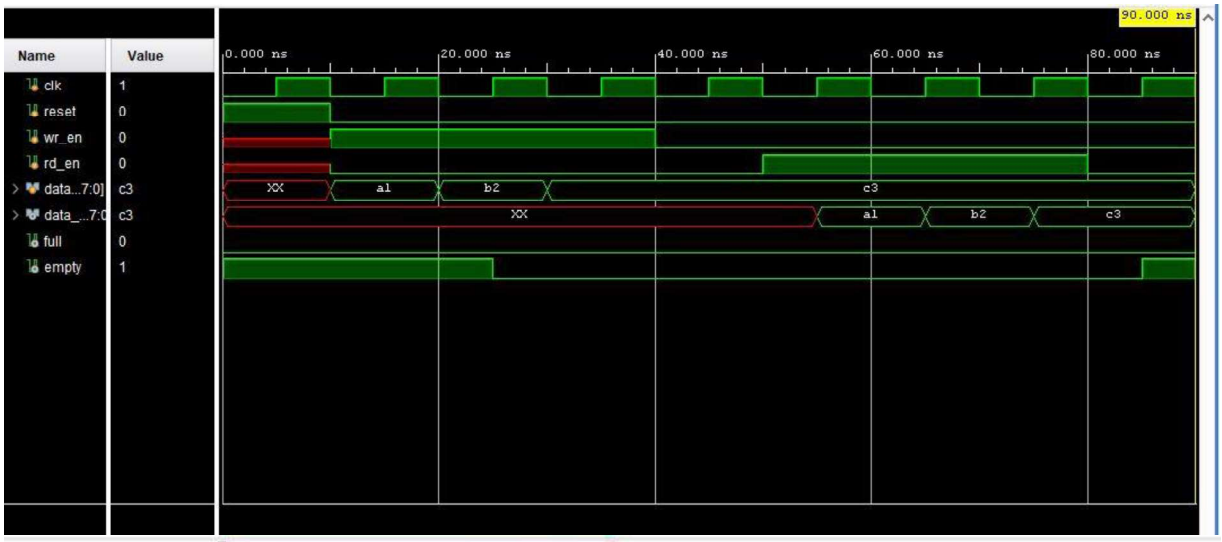
Testbench:

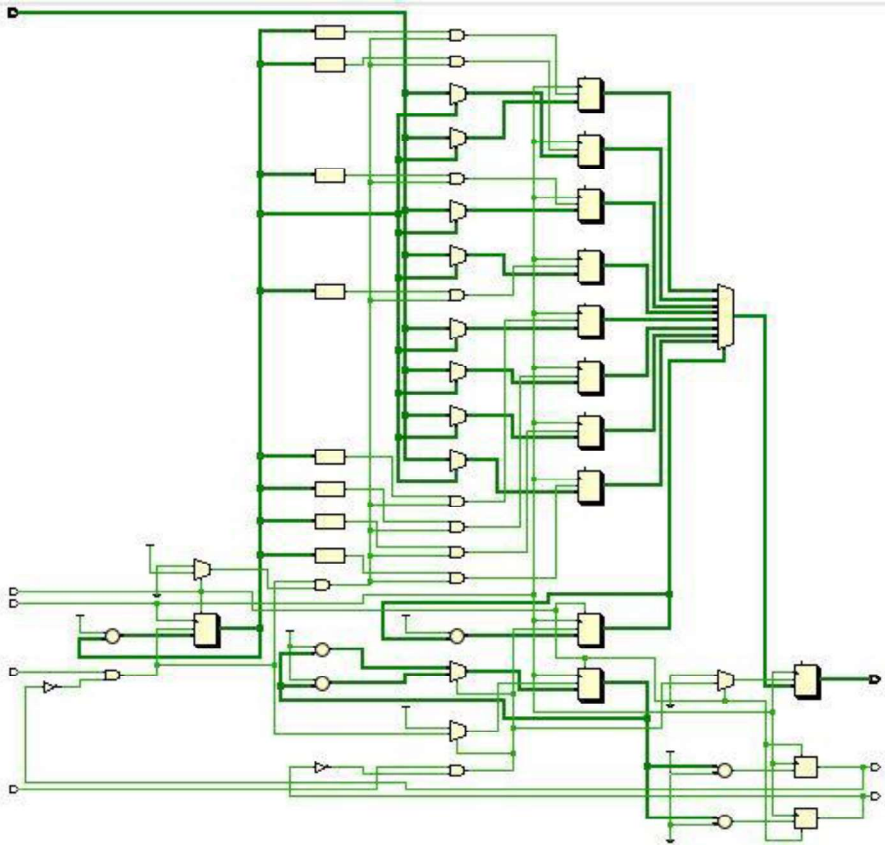
```
module tb_lifo;
  reg clk, reset, push, pop;
  reg [7:0] data_in;
  wire [7:0] data_out;
  wire full, empty;

  // Instantiate LIFO
  lifo uut (
    .clk(clk),
    .reset(reset),
    .push(push),
    .pop(pop),
    .data_in(data_in),
    .data_out(data_out),
    .full(full),
    .empty(empty)
```

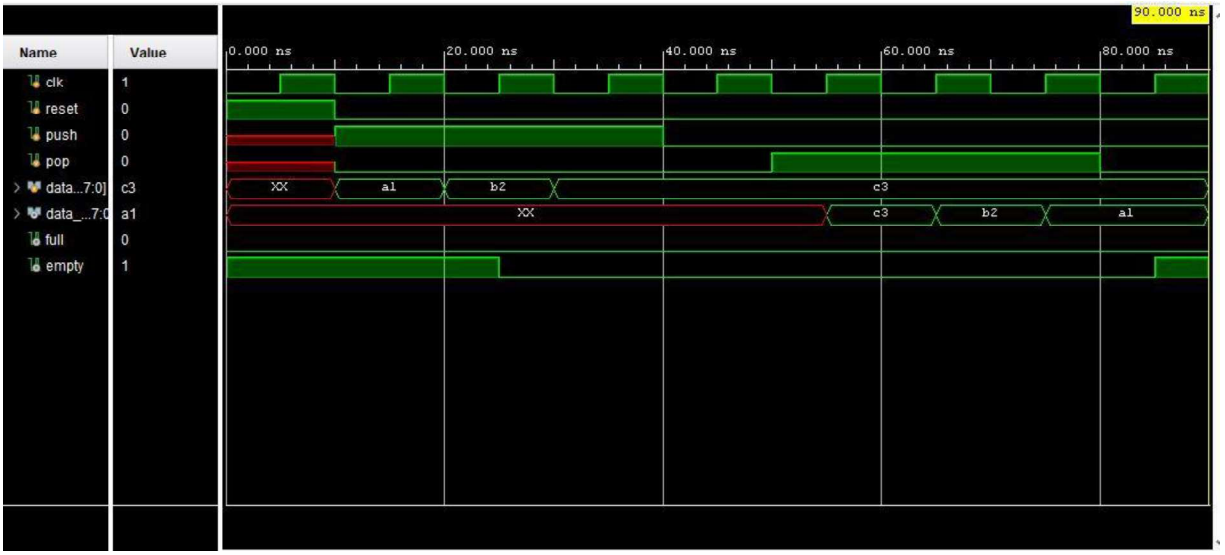
```
    );

    // Clock Generation
    always #5 clk = ~clk;

    initial begin
        clk = 0;
        reset = 1;
        #10 reset = 0;

        // Push data into LIFO
        push = 1; pop = 0; data_in = 8'hA1; #10;
        data_in = 8'hB2; #10;
        data_in = 8'hC3; #10;
        push = 0; #10; // Stop pushing

        // Pop data from LIFO
        pop = 1; #30;
        pop = 0; #10; // Stop popping

        $finish;
    end

    initial begin
        $monitor("Time: %0t | Data In: %h | Data Out: %h | Full: %b | Empty: %b",
                 $time, data_in, data_out, full, empty);
    end
endmodule



module tb_fifo;
    reg clk, reset, wr_en, rd_en;
    reg [7:0] data_in;
    wire [7:0] data_out;
```

