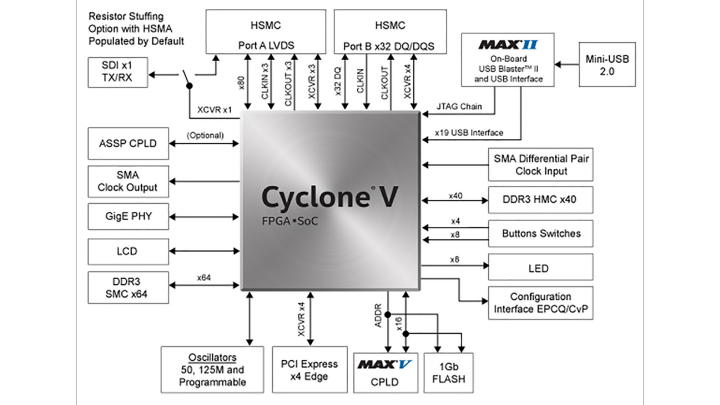
**EC-340**

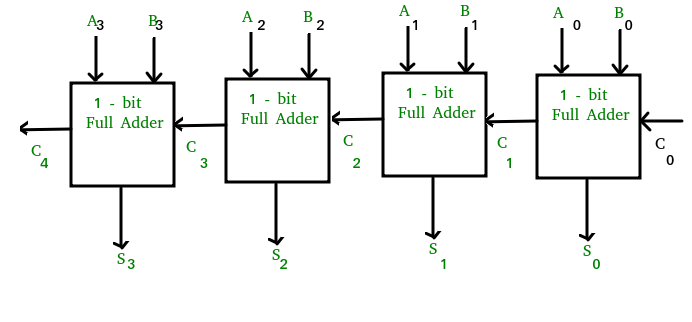
**Computer Organization and Architecture**



Utkarsh Mahajan 201EC164

**1. (a) 8 bit ripple carry adder**

It is a combinational circuit for adding 2 binary numbers requiring N full adders for N bit adder. Here for 8 bits, we will take 8 full adders, connect the cin to the first cin of first full adder and then cout of the first adder to the cin of the next and so on. While inputs a[i], b[i] to the i’th full adder. And take output of each full adder as s[i]. considering the cout of the last full adder as the output cout.



(Image from GFG)

Verilog Code for the module:

module ripple\_carry\_adder (input [7:0] a , b, input cin, output [7:0] s, output cout);

wire [8:0] c;

assign c[0] = cin;

assign cout = c[8];

genvar i;

    generate

    for (i=0; i<8; i= i+1) begin: full\_adders

        full\_adder fa(.a(a[i]), .b(b[i]), .cin(c[i]), .s(s[i]), .cout(c[i+1]));

    end

endgenerate

endmodule

Verilog code for the full adder:

module full\_adder(input a, b, cin, output s, cout);

assign {cout, s} = a + b + cin;

endmodule

Verilog code for the test bench:

module ripple\_carry\_adder (input [7:0] a , b, input cin, output [7:0] s, output cout);

wire [8:0] c;

assign c[0] = cin;

assign cout = c[8];

genvar i;

    generate

    for (i=0; i<8; i= i+1) begin: full\_adders

        full\_adder fa(.a(a[i]), .b(b[i]), .cin(c[i]), .s(s[i]), .cout(c[i+1]));

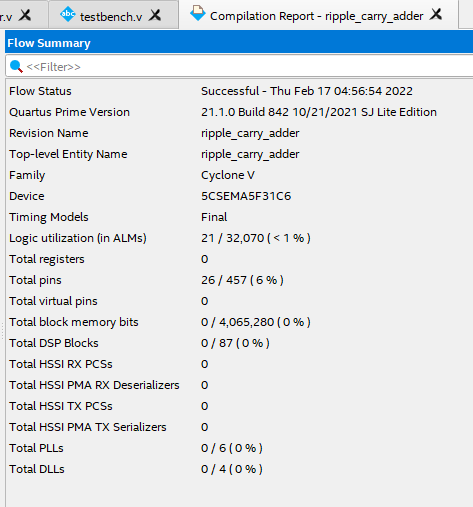
    end

endgenerate

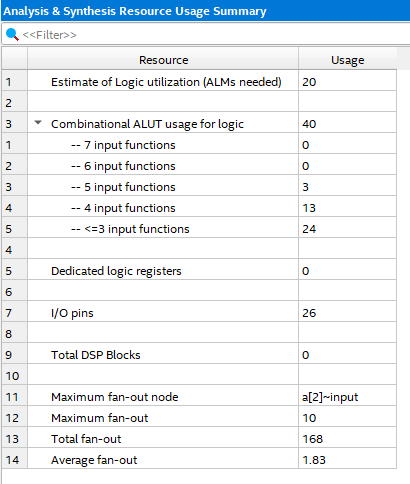
endmodule

Device Utilization Statistics:

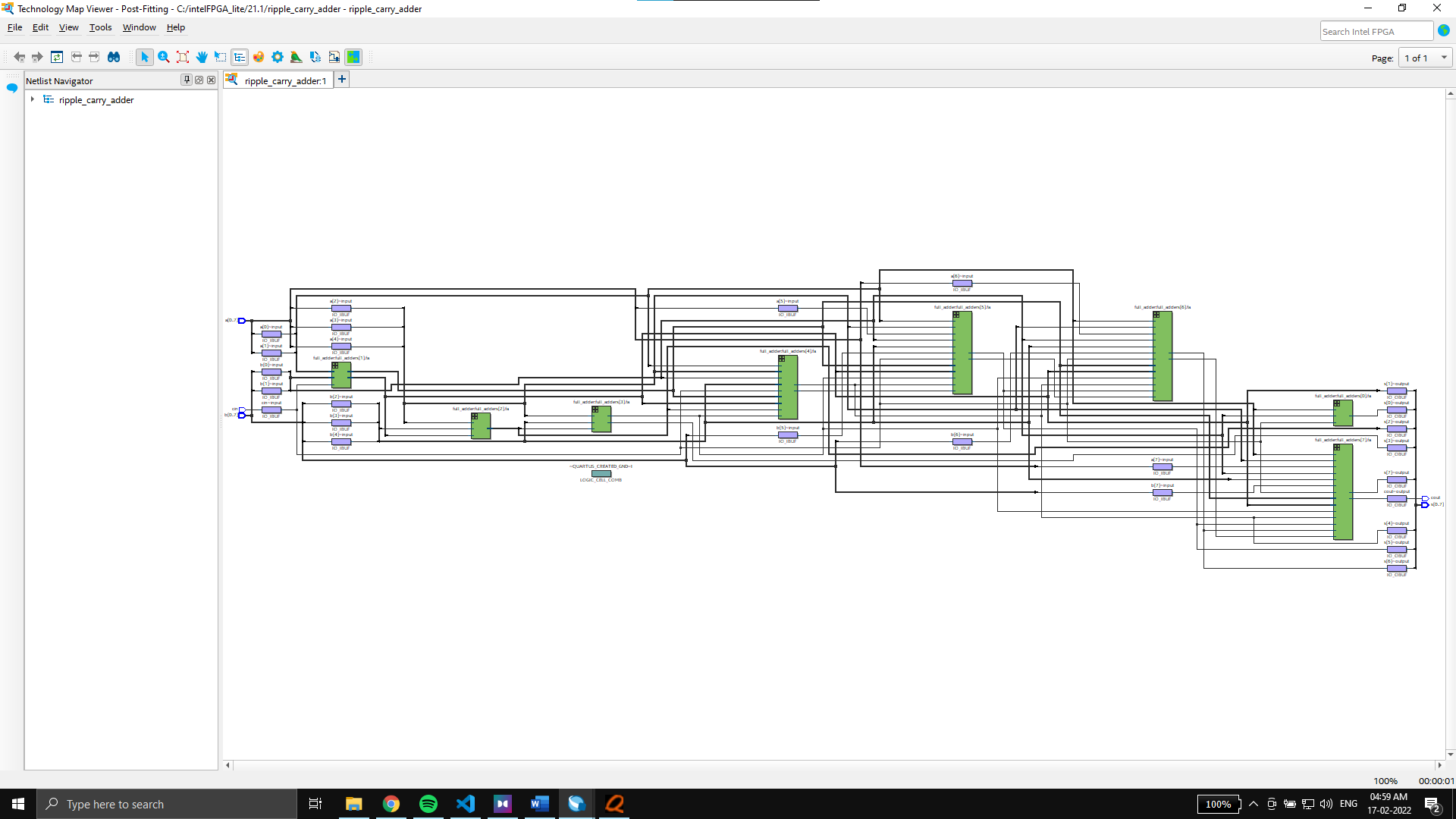
Flow summary:



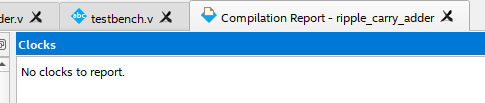
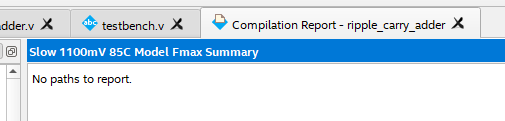
Resource usage summary:

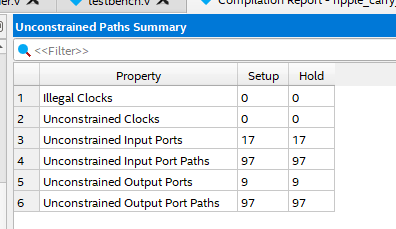
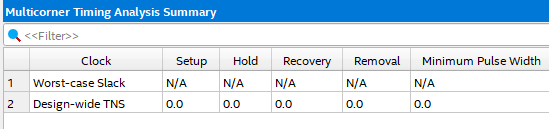


Block Diagram (Technology map viewer):

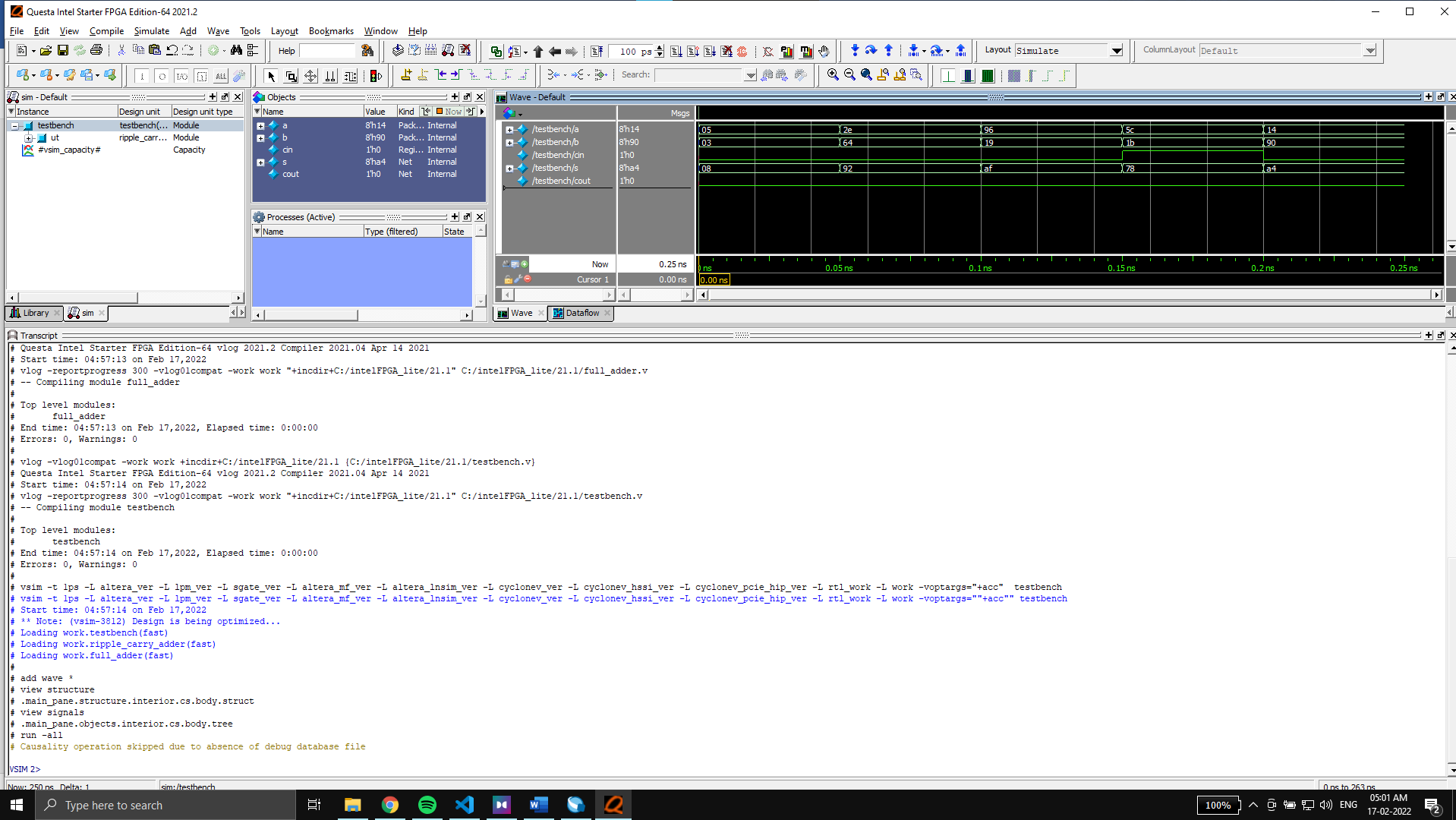


Timing reports:

