Lab 1: 4-bit Adder

Aim: To write a Verilog code for 4bit adder and verify the functionality using Test bench.

Synthesize, Analyse Reports and Netlist, Critical Path and Max Operating Frequency.

From the report generated find the total number of cells, power requirement and total area requirement.

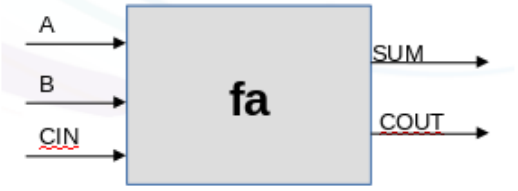
Tool Required:

Functional Simulation: Incisive Simulator (ncvlog, ncelab, ncsim)

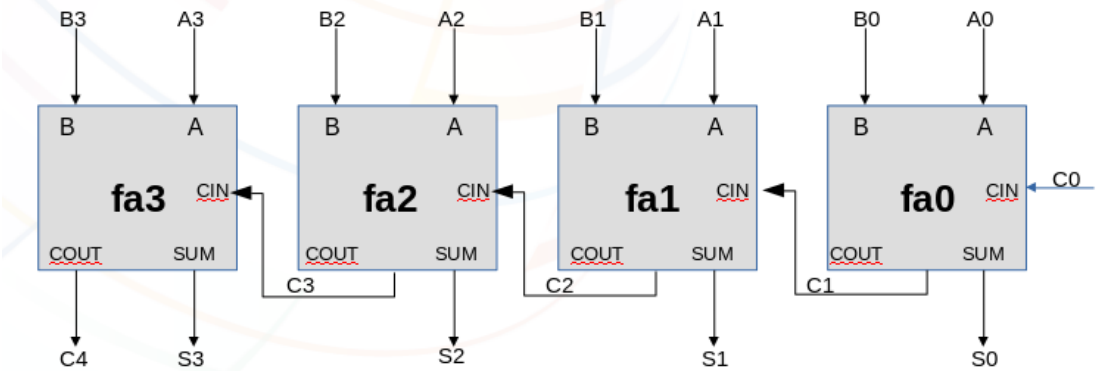
Synthesis: Genus

Design Information and Bock Diagram:

A full adder is a combinational circuit that performs the arithmetic sum of three input bits Ai, addend Bi and carry in C in from the previous adder. Its results contain the sum Si and the carry out, C out to the next stage. So to design a 4-bit adder circuit we start by designing the 1 –bit full adder then connecting the four 1-bit full adders to get the 4-bit adder as shown in the diagram below. For the 1-bit full adder, the design begins by drawing the Truth Table for the three input and the corresponding output SUM and CARRY.



A	B	C	SUM OUT	CARRY OUT
0	0	0	0	0
0	0	1	1	0
0	1	0	1	0
0	1	1	0	1
1	0	0	1	0
1	0	1	0	1
1	1	0	0	1
1	1	1	1	1



Source code:

```
module four_bit_adder(A,B,C0,S,C4);
input[3:0]A,B;
input C0;
output[3:0]S;
output C4;
wire C1,C2,C3;
full_adder fa0 (A[0],B[0],C0,S[0],C1);
full_adder fa1 (A[1],B[1],C1,S[1],C2);
full_adder fa2 (A[2],B[2],C2,S[2],C3);
full_adder fa3 (A[3],B[3],C3,S[3],C4);
endmodule

module full_adder(A,B,CIN,S,COUT);
input A,B,CIN;
output S,COUT; assign S = A^B^CIN;
assign COUT = (A&B) | (CIN&(A^B));
endmodule
```

Testbench code:

```
module test_4_bit;
reg [3:0] A;
reg [3:0] B; reg C0;
wire [3:0] S; wire C4;
four_bit_adder dut (A,B,C0,S,C4); initial begin
A = 4'b0011; B=4'b0011; C0 = 1'b0;
#10; A = 4'b1011; B=4'b0111;C0 = 1'b1;
#10; A = 4'b1111; B=4'b1111; C0 = 1'b1;
#10;
end initial
#50 $finish;
endmodule
```