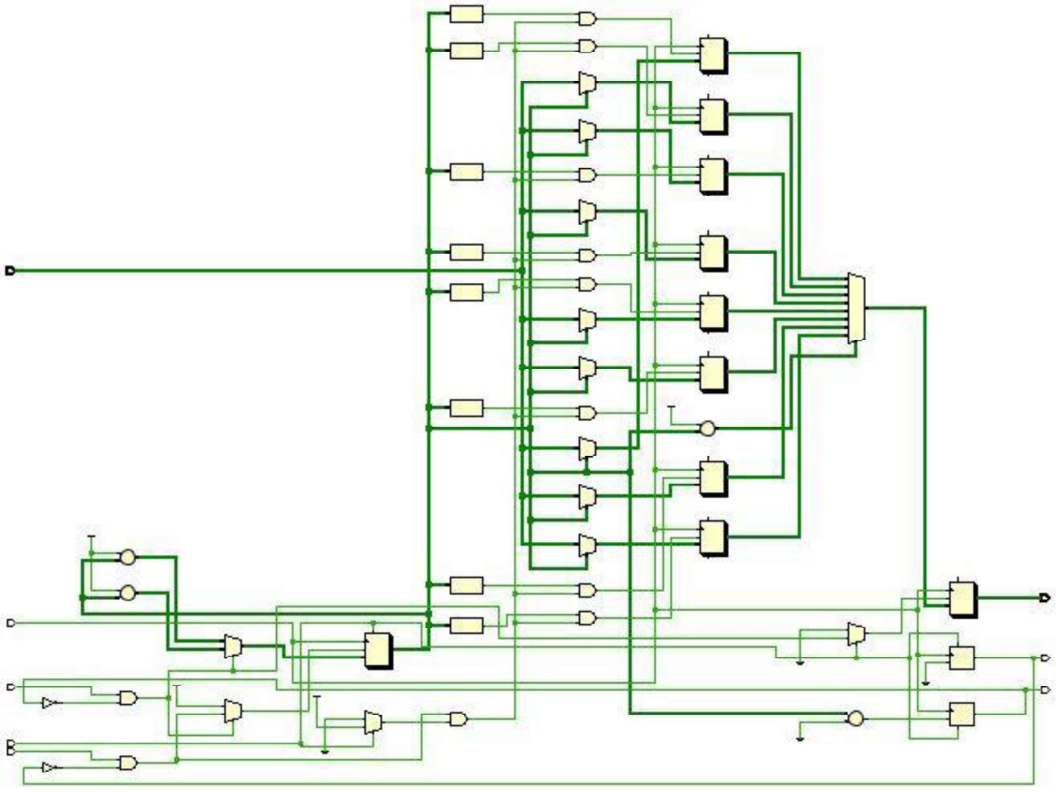
```
wire full, empty;

// Instantiate FIFO
fifo uut (
    .clk(clk),
    .reset(reset),
    .wr_en(wr_en),
    .rd_en(rd_en),
    .data_in(data_in),
    .data_out(data_out),
    .full(full),
    .empty(empty)
);

// Clock Generation
always #5 clk = ~clk;

initial begin
    clk = 0;
    reset = 1;
    #10 reset = 0;

    // Write data into FIFO
    wr_en = 1; rd_en = 0; data_in = 8'hA1; #10;
    data_in = 8'hB2; #10;
    data_in = 8'hC3; #10;
    wr_en = 0; #10; // Stop writing

    // Read data from FIFO
    rd_en = 1; #30;
    rd_en = 0; #10; // Stop reading

    $finish;
end
```

```
    initial begin
        $monitor("Time: %0t | Data In: %h | Data Out: %h | Full: %b | Empty: %b",
                $time, data_in, data_out, full, empty);
    end
endmodule
```









FIFO (First In, First Out)

- FIFO is a type of **queue** where the first data written is the first data read.
- Used in buffering, data streaming, and inter-process communication.
- Implemented using **registers or RAM** with read and write pointers.
- When a new value is written, the write pointer moves forward. When a value is read, the read pointer moves forward.
- Can be **synchronous (clock-based)** or **asynchronous (different clock domains)**.

LIFO (Last In, First Out)

- LIFO is a **stack** where the last data written is the first to be read.
- Used in recursive function calls, undo operations, and backtracking algorithms.
- Implemented using **stack registers or RAM** with a stack pointer.

- Data is **pushed** onto the stack and **popped** when needed.
- Works with **stack overflow/underflow protection** in FPGA designs.

Summary:

In **Xilinx**, both FIFO and LIFO can be implemented using **Verilog or VHDL** and synthesized in **Vivado or ISE**.

- **FIFO** can be implemented using **BRAM (Block RAM)** or **shift registers** for efficient memory utilization.
- **LIFO** is implemented using a **stack structure with a stack pointer** to track the top of the stack.

These structures are widely used in **communication protocols, microprocessor designs, and memory management**.

FLIP-FLOPS

9. D-flip flop

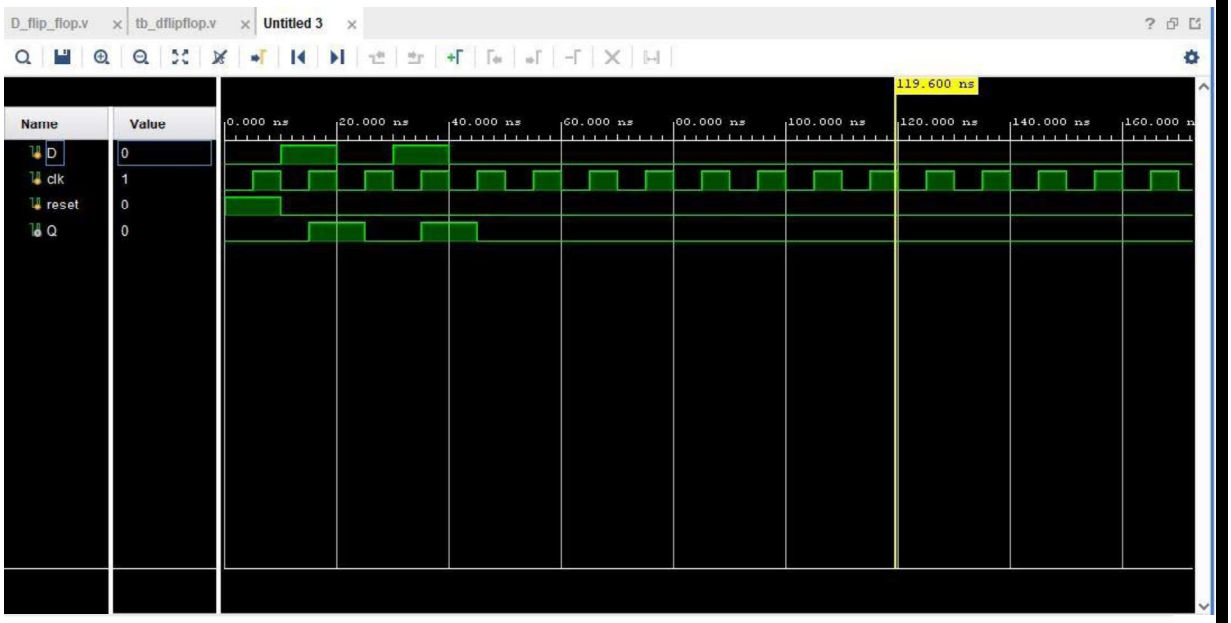
Code:

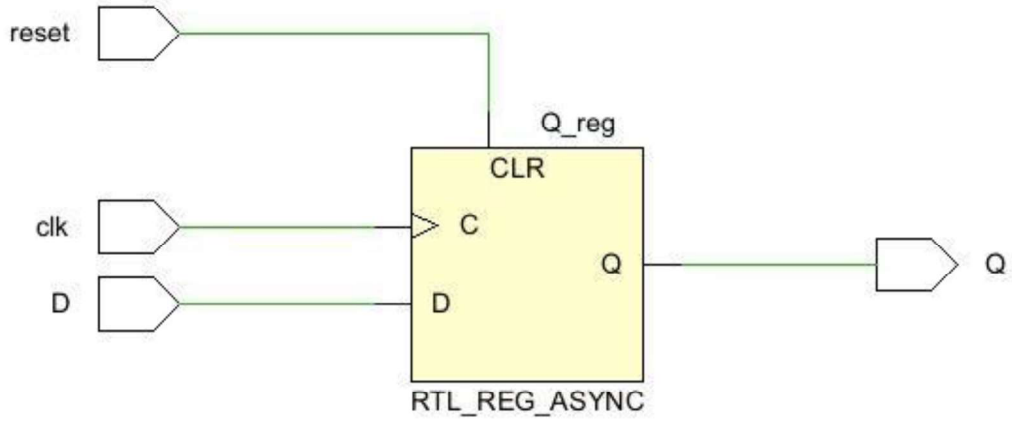
```
module d_flip_flop (
    input D,
    input clk,
    input reset,
    output reg Q
);

    always @(posedge clk or posedge reset) begin
        if (reset) begin
            Q <= 0;
        end else begin
            Q <= D;
        end
    end

endmodule
```

Testbench:

```
module tb_d_flip_flop;

    reg D;
    reg clk;
    reg reset;
    wire Q;


    d_flip_flop uut (
        .D(D),
        .clk(clk),
        .reset(reset),
        .Q(Q)
    );
```

```
    // Generate clock signal
    always begin
        #5 clk = ~clk;
    end

    initial begin

        clk = 0;
        reset = 0;
        D = 0;

        reset = 1;
        #10;
        reset = 0;

        D = 1;
        #10;
        D = 0;
        #10;
        D = 1;  /
        #10;
        D = 0;
        #10;


    end


    initial begin
        $monitor("At time %t, D = %b, Q = %b", $time, D, Q);
    end

endmodule
```


D_flip_flop.v
tb_dflipflop.v
Untitled 3
Name
Value
D
clk
reset
Q
0
1
0
0
119.600 ns
0.000 ns
20.000 ns
40.000 ns
60.000 ns
80.000 ns
100.000 ns
120.000 ns
140.000 ns
160.000 n


reset
Q_reg
CLR
clk
C
Q
Q
D
D
RTL_REG_ASYNC

10. T-Flip Flop

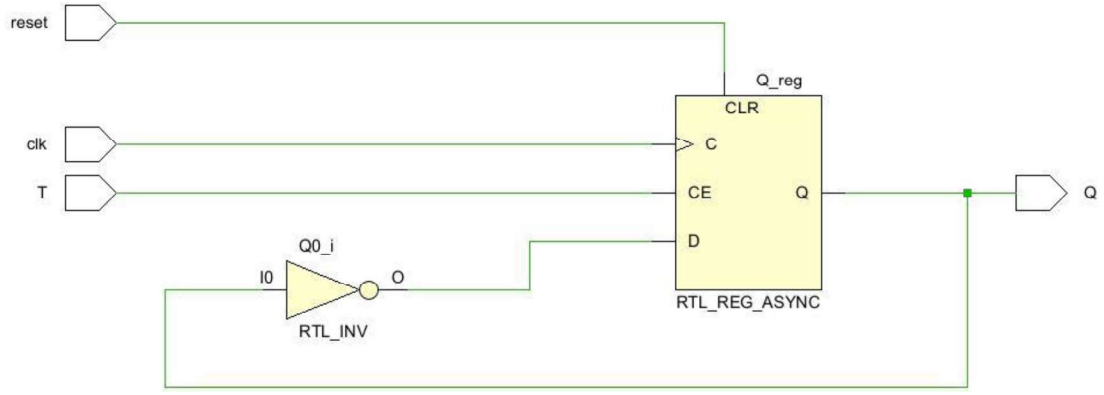
Code:

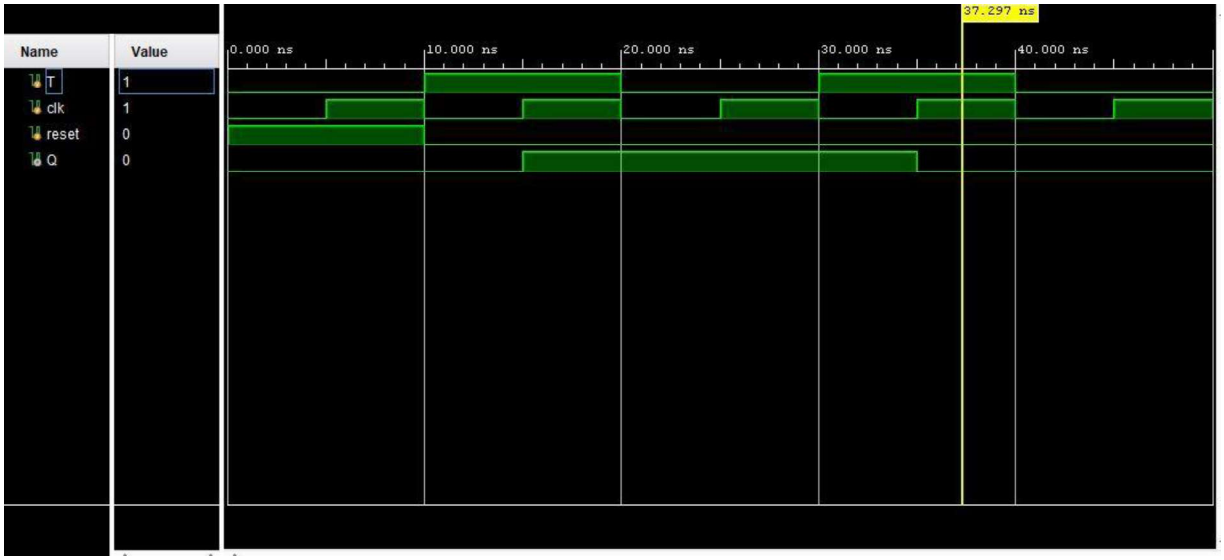
```
module t_flip_flop (
    input T,
    input clk,
    input reset,
    output reg Q
);
    always @(posedge clk or posedge reset) begin
        if (reset) begin
            Q <= 0;
        end else if (T) begin
            Q <= ~Q; 1
        end
    end


endmodule
```

Testbench:

```
module tb_t_flip_flop;

    reg T;
    reg clk;
    reg reset;
    wire Q;


    t_flip_flop uut (
        .T(T),
        .clk(clk),
        .reset(reset),
        .Q(Q)
    );


    always begin
        #5 clk = ~clk;
    end

    initial begin
        // Initialize inputs
        clk = 0;
        reset = 0;
        T = 0;


        reset = 1;
        #10;
        reset = 0;
```

```
        T = 1;
        #10;
        T = 0;
        #10;
        T = 1;
        #10;
        T = 0;
        #10;
        $finish;
    end

    // Monitor the values of Q
    initial begin
        $monitor("At time %t, T = %b, Q = %b", $time, T, Q);
    end

endmodule
```





11. JK-Flip flop

Code:

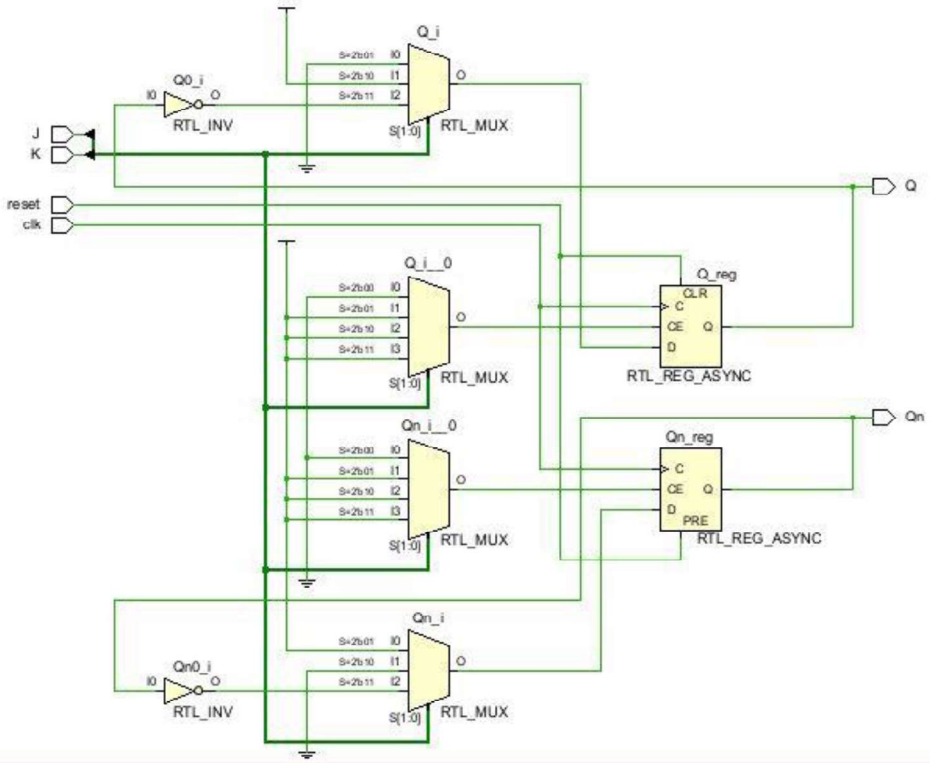
```
module jk_flip_flop (
    input J,
    input K,
    input clk,
    input reset,
    output reg Q,
    output reg Qn
);

    always @(posedge clk or posedge reset) begin
        if (reset) begin
            Q <= 0;
            Qn <= 1;
        end else begin
            case ({J, K})
                2'b00: begin
                    Q <= Q;
                    Qn <= Qn;
                end
                2'b01: begin
                    Q <= 0;
                    Qn <= 1;
                end
                2'b10: begin
                    Q <= 1;
                    Qn <= 0;
                end
                2'b11: begin
                    Q <= ~Q;
                    Qn <= ~Qn;
                end
            endcase
        end
    end

endmodule
```

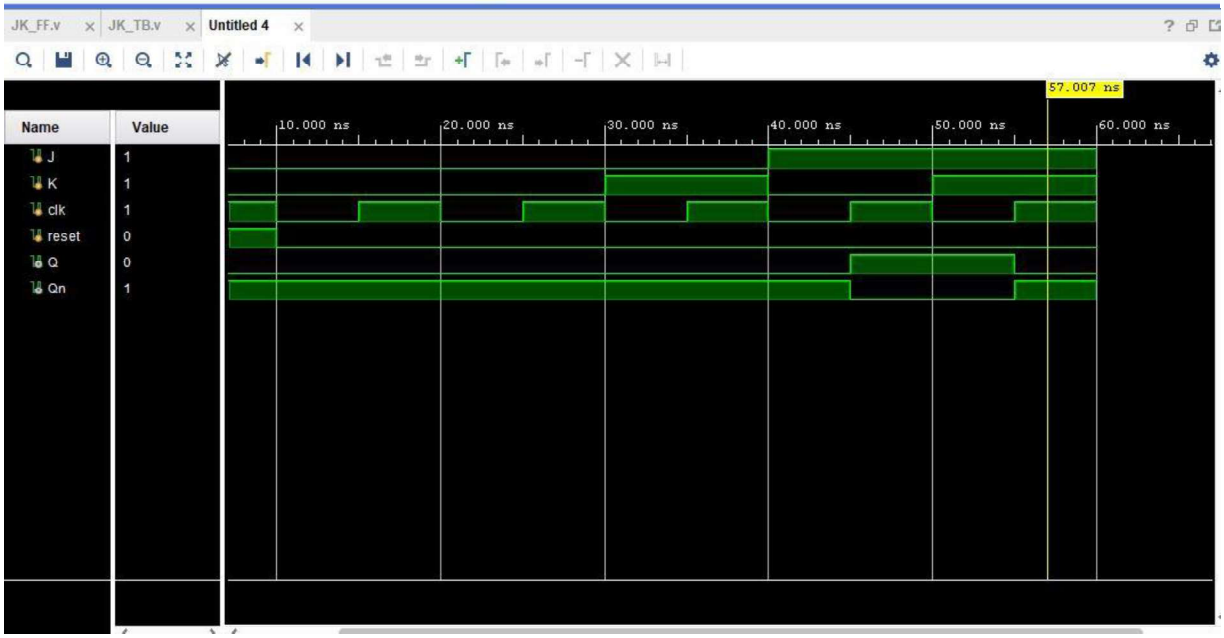
Testbench:

```
module tb_jk_flip_flop;

    reg J;
    reg K;
    reg clk;
    reg reset;
    wire Q;
    wire Qn;

    jk_flip_flop uut (
        .J(J),
        .K(K),
        .clk(clk),
        .reset(reset),
        .Q(Q),
        .Qn(Qn)
    );

    always begin
        #5 clk = ~clk;
    end

    initial begin
```

```
    clk = 0;
    reset = 0;
    J = 0;
    K = 0;

    reset = 1;
    #10;
    reset = 0;

    #10;
    J = 0; K = 0;
    #10;
    J = 0; K = 1;
    #10;
    J = 1; K = 0;
    #10;
    J = 1; K = 1;
    #10;

    $finish;
end

initial begin
    $monitor("At time %t, J = %b, K = %b, Q = %b, Qn = %b", $time, J, K, Q, Qn);
end
```

e


Q_i
Q0_i
RTL_INV
RTL_MUX
J
K
reset
clk
Q
Q_i__0
Q_reg
RTL_REG_ASYNC
Qn
Qn_i__0
Qn_reg
Qn_i
Qn0_i


JK_FF.v
JK_TB.v
Untitled 4
Name
Value
J
K
clk
reset
Q
Qn
1
1
1
0
0
1
10.000 ns
20.000 ns
30.000 ns
40.000 ns
50.000 ns
60.000 ns
57.007 ns