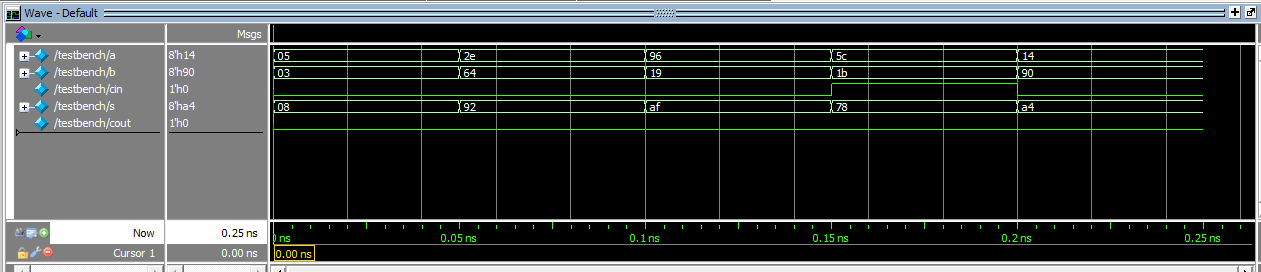
 

Simulation in Questa through Quartus:



Wave:

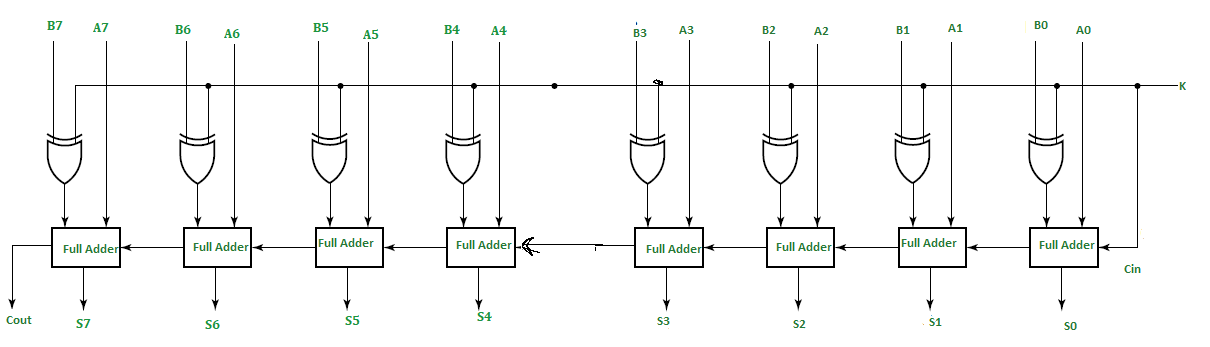
****

**1. (b) 8-bit controllable adder/subtractor**

An 8-bit controllable adder/subtractor is capable of both addition and subtraction of binary numbers. We will make use of 8 full adders and XOR gates for designing the circuit with Verilog.

We will have 3 inputs, A and B of width 8 bits. The addition or subtraction operation will be performed on them depending upon the third input K of size 1 bit.

If k=1, We will perform subtraction and if k=0, then we will perform addition.



(Image from GFG(edited))

Verilog Code for the module:

module add\_sub(input [7:0]a, b, input k, output [7:0]s, output cout);

wire [8:0] c;

genvar i;

assign c[0] = k;

assign cout = c[8];

generate

    for (i=0; i<8; i= i+1) begin: add\_sub

        full\_adder fa(.a(a[i]), .b(b[i]^k), .cin(c[i]), .s(s[i]), .cout(c[i+1]));

    end

endgenerate

endmodule

Verilog code for the full adder:

module full\_adder(input a, b, cin, output s, cout);

    assign {cout, s} = a + b + cin;

endmodule

Verilog code for the test bench:

module testbench();

reg [7:0] a, b;

reg k;

wire [7:0]  s;

wire cout;

add\_sub asc(a, b, k, s, cout);

initial begin

 a=5; b=3; k=0;

#50 a=46; b=100; k=0;

#50 a=150; b=25; k=0;

#50 a=8'b01011100; b=8'b00011011; k=1;

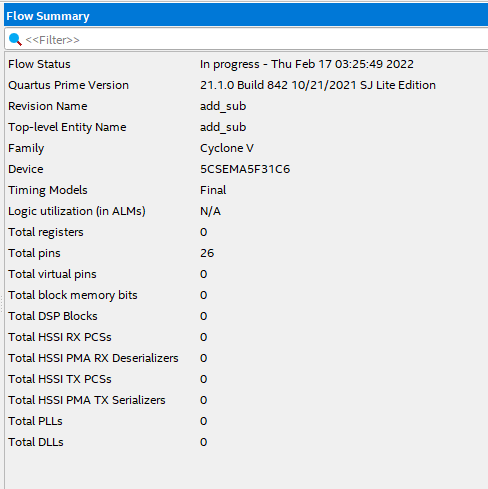
#50 a=20; b=8'b10010000; k=0; #50;

end

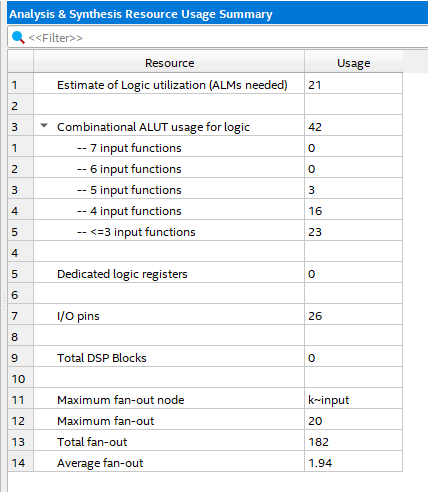
endmodule

Device Utilization Statistics:

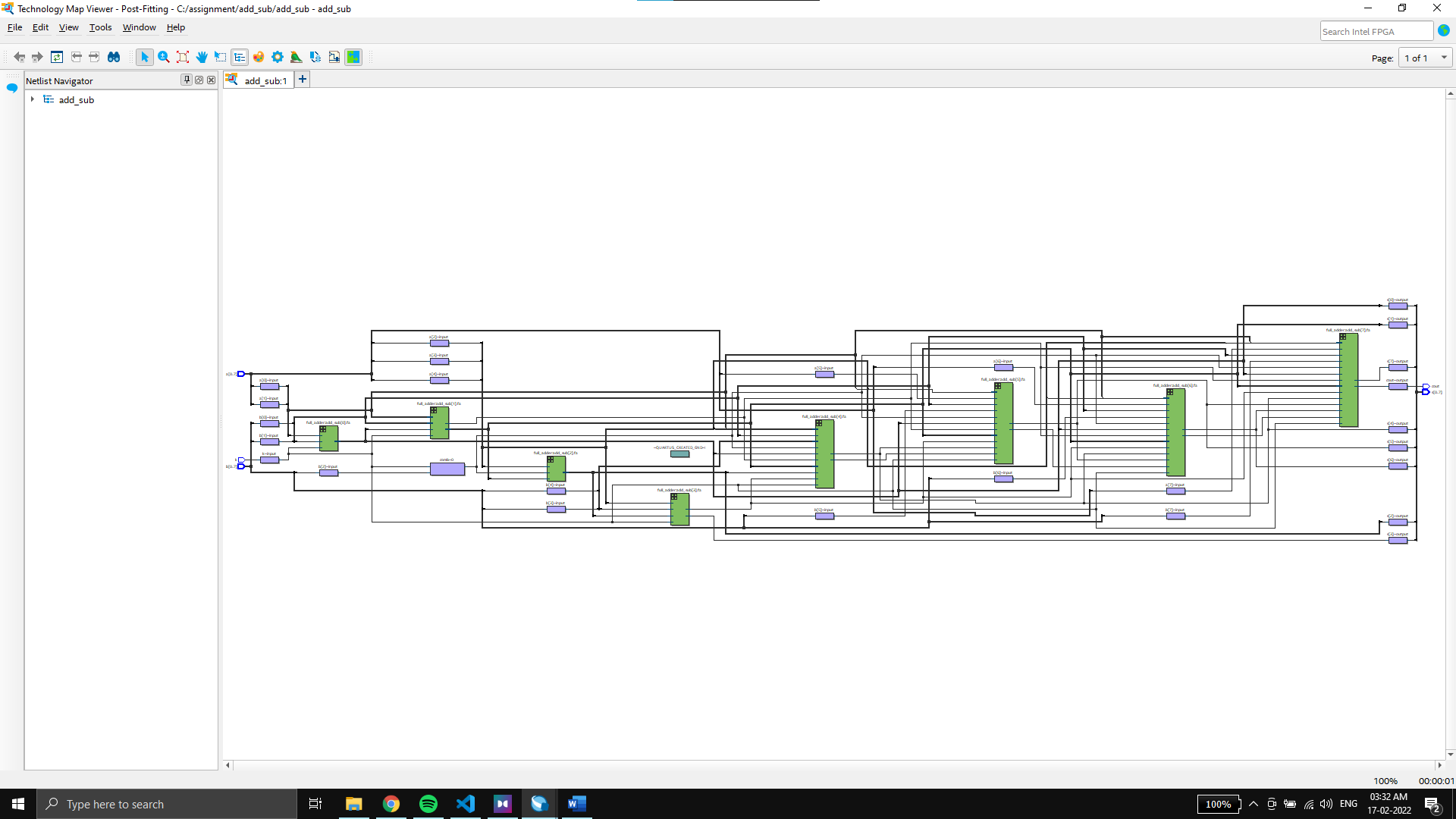
Flow summary:



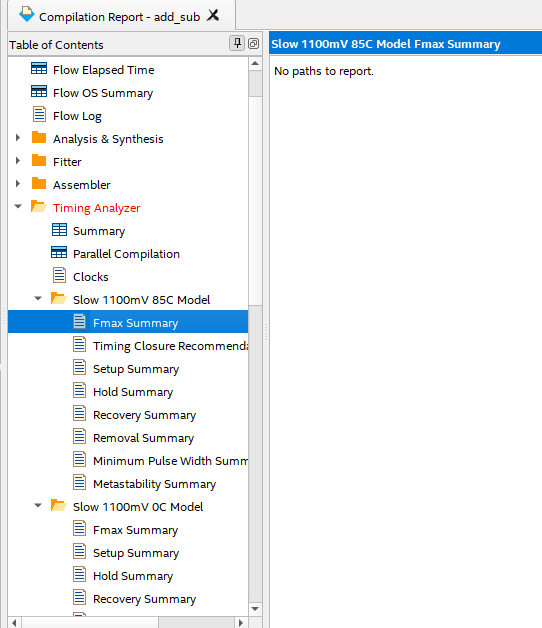
Resource usage summary:

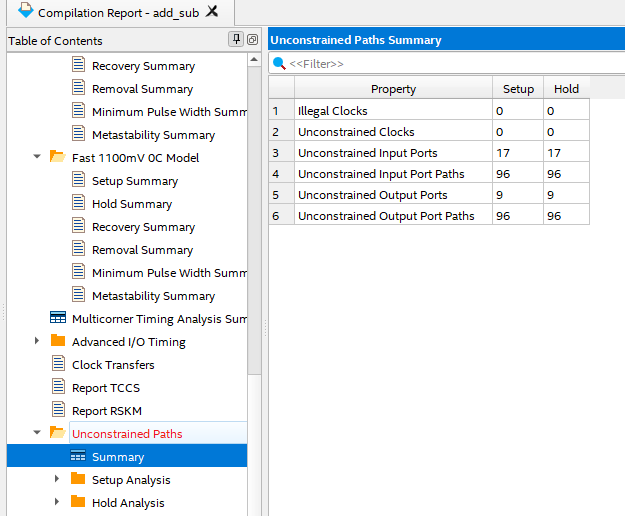


Block Diagram (Technology map viewer):