Input Constraints code:

```
create_clock -name clk -period 10
set_input_delay 2 -clock clk [get_ports "A"]
set_input_delay 2 -clock clk [get_ports "B"]
set_input_delay 2 -clock clk [get_ports "CO"]
set_output_delay 2 -clock clk [get_ports "S"]
set_output_delay 2 -clock clk [get_ports "C4"]
```

Script code:

```
read_libs /home/install/FOUNDRY/digital/90nm/dig/lib/slow.lib
read_hdl source.v
elaborate
read_sdc input_constraints.sdc
set_db syn_generic_effort medium
set_db syn_map_effort medium
set_db syn_opt_effort medium
syn_generic
syn_map
syn_opt
report_timing > timing.rep
report_area > area.rep
report_power > power.rep
write_hdl > netlist.v
write_sdc > output_constaints.sdc
report_gates > gates.v
gui_show
```

Lab 2: 4-bit Up/Down Counter

Aim: To write a Verilog code for 4-bit Up/Down Counter and verify the functionality using Test bench.

Synthesize, Analyse Reports and Netlist, Critical Path and Max Operating Frequency.

From the report generated find the total number of cells, power requirement and total area requirement.

Tool Required:

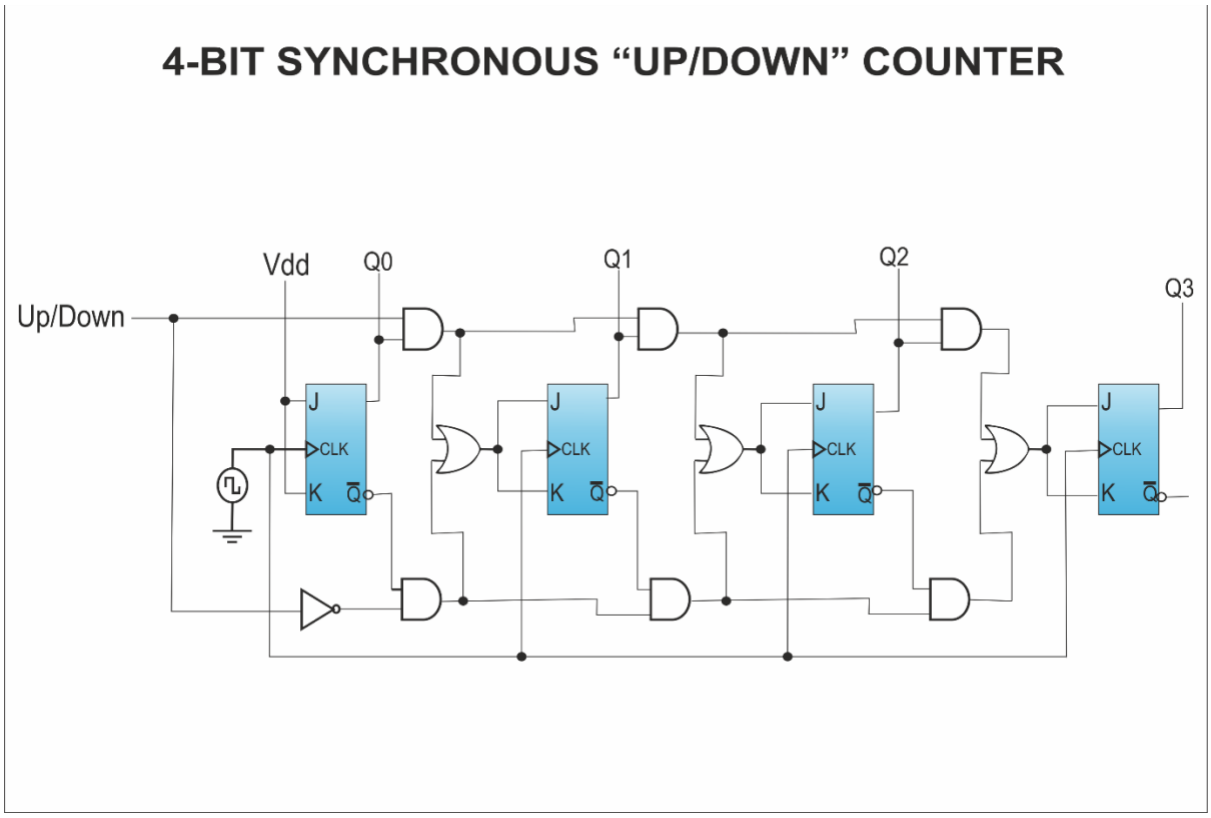
Functional Simulation: Incisive Simulator (ncvlog, ncelab, ncsim)