12. SR Flip Flop

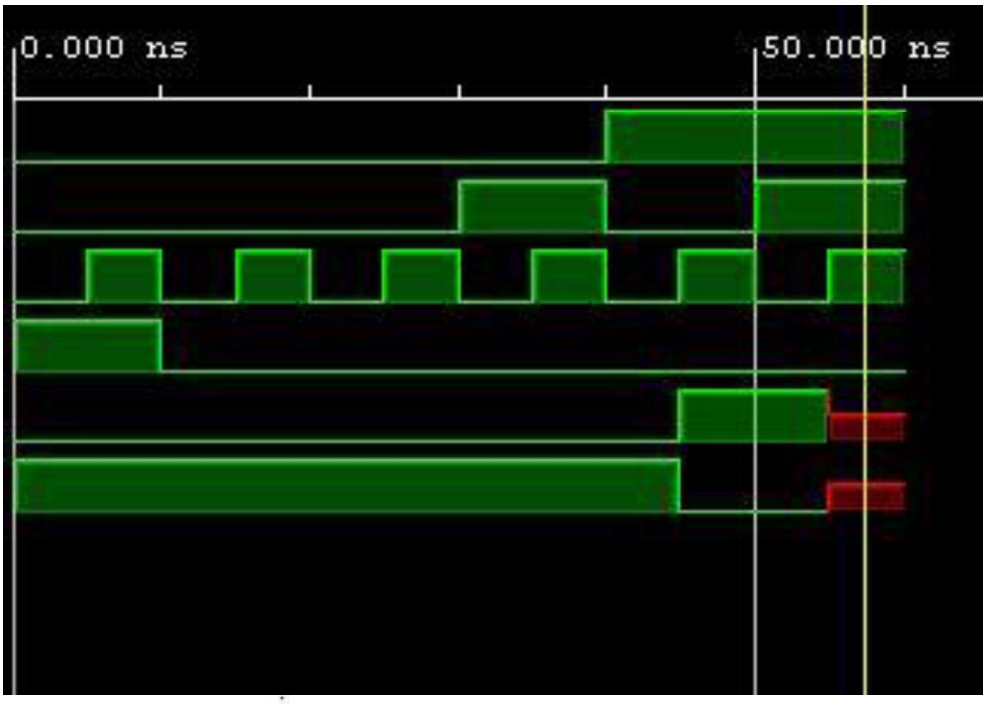
Code:

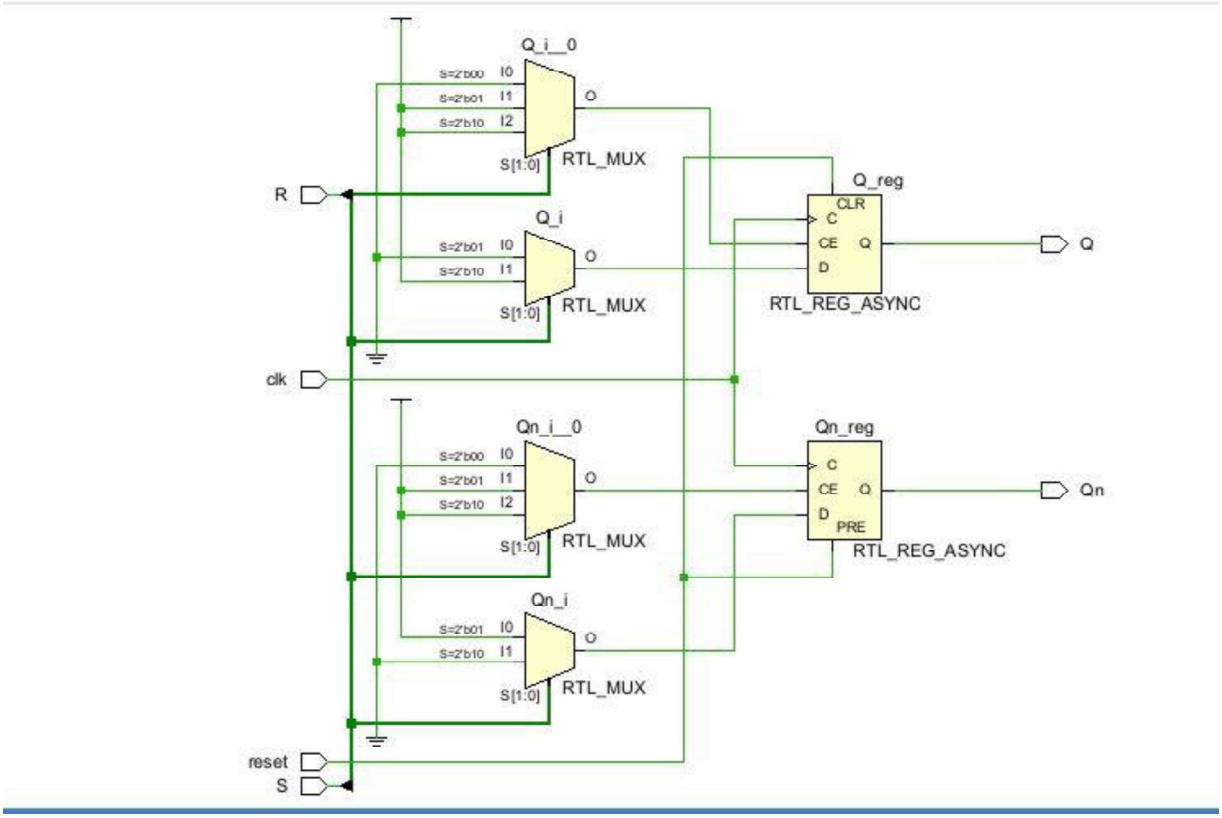
```
module srflipflop (
    input S,
    input R,
    input clk,
    input reset,
    output reg Q,
    output reg Qn
);

    always @(posedge clk or posedge reset) begin
        if (reset) begin
            Q <= 0;
            Qn <= 1;
        end else begin
            case ({S, R})
                2'b00: begin
                    Q <= Q;
                    Qn <= Qn;
                end
                2'b01: begin
                    Q <= 0;
                    Qn <= 1;
                end
                2'b10: begin
                    Q <= 1;
                    Qn <= 0;
                end
                2'b11: begin
                    Q <= 1'bx;
                    Qn <= 1'bx;
                end
            endcase
        end
    end

endmodule
```

Testbench:

```
module srfliptb;

    reg S;
    reg R;
    reg clk;
    reg reset;
    wire Q;
    wire Qn;

    srflipflop uut (
        .S(S),
        .R(R),
        .clk(clk),
        .reset(reset),
        .Q(Q),
        .Qn(Qn)
    );

    always begin
        #5 clk = ~clk;
    end

    initial begin
```

```
        clk = 0;
        reset = 0;
        S = 0;
        R = 0;

        reset = 1;
        #10;
        reset = 0;

        #10;
        S = 0; R = 0;
        #10;
        S = 0; R = 1;
        #10;
        S = 1; R = 0;
        #10;
        S = 1; R = 1;
        #10;

        $finish;
    end

    initial begin
        $monitor("At time %t, S = %b, R = %b, Q = %b, Qn = %b", $time, S, R, Q, Qn);
    end

endmodule
```


0.000 ns
50.000 ns


Q_i_0
S=2'b00 I0
S=2'b01 I1
S=2'b10 I2
O
S[1:0]
RTL_MUX
R
Q_i
S=2'b01 I0
S=2'b10 I1
O
S[1:0]
RTL_MUX
Q_reg
CLR
C
CE Q
D
RTL_REG_ASYNC
Q
clk
Qn_i_0
S=2'b00 I0
S=2'b01 I1
S=2'b10 I2
O
S[1:0]
RTL_MUX
Qn_reg
C
CE Q
D
PRE
RTL_REG_ASYNC
Qn
Qn_i
S=2'b01 I0
S=2'b10 I1
O
S[1:0]
RTL_MUX
reset
S

1. D Flip-Flop (Data/Delay Flip-Flop)

Description:

A D Flip-Flop is a memory element that captures and stores a single-bit value. On the rising edge of the clock signal, the input data (D) is transferred to the output (Q). It ensures a stable and synchronized data transfer.

Use Cases:

- Used in shift registers to store and transfer data.
- Utilized in pipeline registers for clock synchronization.
- Commonly used in data storage and state machines.

2. T Flip-Flop (Toggle Flip-Flop)

Description:

A T Flip-Flop toggles its output state on each clock cycle when the T input is active. It acts as a frequency divider and is commonly used in counters.

Use Cases:

- Essential for designing counters, including binary and ripple counters.
- Used in frequency division applications.
- Forms the basis of many sequential logic circuits.