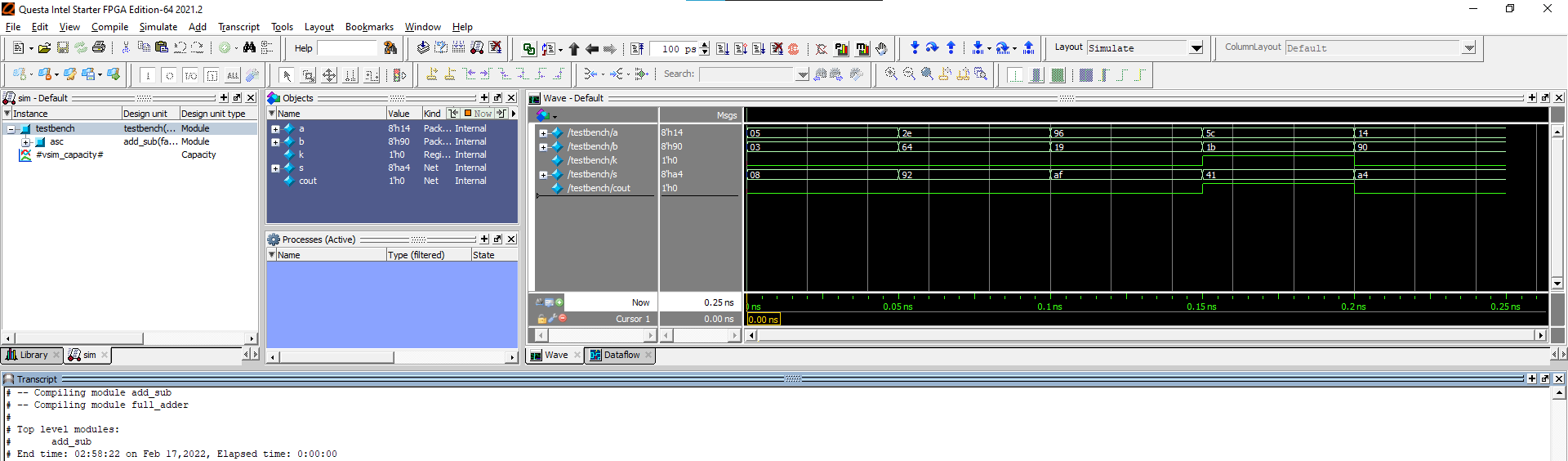


Timing reports:

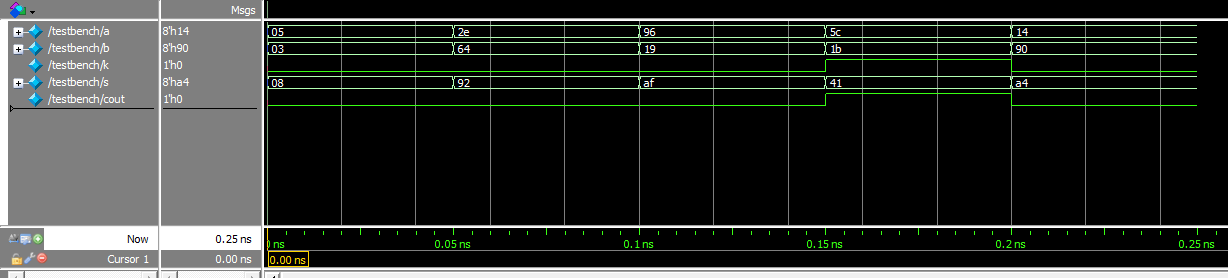




Simulation in Questa through Quartus:



Wave:



**2.** **Decade Counter** - Model a synchronous up/down decade counter with asynchronous reset. The counter is rising edge triggered.

If the **load** = 1, the data **data\_in** is loaded into the counter.

If the **counter\_on**=**counter\_up**=1, the counter is incremented, the Terminal carry output TC=1 when the counter is in state 9

If the **counter\_on**=1 and **counter\_up**=0, the counter is decremented, the Terminal carry output TC=1 when the counter is in state 0.

->

The decade counter counts up and down from 1to 9 and 9 to 1 in binary depending upon the input.

We will consider 10 states from s0 to s9, a clock, asynchronous reset, option to load values.

We have set different output values for different respective states.

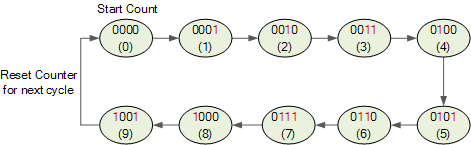
Depending upon the input value of counter\_on. We will set the next\_state for our FSM.

For example: if the system is in state s0 and count\_up is 1, then the next state will be s1 while if the count\_up was 0 then the next state will be s9 as it has to count downwards and switch to maximum after 0.

Our FSM will be positive edge activated hence we use it in our sensitive list for changing states.

For ex: s0 state will give output 0 for count, while s1 will give s1.

We will set TC to 1 for the conditions mentioned above in the question.



Verilog Code for the Decade counter:

module decade\_counter (count\_up, reset, load, counter\_on, clk, data\_in, count, TC);

input count\_up, reset, load, counter\_on, clk;

input [3:0] data\_in;

output reg [3:0]count;

output reg TC;

localparam [3:0] s0=4'b0000, s1=4'b0001, s2=4'b0010, s3=4'b0011, s4=4'b0100,s5=4'b0101, s6=4'b0110 , s7=4'b0111, s8=4'b1000, s9=4'b1001;

reg [3:0] state, next\_state;

always @ (posedge clk or negedge reset) begin

    if(!reset) begin state<= s0; end

    else if(load) begin

        state <= data\_in;

    end else if(counter\_on) state<= next\_state;

end

always @(state) begin

    case(state)

    s0: begin if(count\_up) begin next\_state <= s1; TC<=0; end

                else begin next\_state <= s9;  TC<=1; end

                count<= 0;

        end

    s1: begin if(count\_up)begin next\_state <= s2; end else begin next\_state <= s0; end count <= 1; TC<=1; end

    s2: begin if(count\_up)begin next\_state <= s3; end else begin next\_state <= s1; end count <= 2; TC<=0; end

    s3: begin if(count\_up)begin next\_state <= s4; end else begin next\_state <= s2; end count <= 3; TC<=0; end

    s4: begin if(count\_up)begin next\_state <= s5; end else begin next\_state <= s3; end count <= 4; TC<=0; end

    s5: begin if(count\_up)begin next\_state <= s6; end else begin next\_state <= s4; end count <= 5; TC<=0; end

    s6: begin if(count\_up)begin next\_state <= s7; end else begin next\_state <= s5; end count <= 6; TC<=0; end

    s7: begin if(count\_up)begin next\_state <= s8; end else begin next\_state <= s6; end count <= 7; TC<=0; end

    s8: begin if(count\_up)begin next\_state <= s9; end else begin next\_state <= s7; end count <= 8; TC<=0; end

    s9: begin if(count\_up)begin next\_state <= s0; TC<=1; end else begin next\_state <= s8; TC<=0; end count <= 9;  end

    default: begin next\_state <=s0; count <=0; TC=0; end

    endcase

end

endmodule

Verilog code for the test bench:

module testbench\_1();

integer i;

reg clk, counter\_on, reset, load, count\_up;

reg [3:0] data\_in;

wire [3:0] count;

wire TC;

decade\_counter dc1(count\_up, reset, load, counter\_on, clk, data\_in, count, TC);

initial

begin

counter\_on=1; count\_up =1;

reset=1; clk=1; #1; reset=0; #1 reset=1;

load=0; data\_in =4'b1000;

for( i=0; i<10; i=i+1)begin

clk =1; #20; clk =0; #20;

end

load =1;

clk =1; #20; clk =0; #20; load =0;

count\_up =0;

for( i=0; i<5; i=i+1)begin

clk =1; #20; clk =0; #20;

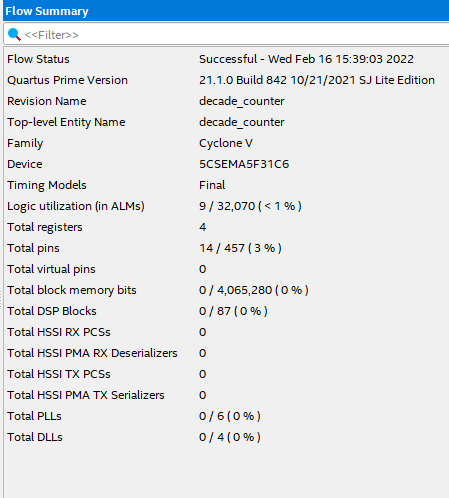
end

end

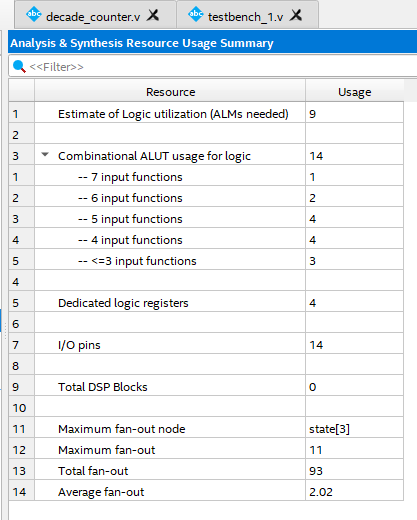
endmodule

Device Utilization Statistics:

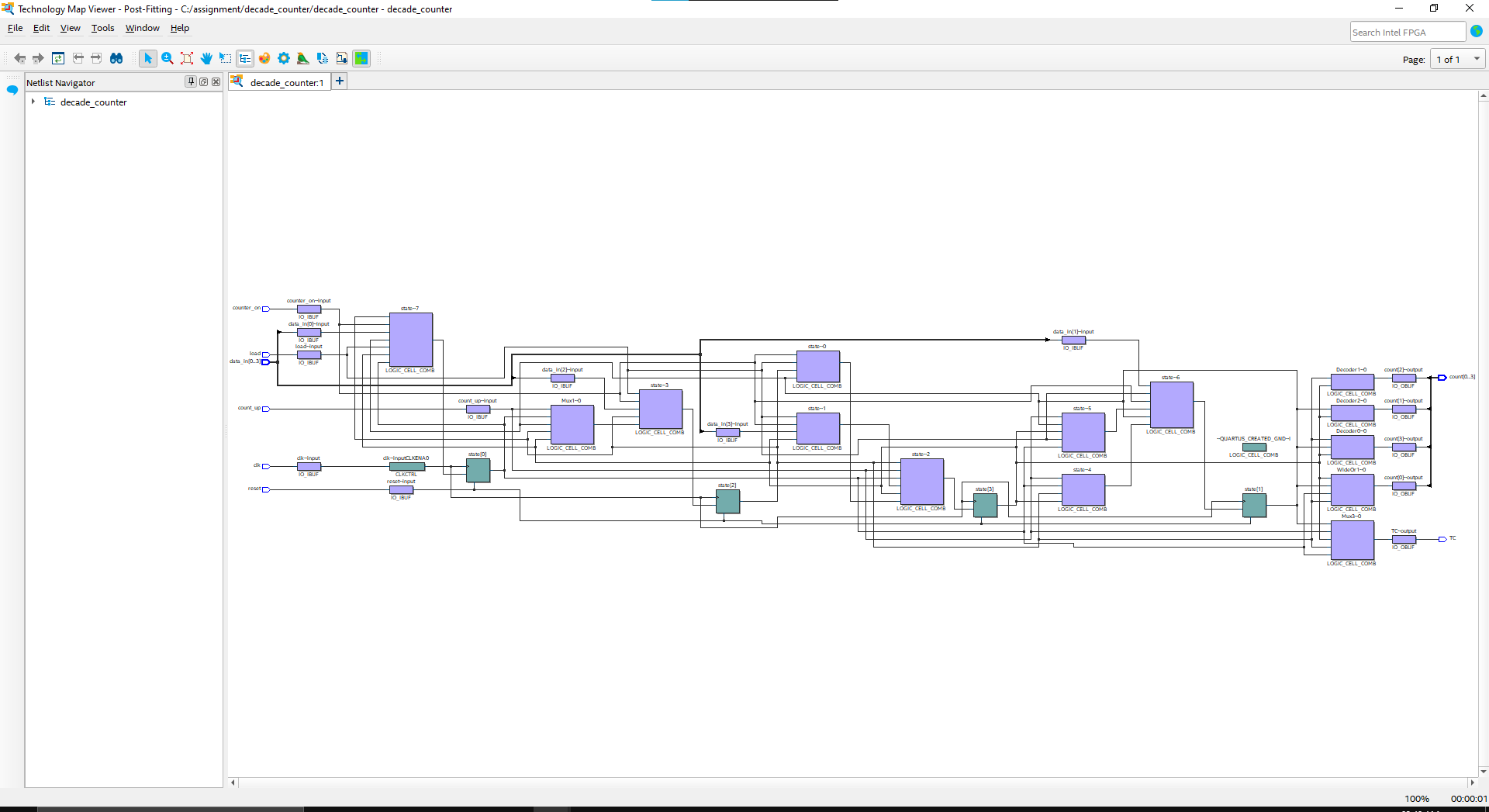
Flow summary:



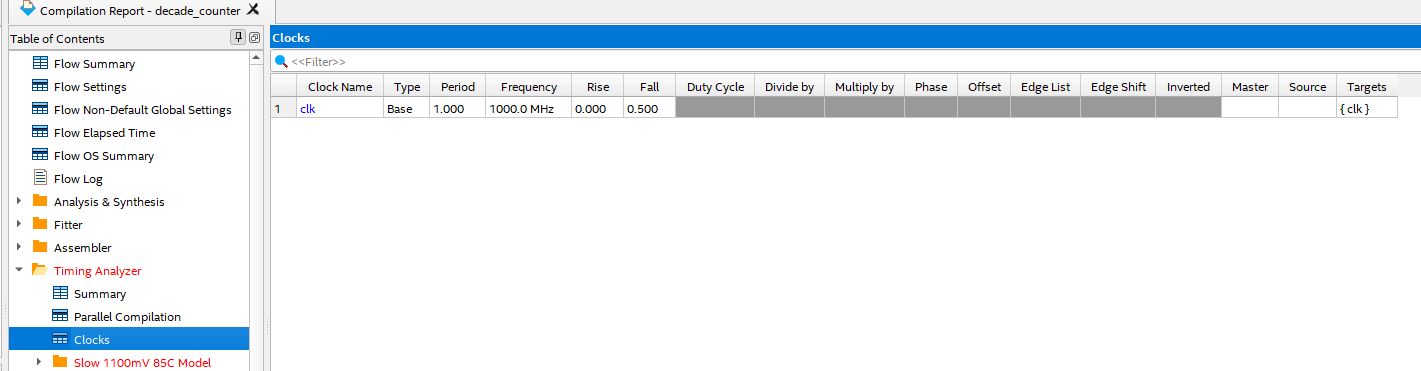
Resource usage summary:

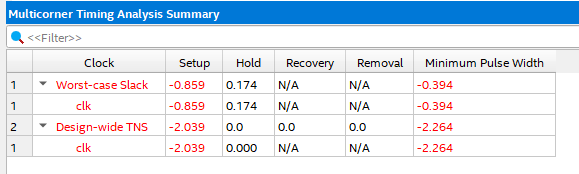


Block Diagram (Technology map viewer):

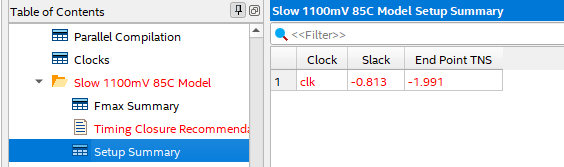
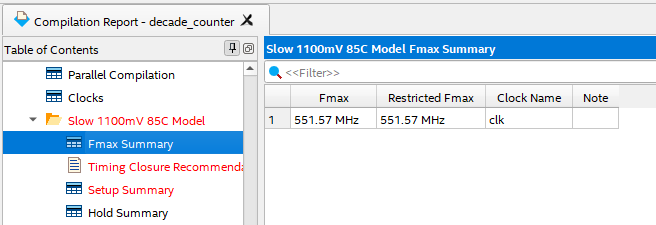
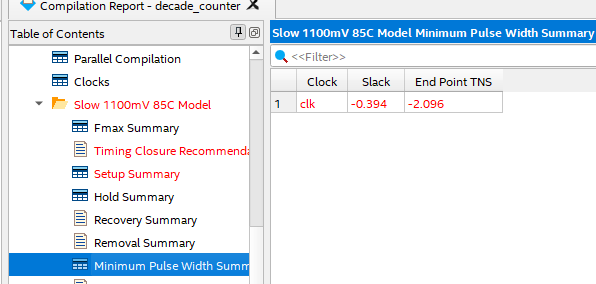
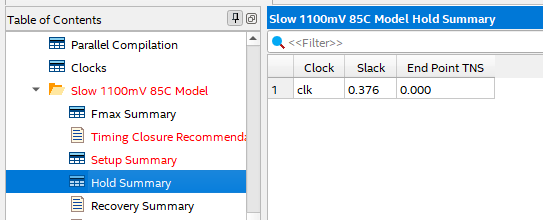


Timing reports:

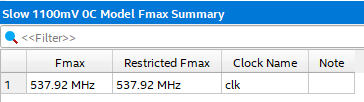
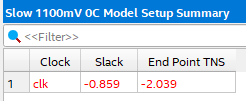
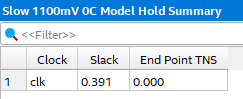
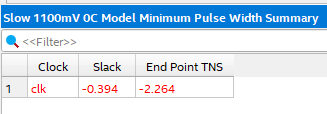




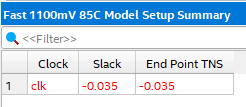
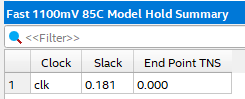
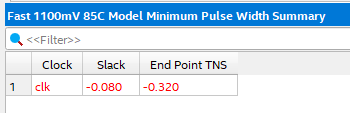
For slow 85C model:

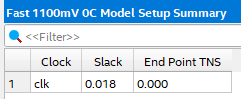
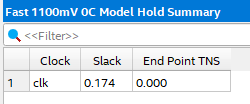
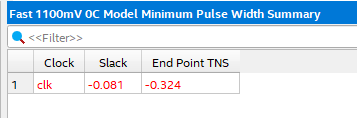
For slow 0C model:

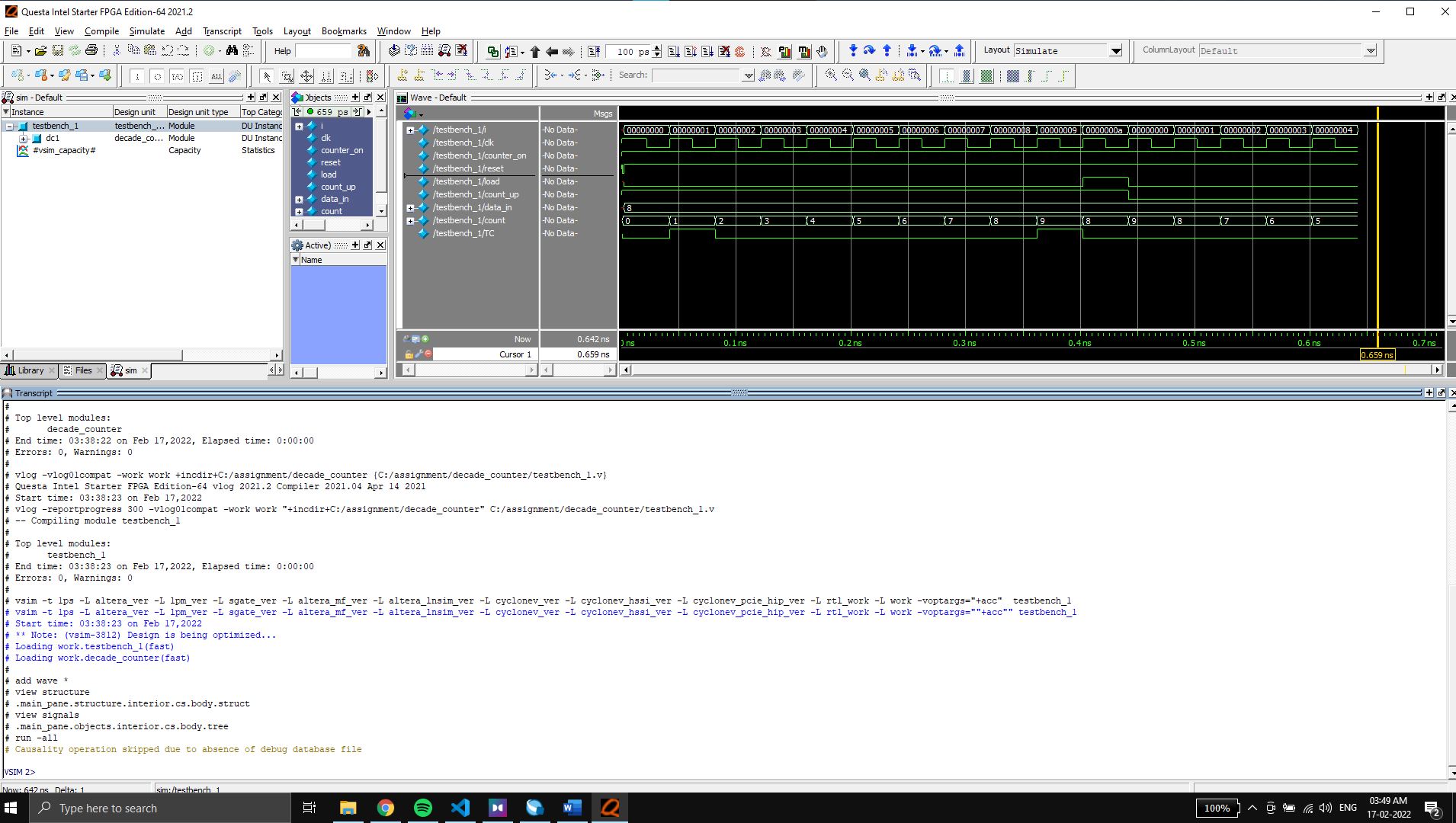
For fast 85C model:

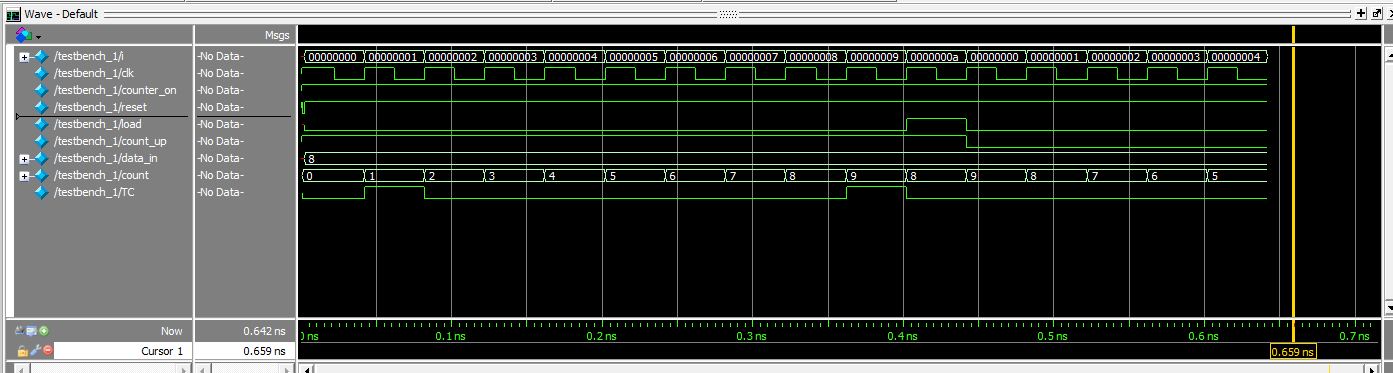
For fast 0C model:

Simulation in Questa through Quartus:



Wave:



3 . Serial adder to add two 8 bit numbers using a single full adder and registers

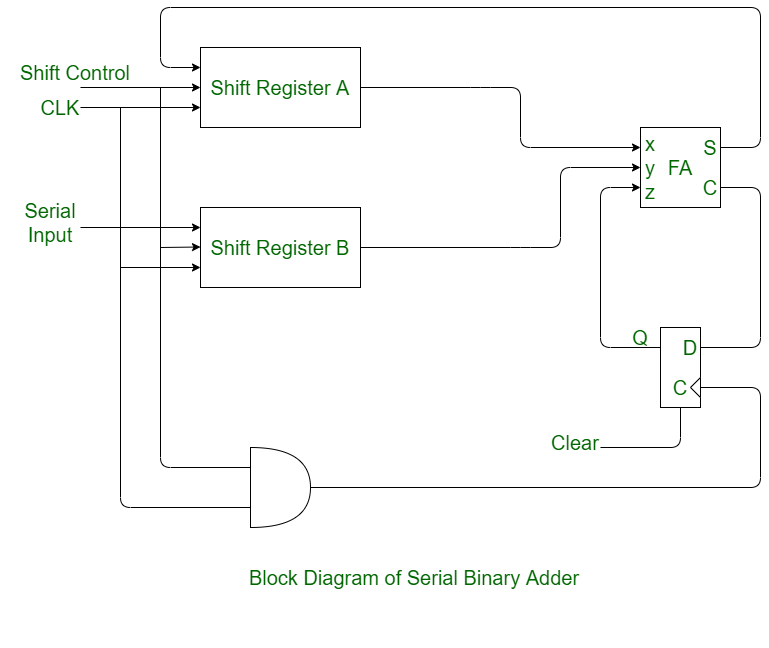
->Serial adder performs the addition of 2 binary numbers in serial form.

For designing the module, we will use separate submodules:

* 8 Right Shift registers: for inputs a, b and output s (sum)
* D flip flop: for storing carry
* Full adder: for carrying out addition of LSB of both inputs a and b and storing it in the msb of output sum.

We will use output register **done** for indicating the completion of addition. Keeping our FSM positive edge activated with a asynchronous reset.