Synthesis: Genus

Design Information and Bock Diagram:

Bidirectional counters, also known as Up/Down counters, are capable of counting in either direction through any given count sequence and they can be reversed at any point within their count sequence by using an additional control input.

As well as counting “up” from zero and increasing or incrementing to some preset value, it is sometimes necessary to count “down” from a predetermined value to zero allowing us to produce an output that activates when the zero count or some other pre-set value is reached.



Source code:

```
module counter(clk,m,rst,count);
input clk,m,rst;
output reg [3:0] count;
always@(posedge clk or negedge rst)
begin
if(!rst)
count=0;
else if(m)
count=count+1;
else
count=count-1;
end
endmodule
```

Testbench code:

```
module counter_test;
reg clk,rst,m;
wire[3:0] count;
initial
begin
clk=0;
rst=0; #5;
rst=1;
rst=0; #265;
rst=1;
end
initial
begin
m=1;
#160 m=0;
#160m=1;
end
counter counter1(clk,m,rst,count);
always #5 clk=~clk;
initial
$monitor("Time=%t rst=%b clk=%b count=%b", $time,rst,clk,count);
initial
#820 $finish;
endmodule
```

Input Constraints code:

```
create_clock -name clk -period 2 -waveform {0 1} [get_ports "clk"]
set_clock_transition -rise 0.1 [get_clocks "clk"]
set_clock_transition -fall 0.1 [get_clocks "clk"]
set_clock_uncertainty 0.01 [get_ports "clk"]
set_input_transition 0.12 [all_inputs]
set_input_delay -max 0.8 [get_ports "clk"] -clock [get_clock "clk"]
set_input_delay -max 0.8 [get_ports "reset"] -clock [get_clock "clk"]
set_input_delay -max 0.8 [get_ports "m"] -clock [get_clock "clk"]
set_output_delay -max 0.8 [get_ports "count"] -clock [get_clock "clk"]
set_load 0.15 [all_outputs]
set_max_fanout 20.00 [current_design]
```

Script code:

```
read_libs /home/install/FOUNDRY/digital/90nm/dig/lib/slow.lib
read_hdl source.v
elaborate
read_sdc input_constraints.sdc
set_db syn_generic_effort medium
set_db syn_map_effort medium
set_db syn_opt_effort medium
syn_generic
syn_map
syn_opt
report_timing > timing.rep
report_area > area.rep
report_power > power.rep
write_hdl > netlist.v
write_sdc > output_constaints.sdc
report_gates > gates.v
gui_show
```

Lab 3: 32-bit ALU

Aim: Write a Verilog code for 32-bit ALU supporting four logical and four arithmetic operations, use case statement and if statement for ALU behavioural modelling.

To Verify the Functionality using Test Bench

Identify Critical Path and constraints

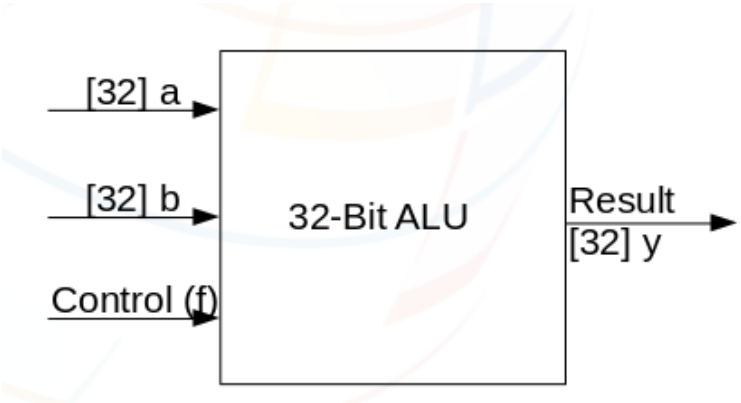
Tool Required:

Functional Simulation: Incisive Simulator (ncvlog, ncelab, ncsim)

Synthesis: Genus

Design Information and Bock Diagram:

The ALU will take in two 32-bit values, and control line. An Arithmetic unit does the following task like addition subtraction, multi-fiction and logical operations. As the input is given in 32 bit we get 32-bit output. The arithmetic will show only one output at a time so a selector is necessary to select one of the operators.



Source Code :

```
module alu_32bit_case(y,a,b,f);
input [31:0]a;
input [31:0]b;
input [2:0]f;
output reg [31:0]y;
always@(*)
begin
	case(f)
		3'b000:y=a&b; //AND Operation
		3'b001:y=a|b; //OR Operation
		3'b010:y=~(a&b); //NAND Operation
		3'b011:y=~(a|b); //NOR Operation
		3'b010:y=a+b; //Addition
		3'b011:y=a-b; //Subtraction
		3'b100:y=a*b; /Multiply
		default:y=32'bx;
	endcase
end
endmodule
```

Testbench code:

```
module alu_32bit_tb_case;
reg [31:0]a;
reg [31:0]b;
reg [2:0]f;
wire [31:0]y;
alu_32bit_case test2(.y(y),.a(a),.b(b),.f(f));
initial
begin
	a=32'h00000000;
	b=32'hFFFFFFFF;
	#10 f=3'b000;
	#10 f=3'b001;
	#10 f=3'b010;
	#10 f=3'b011;
	#10 f=3'b010;
	#10 f=3'b011;
	#10 f=3'b100;
end
initial
#100 $finish;
endmodule
```

Input Constraints:

```
create_clock -name clk -period 10
set_input_delay 2 -clock clk [get_ports a]
set_input_delay 2 -clock clk [get_ports b]
set_input_delay 2 -clock clk [get_ports f]
set_output_delay 2 -clock clk [get_ports y]
```

Script code:

```
read_libs /home/install/FOUNDRY/digital/90nm/dig/lib/slow.lib
read_hdl source.v
elaborate
read_sdc input_constraints.sdc
set_db syn_generic_effort medium
set_db syn_map_effort medium
set_db syn_opt_effort medium
syn_generic
syn_map
syn_opt
report_timing > timing.rep
report_area > area.rep
report_power > power.rep
write_hdl > netlist.v
write_sdc > output_constaints.sdc
report_gates > gates.v
gui_show
```

Lab 4: 4-Bit Shift and Add Multiplier

Aim: To write a Verilog code for 4-Bit Shift and Add Multiplier and verify the functionality using Test bench.

Synthesize, Analyse Reports and Netlist, Critical Path and Max Operating Frequency.

From the report generated, find the total number of cells, power requirement and total area requirement.

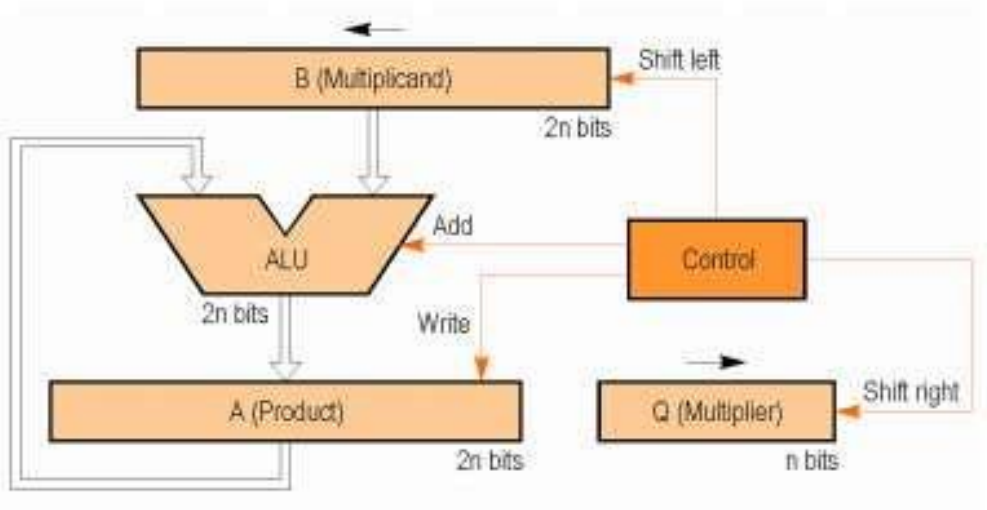
Tool Required:

Functional Simulation: Incisive Simulator (ncvlog, ncelab, ncsim)

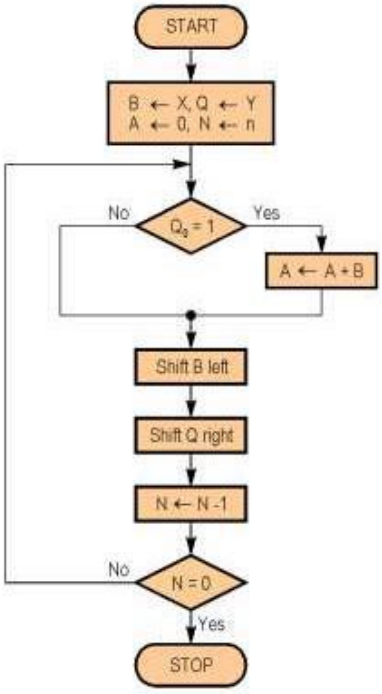
Synthesis: Genus

Design Information and Flow Chart:

Binary multipliers are used for multiplication of 2 binary numbers and are used mainly in signal processing and also in other computationally intensive applications. Shift and add binary multiplier is a type of sequential multiplier. Sequential multipliers generate the partial products sequentially and add each newly generated partial product to the previously accumulated sum. Shift and add binary multiplier is a type of sequential multiplier.



First version of the multiplier circuit.



Shift and add multiplier is similar to multiplication done by paper and pencil. This method adds the multiplicand X to itself Y times, where Y denotes the multiplier. In the case of binary multiplication, since the digits are 0 and 1, if the multiplier digit is 1, a copy of the multiplicand is placed in the proper positions; if the multiplier digit is 0, a number of 0 digits are placed. The 2n-bit product register (A) is initialized to 0. A 2n-bit multiplicand register with the multiplicand placed in the right half of the register and with 0 in the left half is used. The algorithm starts by loading the multiplicand into the B register, loading the multiplier into the Q register, and initializing the A register to 0. The counter N is initialized to n. The least significant bit of the multiplier register (Q0) determines whether the multiplicand is added to the product register. The left shift of the multiplicand has the effect of shifting the intermediate products to the left and right shift prepares the next bit of the multiplier to examine in the following iteration.

Source code:

```
`timescale 1ns/ 1ns
module shift_and_add_binary_multiplier(clk,rst,A,B,C);
parameter m=4, n=4;
integer i;
input clk,rst;
input [m-1:0] A;
input [n-1:0] B;
output reg [m+n-1:0]C;
reg [m+n-1:0]A1;
reg [n-1:0]B1;
always@(posedge clk or posedge rst)
begin
		if (rst)
		begin
		C=0;
		end
		else
		begin
		C=0;
		A1 [m-1:0]=A;
		A1 [m+n-1:m]=0;
		B1=B;
	for(i=0;i<n;i=i+1)
	begin
			if (B1[i]==1'b0)
			begin
				C=C+0;
			end
			else if (B1[i]==1'b1)
			begin
				C=C+(A1<<i);
			end
		end
		end
end
endmodule
```