3. JK Flip-Flop

Description:

A JK Flip-Flop is an advanced version of the SR Flip-Flop, allowing both inputs to be high without causing an invalid state. It can function as a T Flip-Flop when both J and K inputs are high, toggling the output on every clock cycle.

Use Cases:

- Widely used in registers and sequential logic circuits.
- Commonly implemented in synchronous counters.
- Plays a role in various memory and control logic applications.

4. SR Flip-Flop (Set-Reset Flip-Flop)

Description:

An SR Flip-Flop controls the output state based on the Set (S) and Reset (R) inputs. When the Set input is active, the output is set to high, while the Reset input forces the output to low. It is a simple yet effective memory element.

Use Cases:

- Used in basic memory storage applications.
- Functions as a fundamental building block in control circuits.
- Forms the basis for more complex flip-flops such as JK and D Flip-Flops.

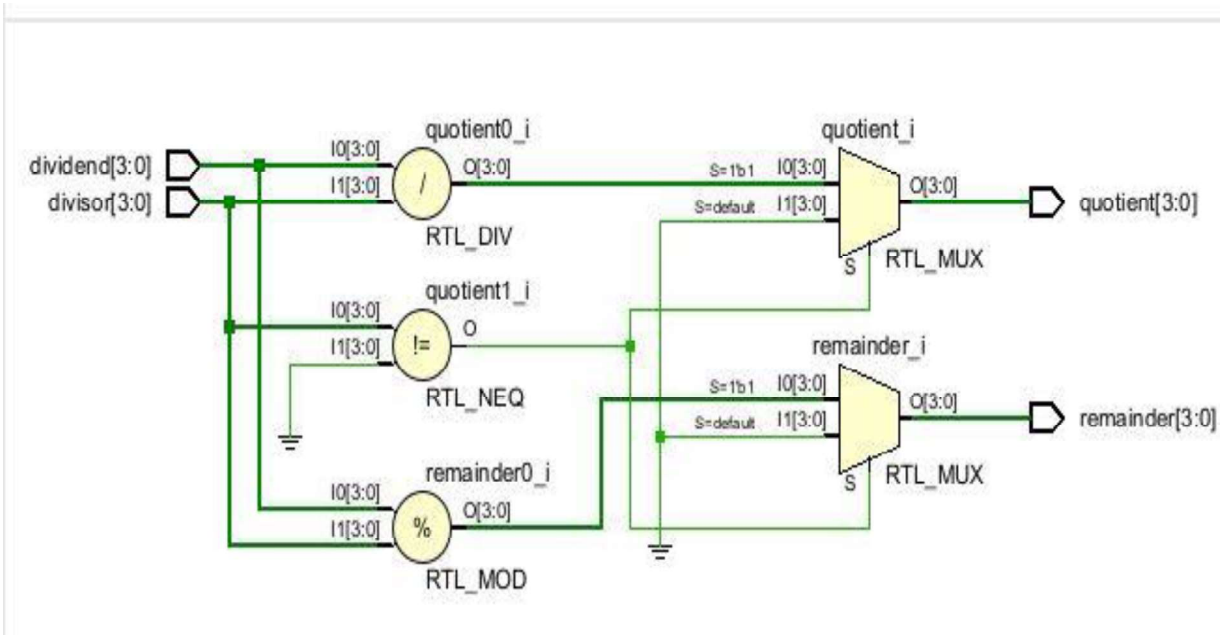
13.Divider

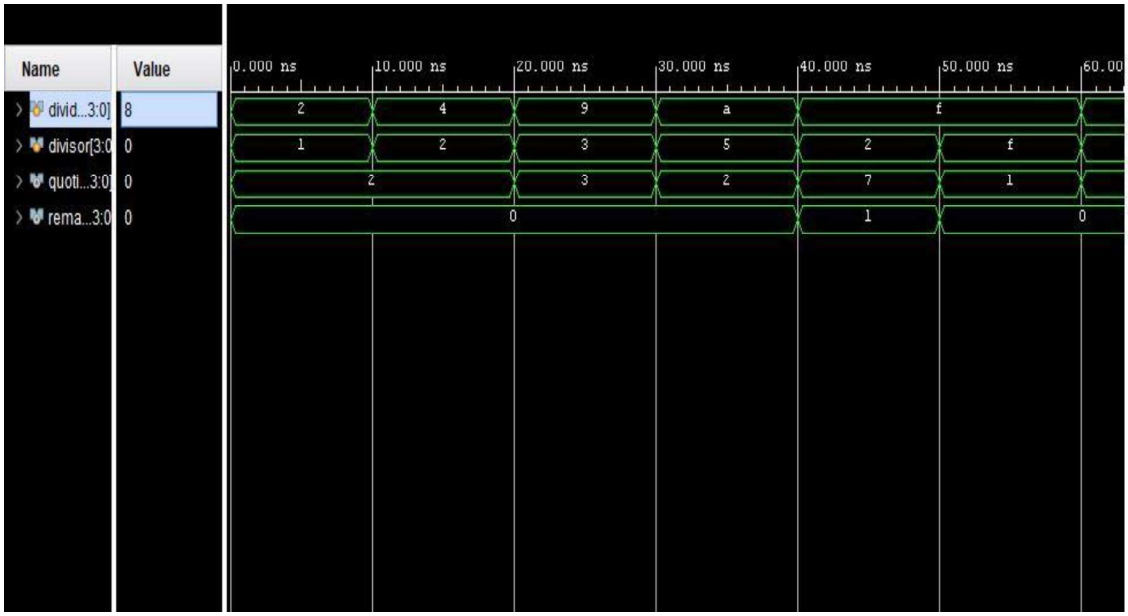
Code:

```
module divider_4bit (
    input [3:0] dividend,
    input [3:0] divisor,
    output reg [3:0] quotient,
    output reg [3:0] remainder
);

    always @ (dividend or divisor) begin
        if (divisor != 0) begin
            quotient = dividend / divisor;
            remainder = dividend % divisor;
        end else begin
            quotient = 4'b0000;
            remainder = 4'b0000;
        end
    end

endmodule
```

Testbench:

```
module tb_divider_4bit;

    reg [3:0] dividend, divisor;
    wire [3:0] quotient, remainder;

    divider_4bit uut (
        .dividend(dividend),
        .divisor(divisor),
        .quotient(quotient),
        .remainder(remainder)
    );

    initial begin
        $display("Dividend   Divisor   | Quotient  Remainder");
        $display("-----------------------------------------");

        dividend = 4'b0010; divisor = 4'b0001;
        #10; $display("%b         %b         | %b         %b", dividend, divisor, quotient, remainder);

        dividend = 4'b0100; divisor = 4'b0010;
        #10; $display("%b         %b         | %b         %b", dividend, divisor, quotient, remainder);

        dividend = 4'b1001; divisor = 4'b0011;
        #10; $display("%b         %b         | %b         %b", dividend, divisor, quotient, remainder);

        dividend = 4'b1010; divisor = 4'b0101;
        #10; $display("%b         %b         | %b         %b", dividend, divisor, quotient, remainder);

        dividend = 4'b1111; divisor = 4'b0010;
        #10; $display("%b         %b         | %b         %b", dividend, divisor, quotient, remainder);

        dividend = 4'b1111; divisor = 4'b1111;
        #10; $display("%b         %b         | %b         %b", dividend, divisor, quotient, remainder);

        dividend = 4'b1000; divisor = 4'b0000;
        #10; $display("%b         %b         | %b         %b", dividend, divisor, quotient, remainder);

        $finish;
    end

endmodule
```


dividend[3:0]
divisor[3:0]
quotient0_i
I0[3:0]
I1[3:0]
O[3:0]
/
RTL_DIV
quotient1_i
!=
RTL_NEQ
remainder0_i
%
RTL_MOD
quotient_i
S=1'b1
S=default
RTL_MUX
quotient[3:0]
remainder_i
remainder[3:0]


Name
Value
divid...3:0]
divisor[3:0]
quoti...3:0]
rema...3:0]
0.000 ns
10.000 ns
20.000 ns
30.000 ns
40.000 ns
50.000 ns

Description:

A **divider** is a digital circuit used to perform division operations on binary numbers. It can be implemented in two ways:

1. **Combinational Divider** – Uses **Restoring** or **Non-Restoring Division** methods to perform division in a single cycle (hardware-intensive).
2. **Sequential Divider** – Uses **Shift-Subtract Division** or **Multiplicative Algorithms** to perform division over multiple clock cycles (resource-efficient).