We will take inputs at the MSB of inputs A and B and then shift them throughout the shift register, while at the same time we will right shift the sum. after 15 clock cycles we will have completely done taking in the input and adding them and storing it in output shift register sum. At which we will enable done. (image from gfg)



Verilog Code for the Serial Adder:

*// sum output is at output reg sum and the carry of the sum is at output reg C.*

module serial\_adder(

input ain, bin, clk, start, resetn,

output [7:0] sum, output reg done, output  c);

*//wires for connecting different modules*

wire [7:0] rxQ, ryQ, rsQ;wire fas, fac, dffc;

reg [3:0] count =4'b1111;

reg [7:0] D =0;

reg load = 0;

*//connecting wires from modules to output registers.*

assign sum = rsQ;

assign c = fac;

shiftreg regX(.D(D), .load(load), .shiftR(!done),.lin(ain), .clk(clk), .Q(rxQ));

shiftreg regY(.D(D), .load(load), .shiftR(!done),.lin(bin), .clk(clk), .Q(ryQ));

shiftreg regSum(.D(D), .load(load), .shiftR(!done),.lin(fas), .clk(clk), .Q(rsQ));

fulladder fa(.a(rxQ[0]), .b(ryQ[0]), .cin(dffc), .sum(fas), .cout(fac));

dflipflop da(.d(fac), .reset(load), .clk(clk), .q(dffc));

*//for decrementing counter and making a reset button for entire module.*

always @(posedge clk, negedge resetn)

begin

if(!resetn) begin

load<= 1;

count<=15;

done<=0;

end else if(start) begin

load<= 0;

if(count >0) begin done <= 0;

count<= count -1;

    end

else done <=1;

end

end

endmodule

*//single bit full adder module required.*

module fulladder(input a, b, cin, output sum, cout);

assign sum = cin^(a^b);

assign cout = (a&b) | cin&(a^b);

endmodule

*// D flip flop*

module dflipflop(input d, reset, clk, output reg q);

always@(posedge clk, posedge reset)

begin

if(reset) q=0;

else begin

q=d;

end

end

endmodule

*//8 bit shift register with load support.*

module shiftreg (input [7:0]D, input load, shiftR, lin, clk, output reg [7:0]Q);

always @(posedge clk)

begin

if(load) Q<=D;

else if(shiftR) Q <= {lin, Q[7:1]};

end

endmodule

Verilog code for the test bench:

module tb2();

reg a, b, clk, start, resetn;

wire [7:0]sum;

wire done, c;

reg [7:0] testinputx, testinputy;

integer i;

serial\_adder ut(.ain(a), .bin(b), .clk(clk), .start(start), .resetn(resetn),.sum(sum),

.done(done), .c(c));

initial

begin

*//giving starting default values*

a=0; b=0; start=1; clk=0;

*//setting teh module at reset.*

resetn=1; #1; resetn=0; #1; resetn=1;

*//input values for x and y, from left to right.*

testinputx = 8'b01010110;

testinputy = 8'b10110010;

*//cyclic inputs x and y transversing the above values.*

for( i=7; i>=0; i=i-1)begin

a = testinputx[i];

b = testinputy[i];

clk =1; #20;

clk =0; #20;

end

*//additional cycles for computing the adder result.*

for( i=7; i>=0; i=i-1)begin

clk =1; #20;

clk =0; #20;

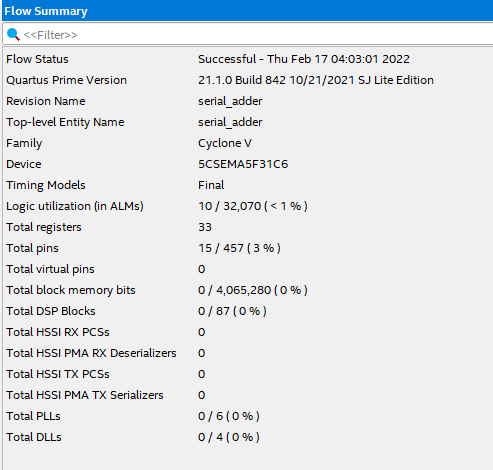
end

end

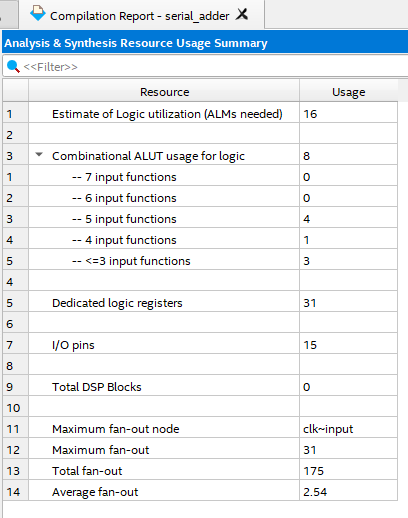
endmodule

Device Utilization Statistics:

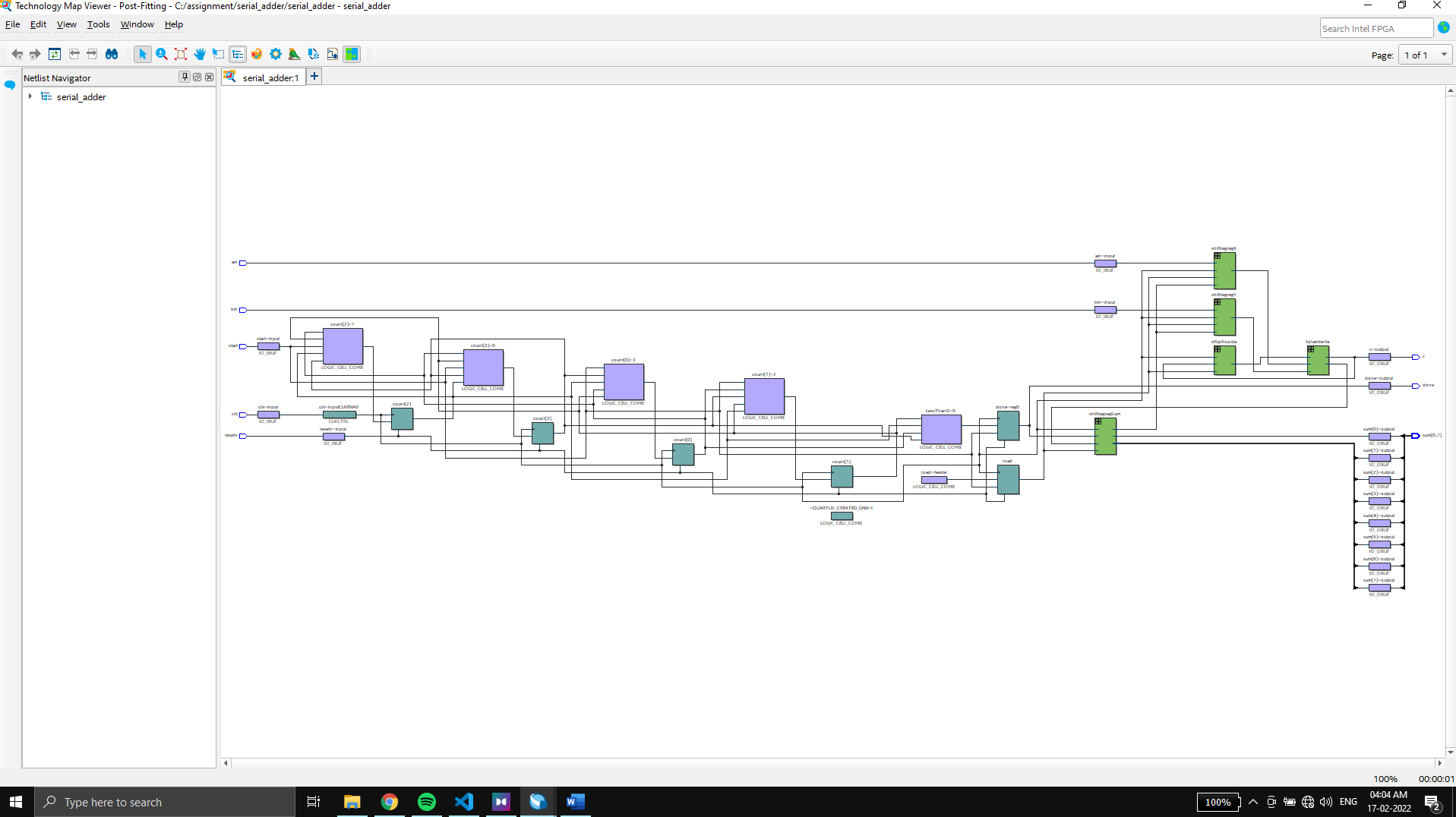
Flow summary:



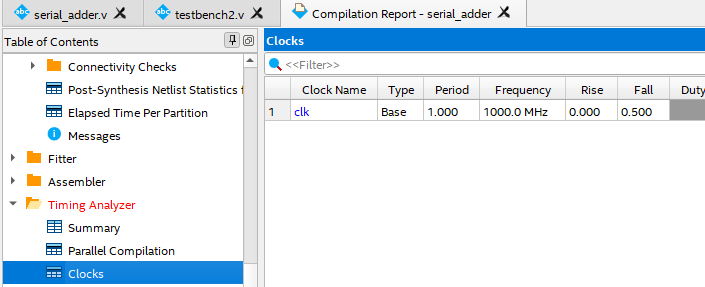
Resource usage summary:



Block Diagram (Technology map viewer):

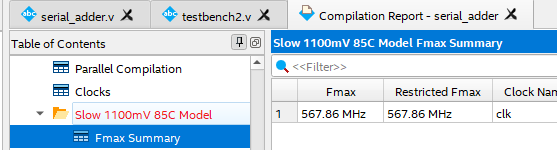


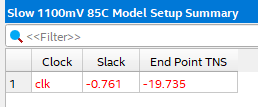
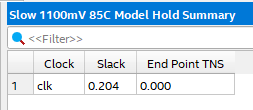
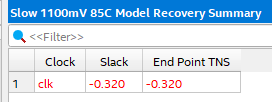
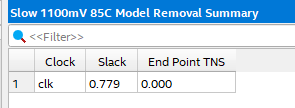
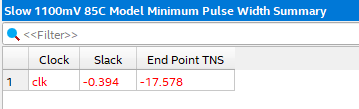
Timing reports:



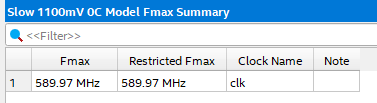
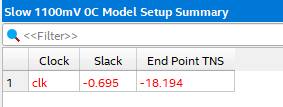
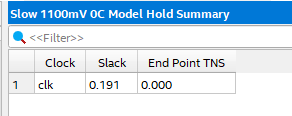
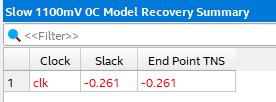
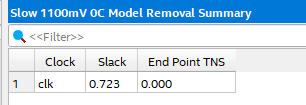
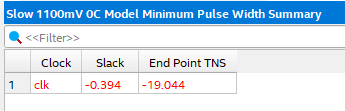


For slow 85C model:

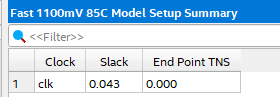
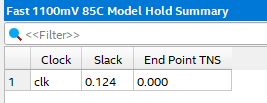
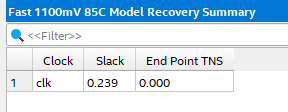
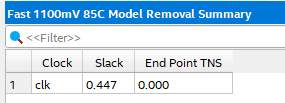
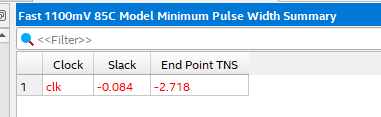


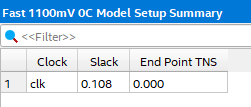
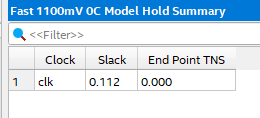
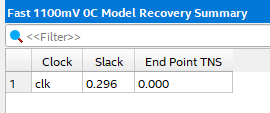
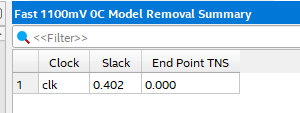
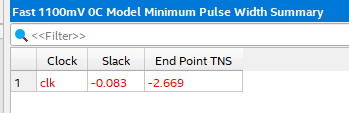
For slow 0C model:

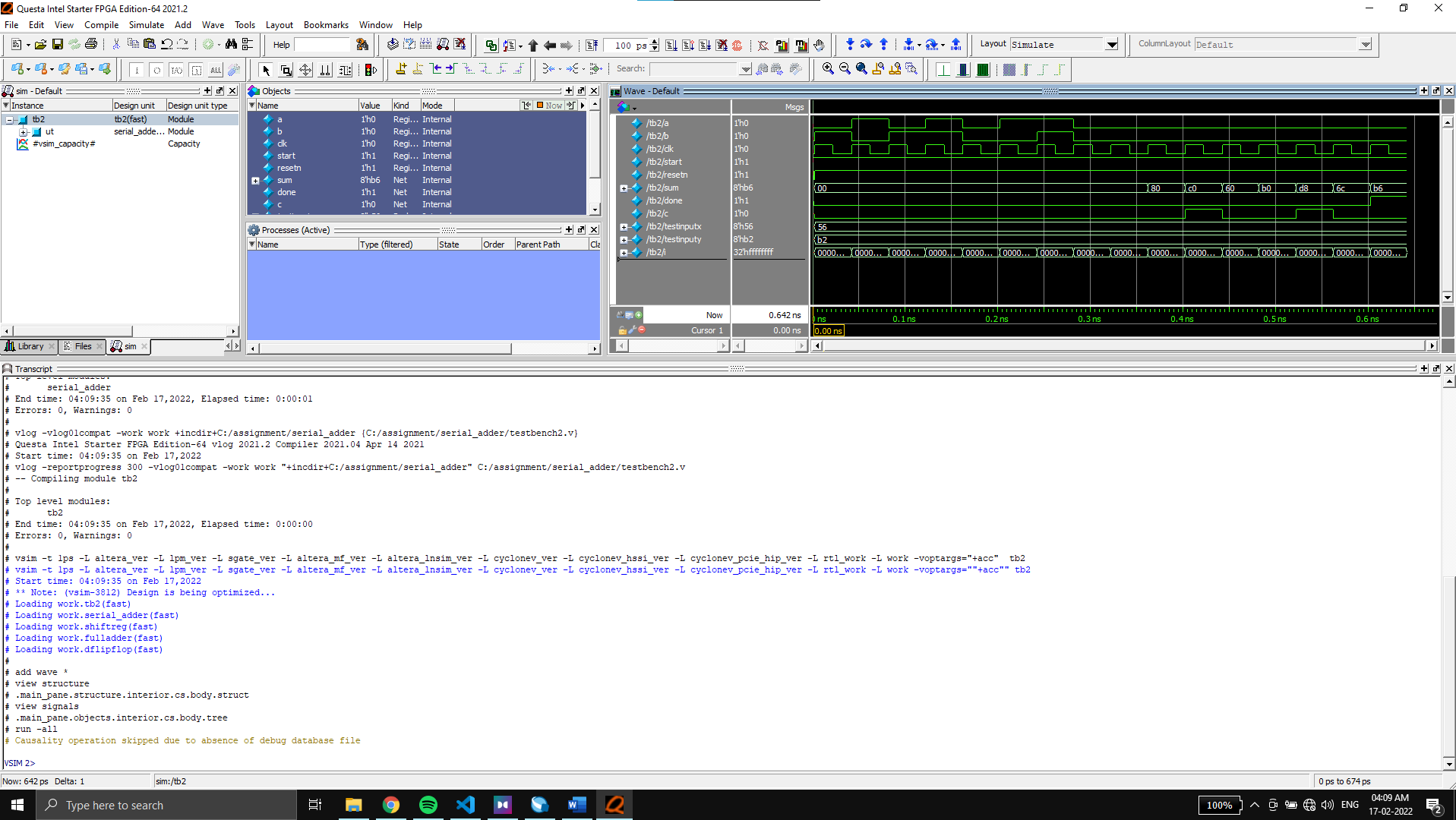
For fast 85C model:

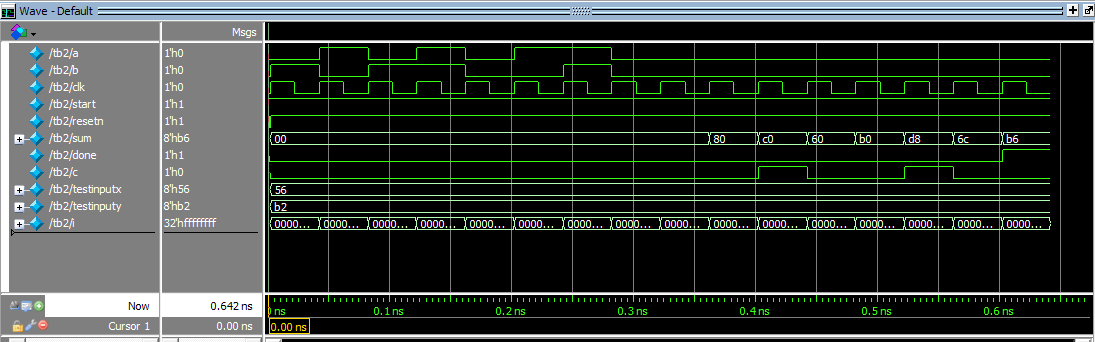
For slow 0C model:

Simulation in Questa through Quartus:



Wave:



**4. Shift and add multiplier in Verilog to multiply two 8 bit numbers using RTL approach. It should be capable of multiplying both positive and negative numbers. Negative numbers are represented in 2’sC representation.**

Storing the input values in 8 bit registers A and B we will display the output using a 16bit Register P.

we know that for signed multiplication when multiplicand is negative we need to ignore carry and use 1 as the msb’s while adding to the register P when the multiplier bit is 1 while when multiplier is negative we just need to add two’s complement in the last step.

We will use 10 states from s0 to s9 indicating value 0 to 9 respectively.

We will make our system positively edge activated with asynchronous reset.

On each state we will be storing the left shifted value of A\_q in A\_d storing the right shifted value of B\_q in B\_d.

On each shift to positive edge of the clk(clock), we will update the output pdt\_q with the calculated next\_ouput from the 2nd always block according to the method discussed above. We will also updating the shifted values in the A\_q and B\_q (from A\_d and B\_d respectively) at that time.

Verilog Code for the Shift and add multiplier:

module shift\_multiply(input clk, resetn, start, input [7:0]A, B, output [15:0] P, output reg done);

localparam S0=0, S1=1, S2=2, S3=3, S4=4, S5=5, S6=6, S7=7, S8=8, S9=9;

reg [4:0] state\_d, state\_q;

reg [7:0] B\_d, B\_q;

reg [15:0] A\_d, A\_q, pdt\_d, pdt\_q;

assign P = pdt\_q;

always @ (posedge clk , negedge resetn)

begin

    if(!resetn) state\_q <=S0;

    else begin

    state\_q <= state\_d;

    pdt\_q <=pdt\_d;

    A\_q <= A\_d;

    B\_q <= B\_d;

    end

end

always @(state\_q) begin

    state\_d = state\_q;

    done = 1'b0;

    case(state\_q)

        S0:if(start) state\_d=S1;

        S1: state\_d=S2;

        S2: state\_d=S3;

        S3: state\_d=S4;

        S4: state\_d=S5;

        S5: state\_d=S6;

        S6: state\_d=S7;

        S7: state\_d=S8;

        S8: state\_d=S9;

        S9: begin

        done= 1'b1;

        if(start)state\_d =S1;

        end

        default: state\_d=S0;

    endcase

end

always @(state\_q or pdt\_q or A\_q or B\_q)

begin

pdt\_d = pdt\_q; A\_d = A\_q; B\_d = B\_q;

case (state\_q)

S0: begin

pdt\_d = {16{1'b0}};

A\_d = A;

B\_d = B;

end

S1: begin

A\_d = A\_q << 1;

B\_d = B\_q >> 1;

if (B\_q[0] == 1'b1) begin

    if(A[7] == 1'b1) pdt\_d = {8'b11111111,A\_q[7:0]} +pdt\_q;

    else pdt\_d = A\_q + pdt\_q;

    end

end

S2: begin

A\_d = A\_q << 1;

B\_d = B\_q >> 1;

if (B\_q[0] == 1'b1) begin

    if(A[7] == 1'b1) pdt\_d = {7'b1111111,A\_q[8:1], 1'b0} +pdt\_q;

    else pdt\_d = A\_q + pdt\_q;

    end

end

S3: begin

A\_d = A\_q << 1;

B\_d = B\_q >> 1;

if (B\_q[0] == 1'b1) begin

    if(A[7] == 1'b1) pdt\_d = {6'b111111,A\_q[9:2], 2'b00} +pdt\_q;

    else pdt\_d = A\_q + pdt\_q;

    end

end

S4: begin

A\_d = A\_q << 1;

B\_d = B\_q >> 1;

if (B\_q[0] == 1'b1) begin

    if(A[7] == 1'b1) pdt\_d = {5'b11111,A\_q[10:3], 3'b000} +pdt\_q;

    else pdt\_d = A\_q + pdt\_q;

    end

end

S5: begin

A\_d = A\_q << 1;

B\_d = B\_q >> 1;

if (B\_q[0] == 1'b1) begin

    if(A[7] == 1'b1) pdt\_d = {4'b1111,A\_q[11:4], 4'b0000} +pdt\_q;

    else pdt\_d = A\_q + pdt\_q;

    end

end

S6: begin

A\_d = A\_q << 1;

B\_d = B\_q >> 1;

if (B\_q[0] == 1'b1) begin

    if(A[7] == 1'b1) pdt\_d = {3'b111,A\_q[12:5], 5'b00000} +pdt\_q;

    else pdt\_d = A\_q + pdt\_q;

    end

end

S7: begin

A\_d = A\_q << 1;

B\_d = B\_q >> 1;

if (B\_q[0] == 1'b1) begin

    if(A[7] == 1'b1) pdt\_d = {2'b11,A\_q[13:6], 6'b000000} +pdt\_q;

    else pdt\_d = A\_q + pdt\_q;

    end

end

S8: begin

    A\_d = A\_q << 1;

    B\_d = B\_q >> 1;

    if (B\_q[0] == 1'b1) begin

        if(B[3] == 1'b1) pdt\_d = {1'b0, (~A+1'b1),7'b0000000} +pdt\_q;

        else if(A[3] == 1'b1) pdt\_d = {1'b1,A\_q[15:8], 7'b0000000} +pdt\_q;

        else pdt\_d = A\_q + pdt\_q;

    end

end

default: pdt\_d = pdt\_q;

endcase

end

endmodule

Verilog code for the test bench:

module test\_bench();

reg clk, resetn, start;

reg [7:0] A, B;

wire [15:0] P;

wire done;

integer i;

shift\_multiply ut(.clk(clk), .resetn(resetn), .start(start), .A(A), .B(B), .P(P), .done(done));

initial

begin

    clk=0;resetn=0;

    #2; clk=1;

    A=8'b00101011; B= 8'b01001101;

    #2 resetn=1; clk=0; start=1'b1;

    for(i=0; i<=9;i=i+1) begin

    #20 clk =1; #20 clk=0;

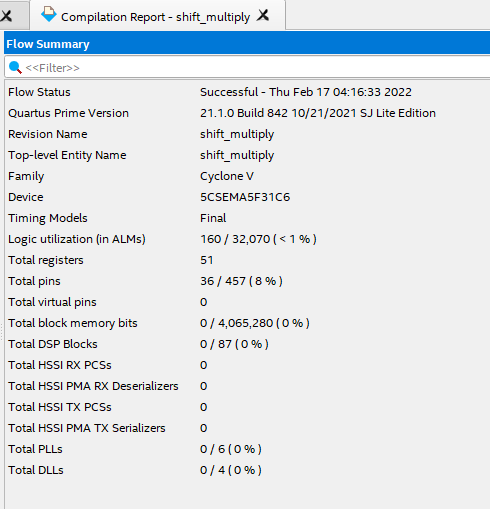
    end

end

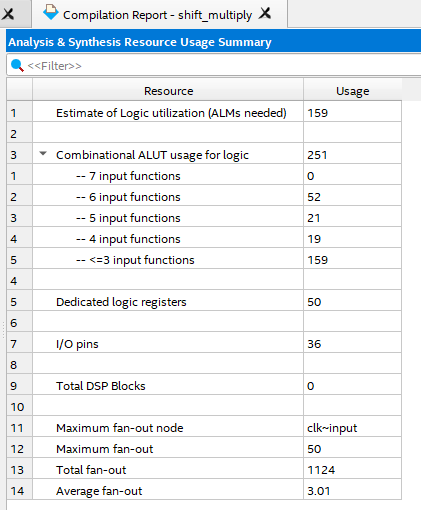
endmodule

Device Utilization Statistics:

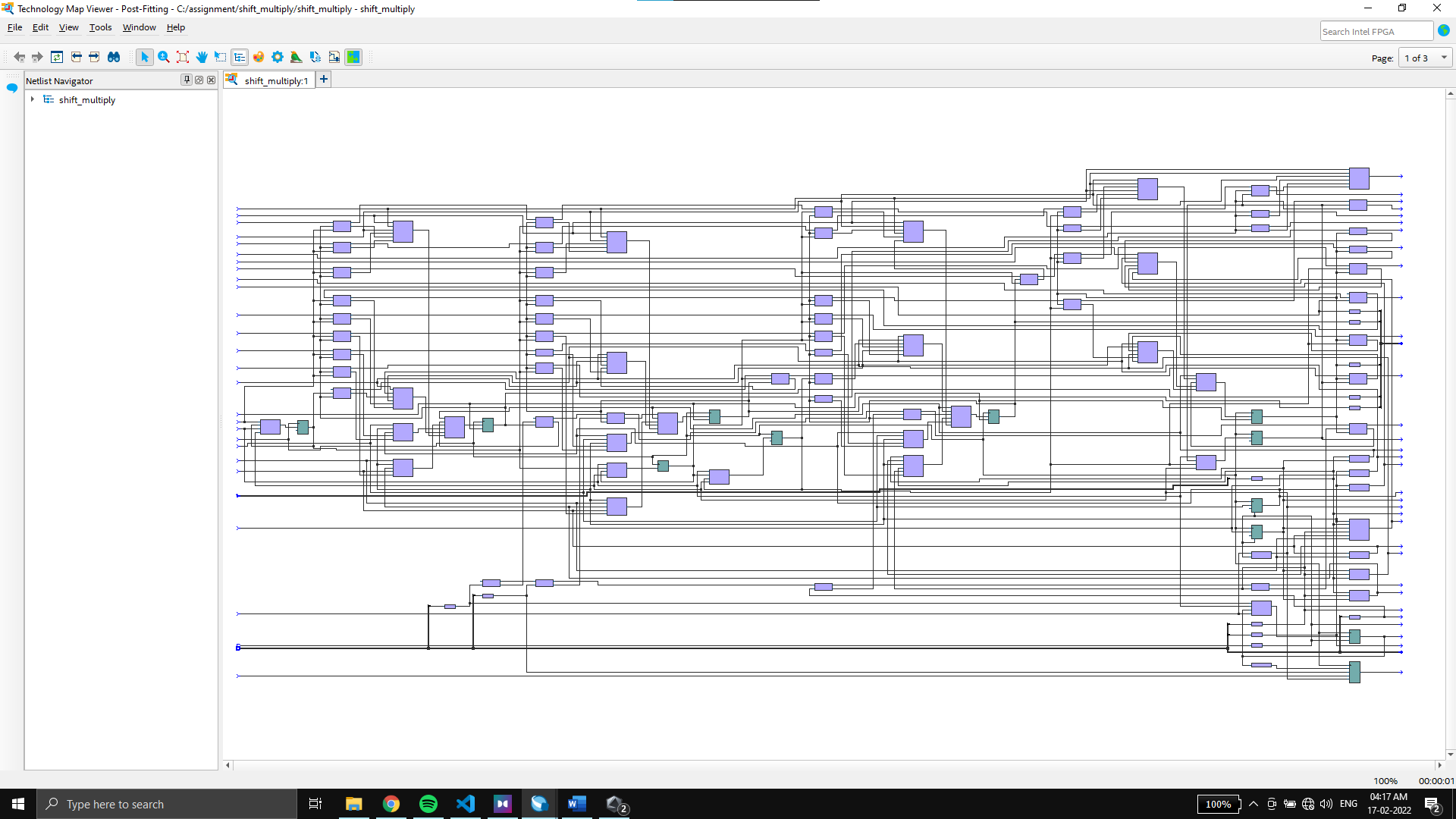
Flow summary:



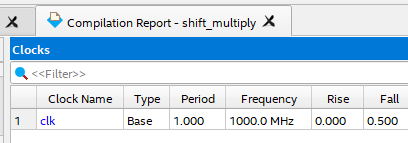
Resource usage summary:



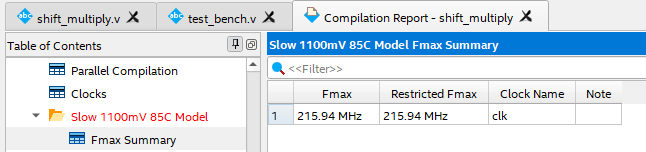
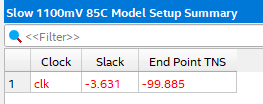
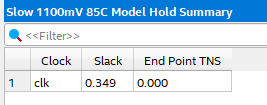
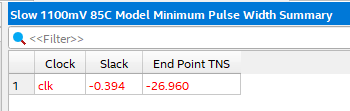
Block Diagram (Technology map viewer):



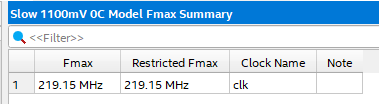
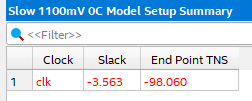
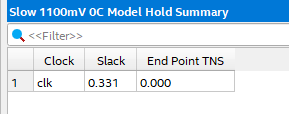
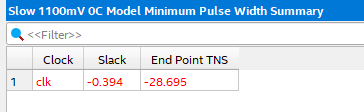
Timing reports:



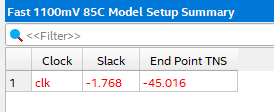
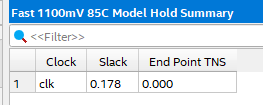
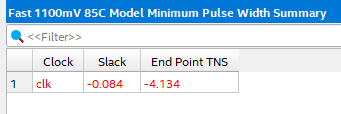
For slow 85C model:

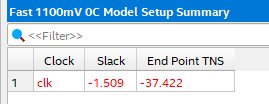
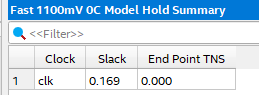
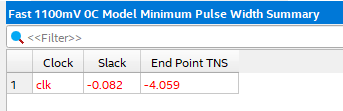
For slow 0C model:

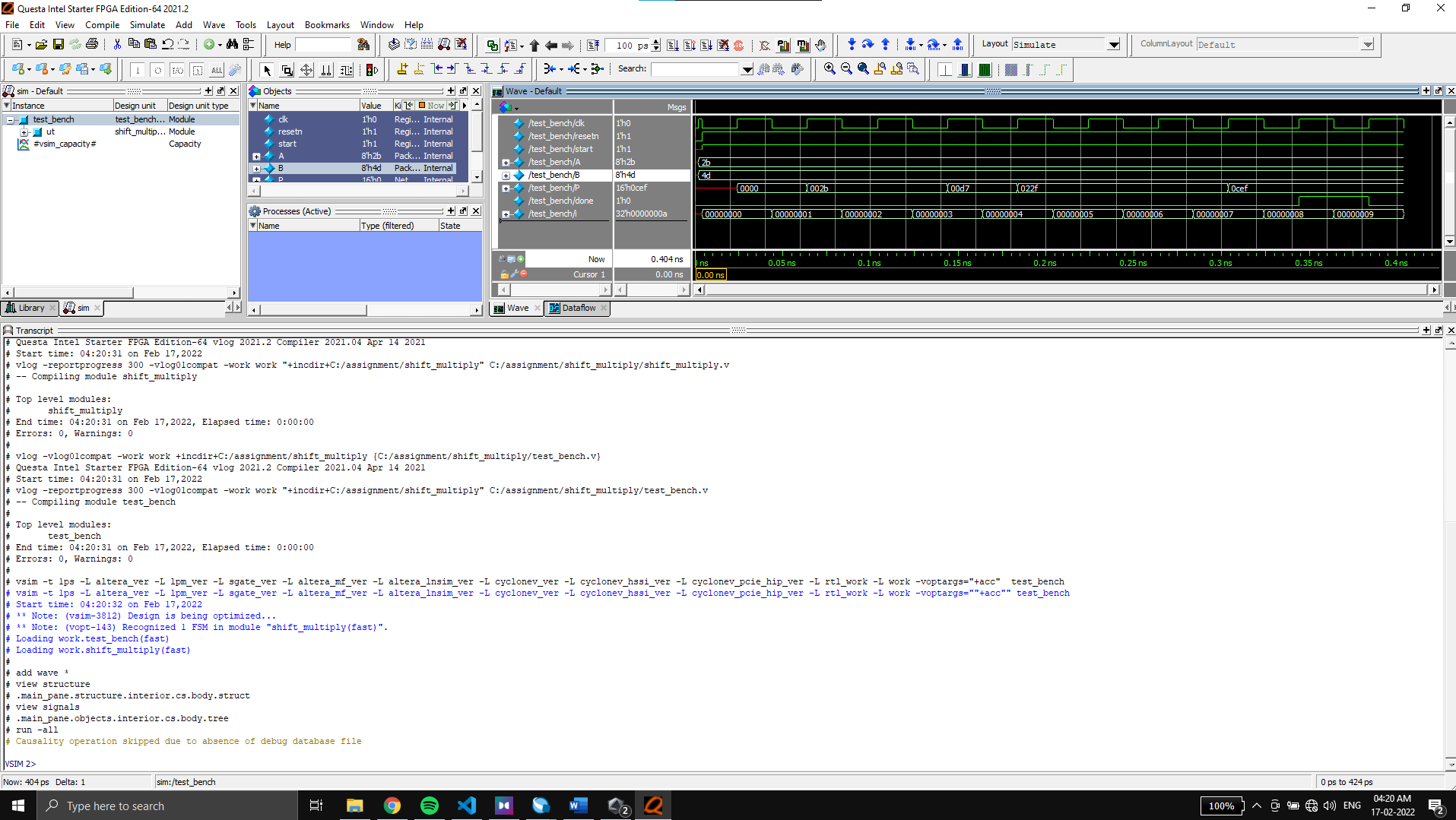
For fast 85C model:

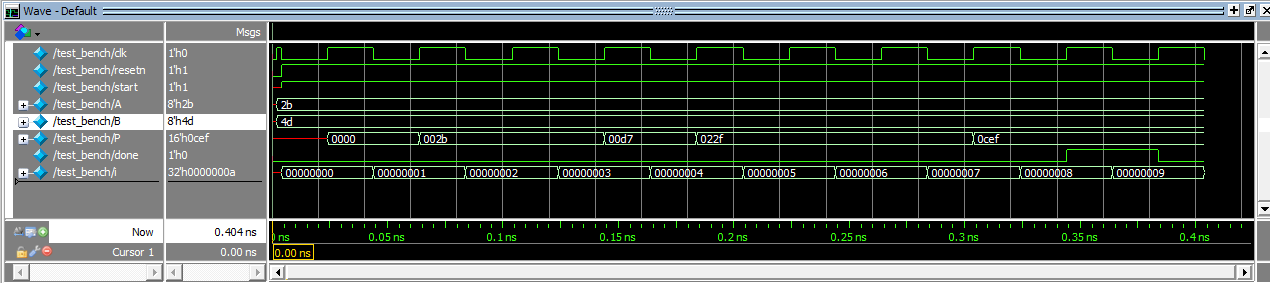
For fast 0C model:

Simulation in Questa through Quartus:



Wave:



**5. To compute the factorial of a 3-bit number using RTL approach.**

->

To design a system which calculates factorial of up to 3-bit numbers.

We will consider a 3bit input **a**, of which the factorial has to be computed and it will be displayed on the 13 bits output **out**.

We will consider an FSM with **8 states** with positive edge activated clock and asynchronous reset **resetn**.

In states 0 to 6, we will be multiplying the output with corresponding state value, and storing it in **out\_next**.

On switch from negative edge to positive edge of the clock, **out\_next** value will get stored in the output register **out**.

On reset, the output will revert to 0 and state to 0.

On competition of necessary factorial calculation, a condition will be met **(count<=state+1)** which will switch the state to **7** at which it will be stuck forever showing the required factorial and a **done** output bit will turn to 1 to indicate the completion of it.

The count register stores the value of input a.

We will assume that the input isn’t 0.