Testbench code:

```
`timescale 1ns/1ns
module shift_and_add_binary_multiplier_tb;
parameter m=4, n=4;
reg clk, rst;
reg [m-1:0] A;
reg [n-1:0] B;
wire [m+n-1:0] C;

shift_and_add_binary_multiplier uut(clk, rst, A, B,C);
initial
begin
clk = 1'b1;
forever #4 clk = ~clk;
end
initial
begin
            rst = 1;
      #2 rst = 0;
        A = 4'b1111;
            B = 4'b1111;

          #20 rst = 1;
          #2  rst = 0;

            A = 4'b0011;
            B = 4'b0011;
          #20 rst = 1;
          #2  rst = 0;
            A = 4'b1100;
            B = 4'b0010;
          #20;
     end
initial
begin #100 $finish;
end
endmodule
```

Input Constraints:

```
create_clock -name clk -period 2 -waveform {0 1} [get_ports "clk"]
set_clock_transition -rise 0.1 [get_clocks "clk"]
set_clock_transition -fall 0.1 [get_clocks "clk"]
set_clock_uncertainty 0.01 [get_ports "clk"]
set_input_delay -max 1.0 [get_ports "rst"] -clock [get_clocks "clk"]
set_input_delay -max 1.0 [get_ports "A"] -clock [get_clocks "clk"]
set_input_delay -max 1.0 [get_ports "B"] -clock [get_clocks "clk"]
set_output_delay -max 1.0 [get_ports "C"] -clock [get_clocks "clk"]
```

Script code:

```
read_libs /home/install/FOUNDRY/digital/90nm/dig/lib/slow.lib
read_hdl source.v
elaborate
read_sdc input_constraints.sdc
set_db syn_generic_effort medium
set_db syn_map_effort medium
set_db syn_opt_effort medium
syn_generic
syn_map
syn_opt
report_timing > timing.rep
report_area > area.rep
report_power > power.rep
write_hdl > netlist.v
write_sdc > output_constaints.sdc
report_gates > gates.v
gui_show
```

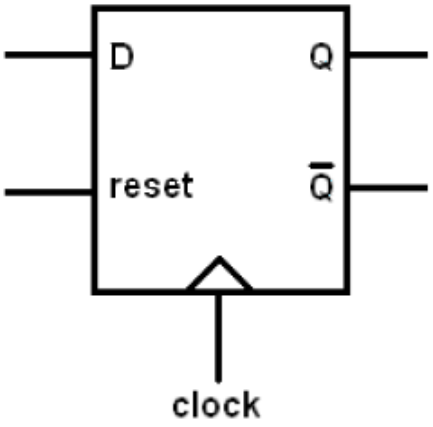
Lab 5: Flip-Flops

Aim: Write a Verilog code for Flip-flops (D, SR, JK), Synthesize the design and compare the synthesis report.

Tool Required:
Functional Simulation: Incisive Simulator (ncvlog, ncelab, ncsim)
Synthesis: Genus

Design Information and Bock Diagram:
Latches and flip-flops are the basic elements for storing information. One latch or flip-flop can store one bit of information. The main difference between latches and flip-flops is that for latches, their outputs are constantly affected by their inputs as long as the enable signal is asserted.
In other words, when they are enabled, their content changes immediately when their inputs change. Flip-flops, on the other hand, have their content change only either at the rising or falling edge of the enable signal. This enable signal is usually the controlling clock signal. After the rising or falling edge of the clock, the flip-flop content remains constant even if the input changes.
There are basically three main types of latches and flip-flops: SR, D, and JK. The major differences in these flip-flop types are the number of inputs they have and how they change state. For each type, there are also different variations that enhance their operations.

D Flip-Flop:



Input			Output	
D	reset	clock	Q	Q'
0	0	0	0	1
0	0	1	0	1
0	1	0	0	1
0	1	1	0	1
1	0	0	0	1
1	0	1	1	0
1	1	0	0	1
1	1	1	0	1

Source code:

```
module DFF(Q,Qbar,D,clk,Reset);
output reg Q;
output Qbar;
input D,clk,Reset;
always @(posedge clk)
begin
if (Reset == 1'b1)
Q <= 1'b0;
else
Q <= D;
end
assign Qbar = ~Q;
endmodule
```

Testbench:

```
`timescale 1ns/1ps
module DFF_tb;
 reg D, clk, Reset;
 wire Q, Qbar;
 DFF uut ( .Q(Q), .Qbar(Qbar), .D(D),  .clk(clk), .Reset(Reset) );
 initial begin
   clk = 0;
   forever #5 clk = ~clk
 end
 initial begin
   $monitor("Time=%0t | Reset=%b | D=%b | Q=%b | Qbar=%b", $time, Reset, D, Q, Qbar);
   // Initialize inputs
   D = 0; Reset = 1;
   #10;
   Reset = 0;
   #10 D = 1;
   #10 D = 0;
   #10 D = 1;
   #10 Reset = 1;
   #10 Reset = 0;
   #20 $finish;
 end
endmodule
```

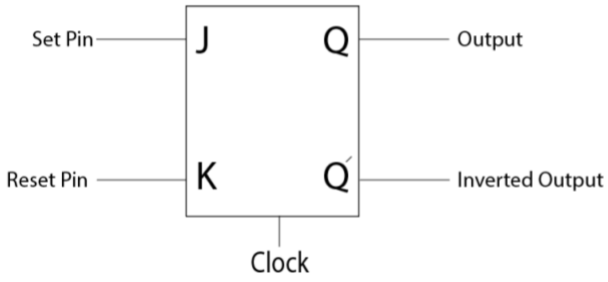
Input Constraints:

```
create_clock -name clk -period 2 -waveform {0 1} [get_ports "clk"]
set_clock_transition -rise 0.1 [get_clocks "clk"]
set_clock_transition -fall 0.1 [get_clocks "clk"]
set_clock_uncertainty 0.01 [get_ports "clk"]
set_input_delay -max 0.8 [get_ports "D"] -clock [get_clock "clk"]
set_input_delay -max 0.8 [get_ports "Reset"] -clock [get_clock "clk"]
set_input_delay -max 0.8 [get_ports "Q"] -clock [get_clock "clk"]
set_output_delay -max 0.8 [get_ports "Qbar"] -clock [get_clock "clk"]
```

Script code:

```
read_libs /home/install/FOUNDRY/digital/90nm/dig/lib/slow.lib
read_hdl source.v
elaborate
read_sdc input_constraints.sdc
set_db syn_generic_effort medium
set_db syn_map_effort medium
set_db syn_opt_effort medium
syn_generic
syn_map
syn_opt
report_timing > timing.rep
report_area > area.rep
report_power > power.rep
write_hdl > netlist.v
write_sdc > output_constaints.sdc
report_gates > gates.v
gui_show
```

JK Flip-Flop:



C	J	K	Q	Q`
HIGH	0	0	Latch	Latch
HIGH	0	1	0	1
HIGH	1	0	1	0
HIGH	1	1	Toggle	Toggle
LOW	0	0	Latch	Latch
LOW	0	1	Latch	Latch
LOW	1	0	Latch	Latch
LOW	1	1	Latch	Latch

Source code:

```
module jkff( input J, input K, input clk, output reg Q, output Qbar );
  assign Qbar = ~Q;
  always @(posedge clk) begin
    if (J == 1 && K == 0)
      Q <= 1;
    else if (J == 0 && K == 1)
      Q <= 0;
    else if (J == 1 && K == 1)
      Q <= ~Q;
  end
endmodule
```

Testbench code:

```
module jkff_tb;
  reg J,K,clk;
  wire Q, Qbar;
  jkff uut (  .Q(Q),  .Qbar(Qbar),  .J(J),  .K(K),  .clk(clk)  );
  initial begin
    clk = 0;
    forever #5 clk = ~clk
  end
  initial begin
    $monitor("Time=%0t | J=%b |  K=%b | Q=%b | Qbar=%b", $time, J, K, Q, Qbar);
    #10 J = 0; K = 1;
    #10 J = 0; K = 0;
    #10 J = 1; K = 0;
    #10 J = 1; K = 1;
    #20 $finish;
  end
endmodule
```

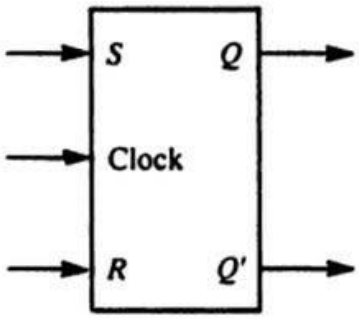
Input Constraints:

```
create_clock -name clk -period 2 -waveform {0 1} [get_ports "clk"]
set_clock_transition -rise 0.1 [get_clocks "clk"]
set_clock_transition -fall 0.1 [get_clocks "clk"]
set_clock_uncertainty 0.01 [get_ports "clk"]
set_input_delay -max 0.8 [get_ports "J"] -clock [get_clock "clk"]
set_input_delay -max 0.8 [get_ports "K"] -clock [get_clock "clk"]
set_output_delay -max 0.8 [get_ports "Q"] -clock [get_clock "clk"]
set_output_delay -max 0.8 [get_ports "Qbar"] -clock [get_clock "clk"]
```

Script code:

```
read_libs /home/install/FOUNDRY/digital/90nm/dig/lib/slow.lib
read_hdl source.v
elaborate
read_sdc input_constraints.sdc
set_db syn_generic_effort medium
set_db syn_map_effort medium
set_db syn_opt_effort medium
syn_generic
syn_map
syn_opt
report_timing > timing.rep
report_area > area.rep
report_power > power.rep
write_hdl > netlist.v
write_sdc > output_constaints.sdc
report_gates > gates.v
gui_show
```

SR Flip-Flop:



INPUTS			OUTPUT	STATE
CLK	S	R	Q	
X	0	0	No Change	Previous
↑	0	1	0	Reset
↑	1	0	1	Set
↑	1	1	-	Forbidden

Source code:

```
module srff( input S, input R, input clk, output reg Q, output Qbar );
  assign Qbar = ~Q;
  always @(posedge clk) begin
    if (S == 1 && R == 0)
      Q <= 1;
    else if (S == 0 && R == 1)
      Q <= 0;
    else if (S == 1 && R == 1)
      Q <= 1'bz;
  end
endmodule
```

Testbench code:

```
module srff_tb;
  reg S,R,clk;
  wire Q, Qbar;
  srff uut (  .Q(Q),  .Qbar(Qbar),  .S(S),  .R(R),  .clk(clk)  );
  initial begin
    clk = 0;
    forever #5 clk = ~clk;
  end
  initial begin
    $monitor("Time=%0t | S=%b |  R=%b | Q=%b | Qbar=%b", $time, S, R, Q, Qbar);
    #10 S = 0; R = 1;
    #10 S = 0; R = 0;
    #10 S = 1; R = 0;
    #10 S = 1; R = 1;
    #20 $finish;
  end
endmodule
```

Input Constraints:

```
create_clock -name clk -period 2 -waveform {0 1} [get_ports "clk"]
set_clock_transition -rise 0.1 [get_clocks "clk"]
set_clock_transition -fall 0.1 [get_clocks "clk"]
set_clock_uncertainty 0.01 [get_ports "clk"]
set_input_delay -max 0.8 [get_ports "S"] -clock [get_clock "clk"]
set_input_delay -max 0.8 [get_ports "R"] -clock [get_clock "clk"]
set_output_delay -max 0.8 [get_ports "Q"] -clock [get_clock "clk"]
set_output_delay -max 0.8 [get_ports "Qbar"] -clock [get_clock "clk"]
```

Script code:

```
read_libs /home/install/FOUNDRY/digital/90nm/dig/lib/slow.lib
read_hdl source.v
elaborate
read_sdc input_constraints.sdc
set_db syn_generic_effort medium
set_db syn_map_effort medium
set_db syn_opt_effort medium
syn_generic
syn_map
syn_opt
report_timing > timing.rep
report_area > area.rep
report_power > power.rep
write_hdl > netlist.v
write_sdc > output_constaints.sdc
report_gates > gates.v
gui_show
```