For dividing two numbers : Y=A/B

In **Xilinx**, a divider can be implemented using **Verilog or VHDL** and synthesized in **Vivado or ISE**.

- **Combinational dividers** use parallel subtractors and logic circuits, making them fast but hardware-intensive.
- **Sequential dividers** use shift registers and subtraction logic, trading speed for reduced hardware usage.
- FPGA implementations often use **DSP blocks** or **CORDIC algorithms** for efficient division in real-time applications.

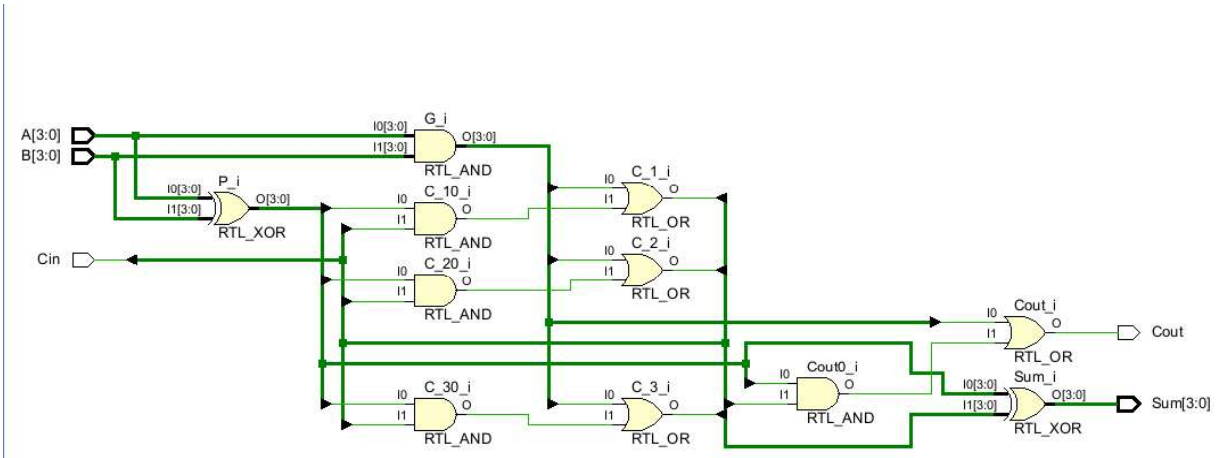
14. Parallel Adder

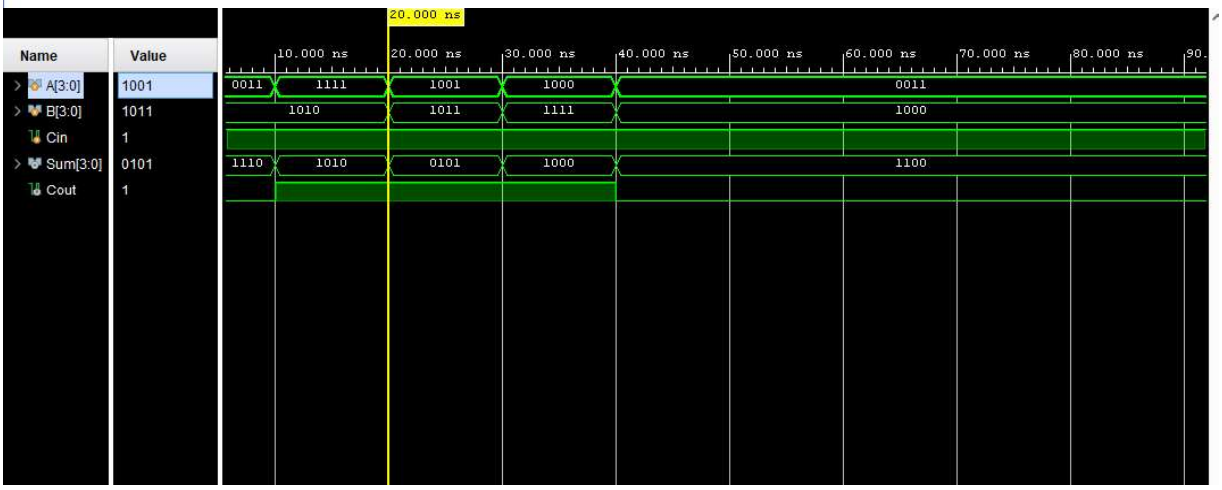
Code:

```
module carry_lookahead(
   input [3:0]A,B,
   input Cin,
   output [3:0]Sum,
   output Cout
   );
   wire [3:0]G,P;
   wire [3:0]C;
   assign G=A&B;
   assign P=A^B;
   assign C[0]=Cin;
   assign C[1]=(G[0]|P[0]&C[0]);
   assign C[2]=(G[1]|P[1]&C[1]);
   assign C[3]=(G[2]|P[2]&C[2]);
   assign Cout=(G[3]|P[3]&C[3]);
   assign Sum=P^C;

endmodule
```

Testbench:

```
module CLA_tb;
reg [3:0]A,B;
reg Cin;
wire [3:0]Sum;
wire Cout;
carry_lookahead uut(
.A(A),
.B(B),
.Cin(Cin),
.Sum(Sum),
.Cout(Cout)
);
initial begin
A=4'b0011;B=4'b1010;Cin=1;#10;
A=4'b1111;B=4'b1010;Cin=1;#10;
A=4'b1001;B=4'b1011;Cin=1;#10;
A=4'b1000;B=4'b1111;Cin=1;#10;
A=4'b0011;B=4'b1000;Cin=1;#10;

end
endmodule
```


A[3:0]
B[3:0]
Cin
G_i
I0[3:0]
I1[3:0]
O[3:0]
RTL_AND
P_i
RTL_XOR
C_10_i
C_20_i
C_30_i
C_1_i
C_2_i
C_3_i
RTL_OR
Cout_i
Cout
Cout0_i
Sum_i
Sum[3:0]


Name
Value
A[3:0]
1001
B[3:0]
1011
Cin
1
Sum[3:0]
0101
Cout
1
20.000 ns
10.000 ns
30.000 ns
40.000 ns
50.000 ns
60.000 ns
70.000 ns
80.000 ns
0011
1111
1001
1000
1010
1011
1110
0101
1100

15. Serial Adder

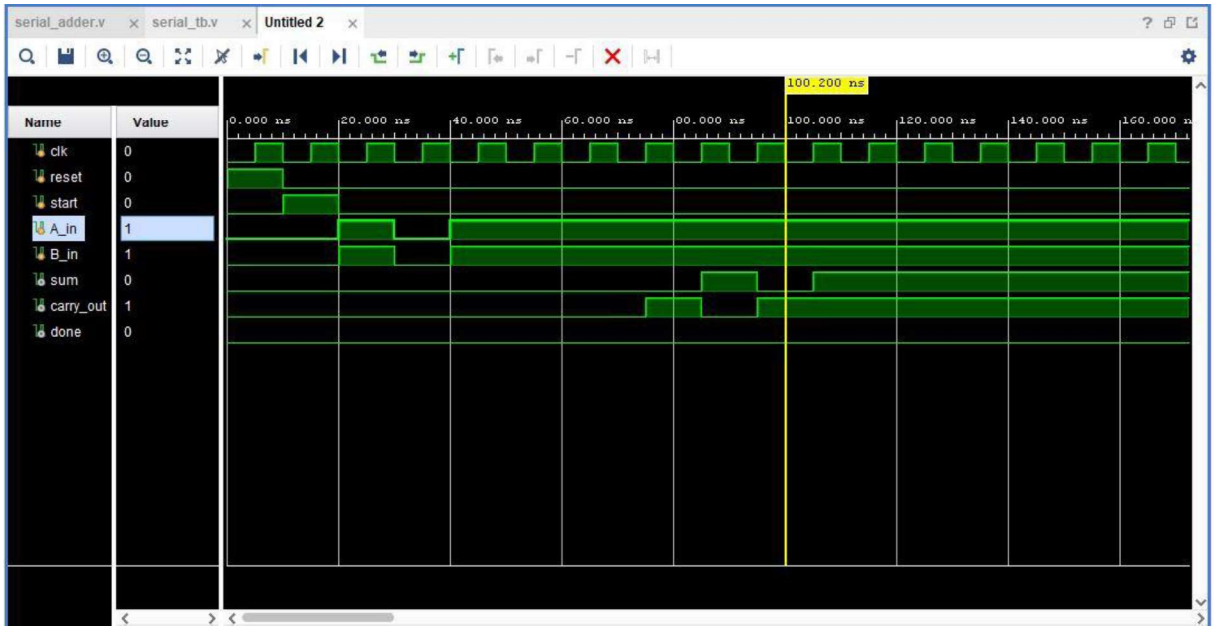
Code:

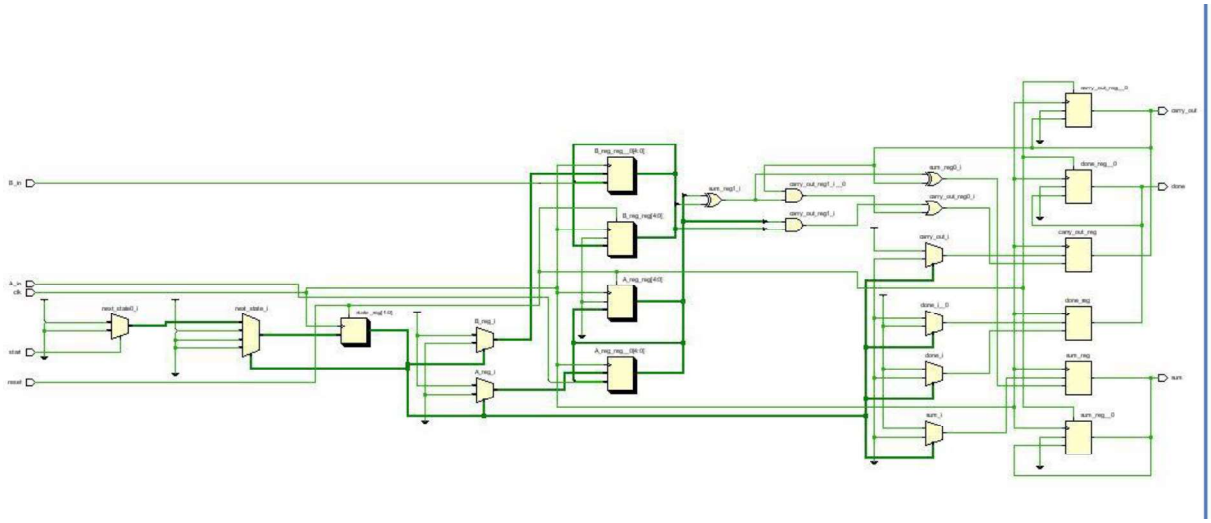
```
module serial_adder_4bit (
input clk,
input reset,
input start,
input A_in,
input B_in,
output reg sum,
output reg carry_out,
output reg done
);
reg [1:0] bit_count;
reg [4:0] A_reg, B_reg;
reg [1:0] state, next_state;
parameter IDLE = 2'b00, ADD = 2'b01, DONE = 2'b10;
always @(posedge clk or posedge reset) begin
if (reset) begin
state <= IDLE;
bit_count <= 0;
A_reg <= 0;
B_reg <= 0;
sum <= 0;
carry_out <= 0;
done <= 0;
end else begin
state <= next_state;
```

```
end

end

always @(*) begin

case (state)

IDLE: next_state = start ? ADD : IDLE;

ADD: next_state = (bit_count == 4) ? DONE : ADD;

DONE: next_state = IDLE;

default: next_state = IDLE;

endcase

end

always @(posedge clk) begin

if (state == ADD) begin

A_reg <= {A_in, A_reg[4:1]};

B_reg <= {B_in, B_reg[4:1]};

sum <= A_reg[0] ^ B_reg[0] ^ carry_out;

carry_out <= (A_reg[0] & B_reg[0]) | (carry_out & (A_reg[0] ^ B_reg[0]));

bit_count <= bit_count + 1;

end else if (state == DONE) begin

done <= 1;

end else begin

done <= 0;

end

end
```

Testbench:

```
module tb_serial_adder_4bit;

    reg clk;
    reg reset;
    reg start;
    reg A_in, B_in;
    wire sum;
    wire carry_out;
    wire done;

    serial_adder_4bit uut (
        .clk(clk),
        .reset(reset),
        .start(start),
        .A_in(A_in),
        .B_in(B_in),
        .sum(sum),
        .carry_out(carry_out),
        .done(done)
    );

    always begin
        #5 clk = ~clk;
    end

    initial begin
        clk = 0;
        reset = 0;
        start = 0;
        A_in = 0;
        B_in = 0;

        reset = 1;
        #10;
        reset = 0;

        start = 1;
        #10;
        start = 0;

        A_in = 1; B_in = 1; #10;
        A_in = 0; B_in = 0; #10;
        A_in = 1; B_in = 1; #10;
        A_in = 1; B_in = 1; #10;

        #20;
        if (done) $display("Addition is done. Sum: %b, Carry Out: %b", sum, carry_out);
    end

    initial begin
        $monitor("At time %t, A_in = %b, B_in = %b, Sum = %b, Carry Out = %b, Done = %b", $time, A_in, B_in, sum,
carry_out, done);
    end

endmodule
```





Parallel Adder vs. Serial Adder Using Xilinx

1. Parallel Adder

- A **Parallel Adder** adds multiple bits simultaneously using multiple full adders.
- Common types include **Ripple Carry Adder (RCA), Carry Look-Ahead Adder (CLA), and Carry Skip Adder (CSA)**.
- Faster than a serial adder because all bits are processed in parallel.
- Requires more hardware resources (more full adders).

Equation:

S=A+BS = A + BS=A+B

(where AAA and BBB are multi-bit numbers).

Use Cases: High-speed ALUs, DSP applications, processors.

2. Serial Adder

- A **Serial Adder** adds numbers **one bit at a time** over multiple clock cycles.
- Uses a **single full adder** with a shift register to process bits sequentially.
- Requires a **clock signal** for step-by-step addition.
- Slower but consumes less hardware (ideal for low-power applications).

Working:

1. LSBs are added first, with carry stored in a **flip-flop**.
2. Carry is used in the next cycle with the next higher bit.
3. Repeats until all bits are processed.

Use Cases: Low-power microcontrollers, serial communication, embedded systems

Feature	Parallel Adder	Serial Adder
Speed	Fast(all bits at once)	Slow(one bit at a time)
Hardware	Requires multiple full adders	Requires one full adder and shift registers
Efficiency	High Speed but more area	Slow speed but less area
Use cases	High speed processors, DSPs	Embedded systems, Low power applications