Verilog Code for the Factorial calculator:

module factorial (input [2:0]a, input clock, resetn, output reg [12:0] out, output reg done);

reg [3:0] state, next\_state;

reg [2:0] count;

reg [12:0] out\_next;

always @(posedge clock, negedge resetn)

begin

    if(!resetn) begin out <=1;state<=0; end

    else begin

    state <= next\_state;

    out <= out\_next;

    end

end

always @ (state) begin

    count <=a;

    case(state)

    0: begin

            if(count<=state+1) begin next\_state <= 7; end

            else begin next\_state <= 1;

            done <=0;

            out\_next <= 1\*out;

            end

        end

    1: begin

            if(count<=state+1) begin next\_state <= 7; end

            else begin next\_state <= 2;

            end

            done <=0;

            out\_next <= 2\*out;

        end

    2: begin

            if(count<=state+1) begin next\_state <= 7; end

            else begin next\_state <= 3;

            end

            done <=0;

            out\_next <= 3\*out;

        end

    3: begin

            if(count<=state+1) begin next\_state <= 7; end

            else begin next\_state <= 4;

            end

            done <=0;

            out\_next <= 4\*out;

        end

    4: begin

            if(count<=state+1) begin next\_state <= 7; end

            else begin next\_state <= 5;

            end

            done <=0;

            out\_next <= 5\*out;

        end

    5: begin

            if(count<=state+1) begin next\_state <= 7; end

            else begin next\_state <= 6;

            end

            done <=0;

            out\_next <= 6\*out;

        end

    6: begin

            if(count<=state+1) begin next\_state <= 7; end

            else begin next\_state <= 7;

            end

            done <=0;

            out\_next <= 7\*out;

        end

    7: begin

            next\_state <= 7;

            done <=1;

        end

    endcase

end

endmodule

Verilog code for the test bench:

module testbench();

reg [2:0] a;

reg clock, resetn;

wire [12:0] out;

wire done;

integer i;

factorial faccalc(a, clock, resetn, out, done);

initial begin

a=5;

resetn=1; clock =1;

#5 resetn=0; #5 resetn=1;

for(i=0; i<10; i=i+1)begin

    clock=1;#50; clock=0; #50;

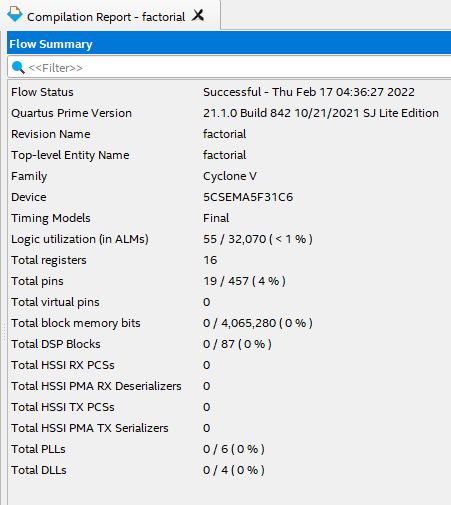
end

end

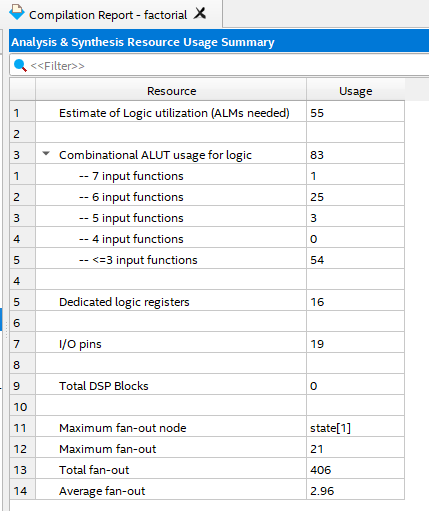
endmodule

Device Utilization Statistics:

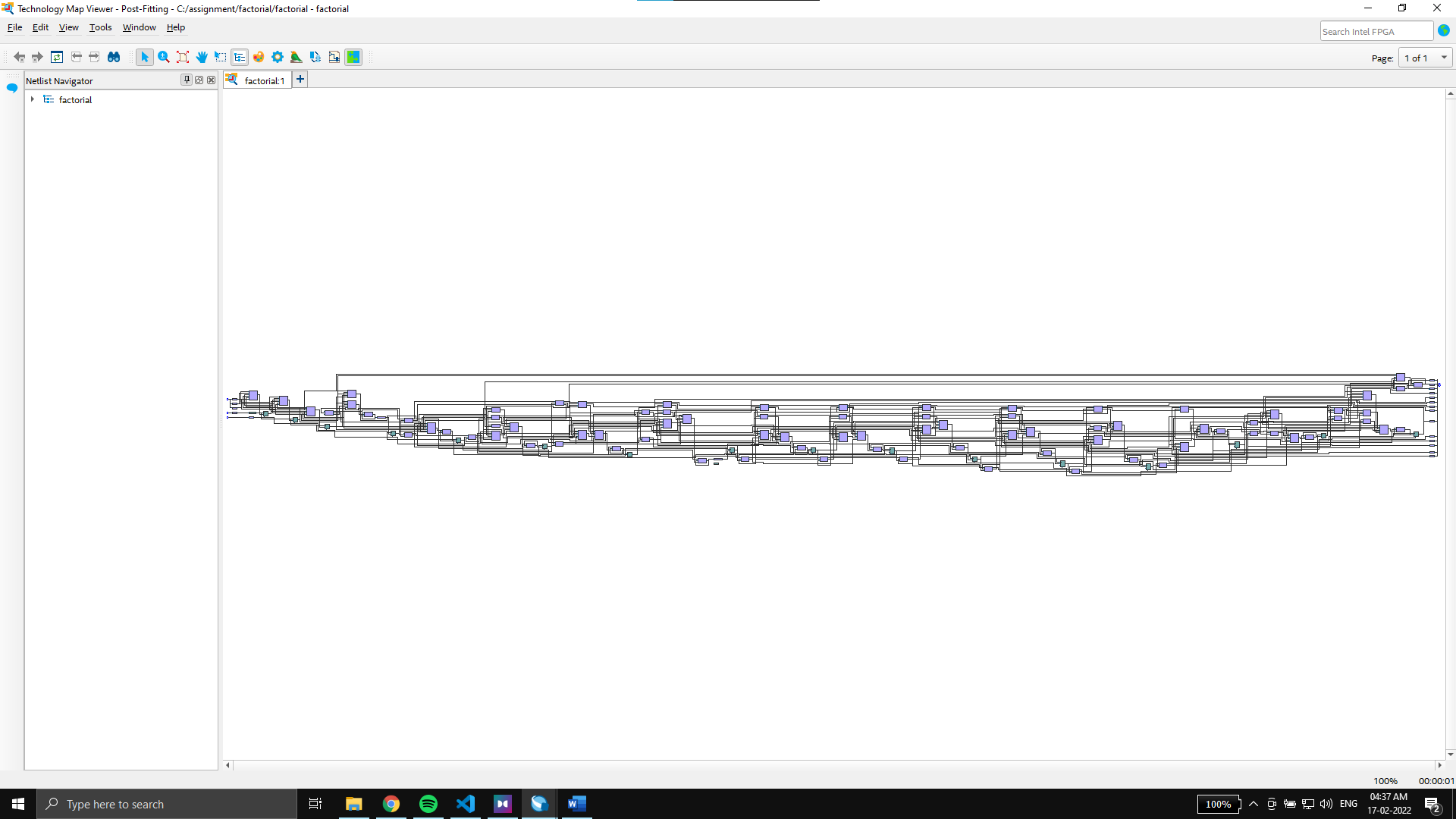
Flow summary:



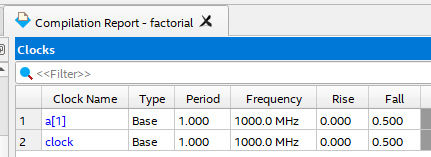
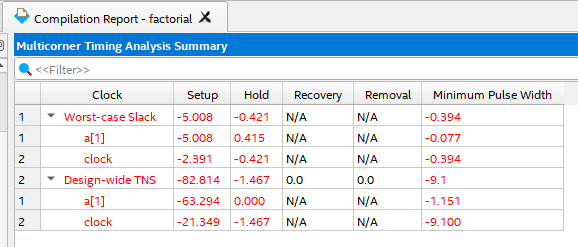
Resource usage summary:



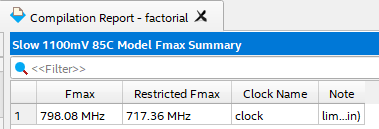
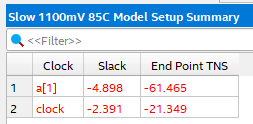
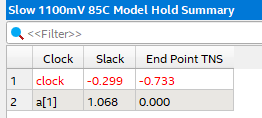
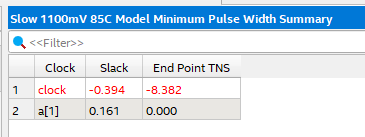
Block Diagram (Technology map viewer):



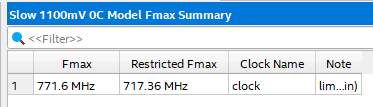
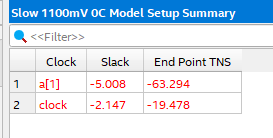
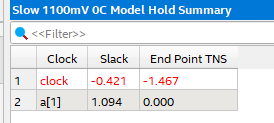
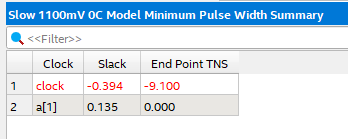
Timing reports:

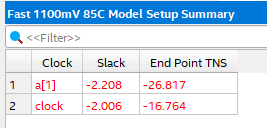
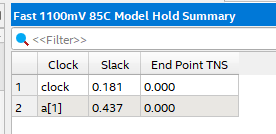
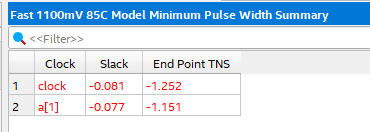
For slow 85C model:

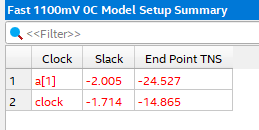
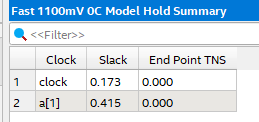
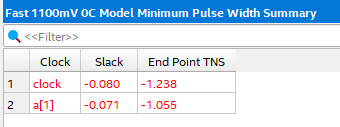
For slow 0C model:

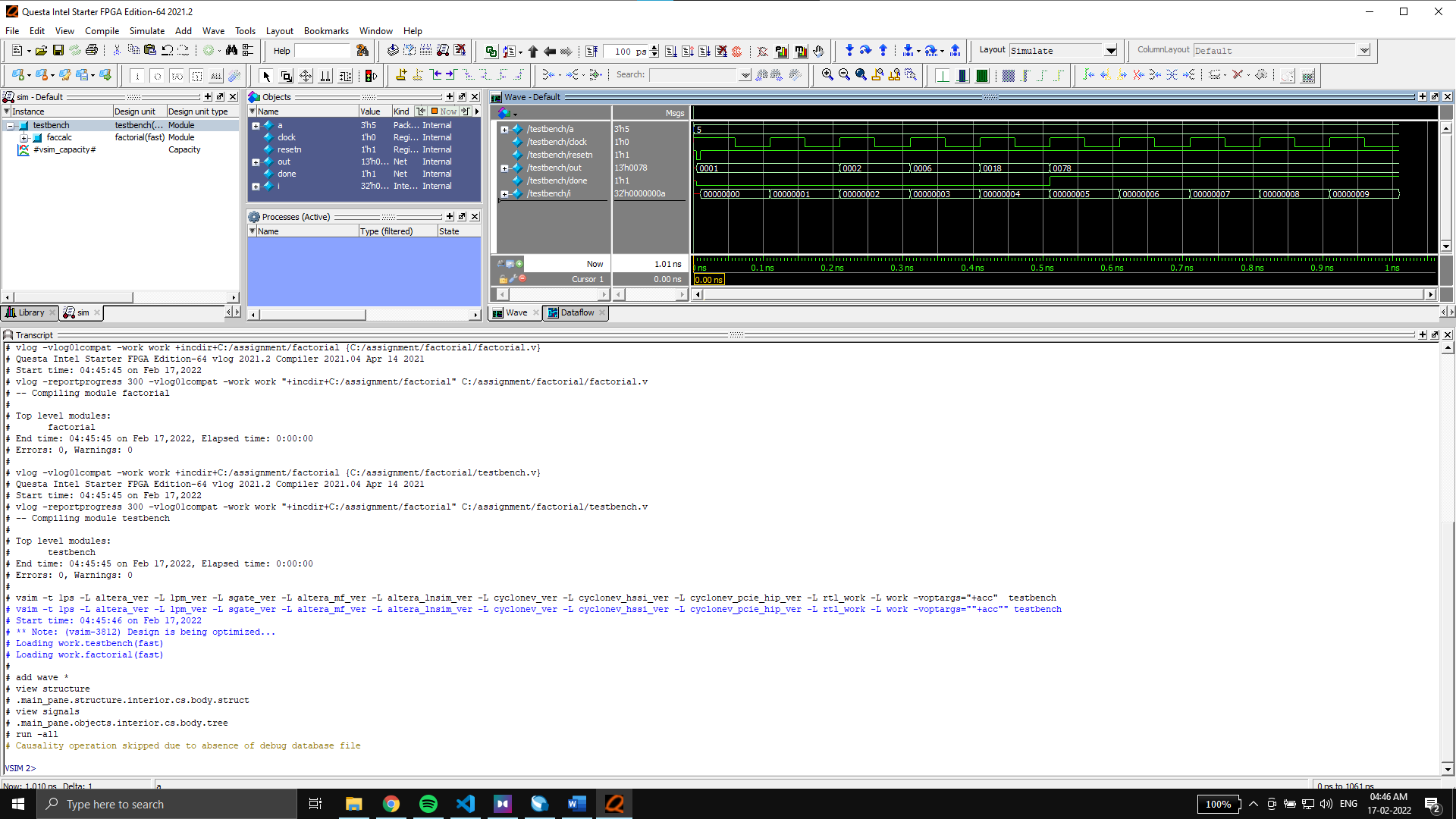
For fast 85C model:

For fast 0C model:

Simulation in Questa through Quartus:



Wave:

