Experiment - 3: Parallel and Array Multiplier Simulation Using Quartus Prime

NAME: Jayakrishnan Menon

REG NO: 22BEC1205

DATE: 013/01/2025

Aim:
Write a Verilog RTL code for Parallel and Array Multipliers. Also, perform the simulation of the Multipliers using Quartus Prime and Model Sim. Finally, implement these circuits on the FPGA kit, “5CSXFC6D6F31C6N”

Software Required: Quartus Prime**,** ModelSim

Haraware Required: Altera Cyclone V 5CSXFC6D6F31C6N

Procedure:

- Open Quartus Prime 21.1.
- Go to File -> New Project wizard -> select Source folder & type file name -> Next.
- Select Empty Project in Project type -> Next.
- Click on Next in Add Files dialog box.
- Select “Cyclone V SX Extended Features” in Device Family.
- In available devices, select the one ending with “31C6” -> Next.
- Select the tool name as “ModelSim” and Format as “VerilogHDL” in simulation.
- Click on Finish. The project is now created.
- In the Task window, select RTL simulation and run, this would open the ModelSim window.
- Simulate the full adder as you would do using ModelSim by forcing the input values.
- In Compilation -> select compile design.
- Go to Assignments tab -> Pin planner -> Give the location for each i/o pin.
- Go to Hardware Setup -> Select USB.
- Change the file to Fulladder.v in the program/configure option and select Start.

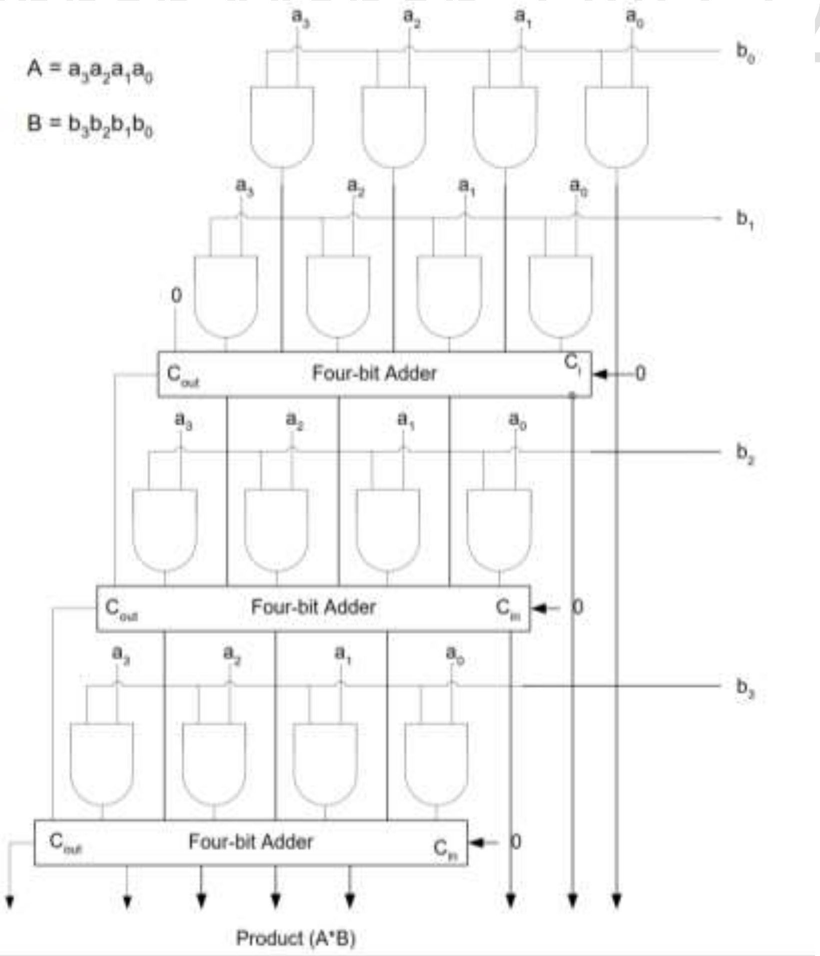
Theory

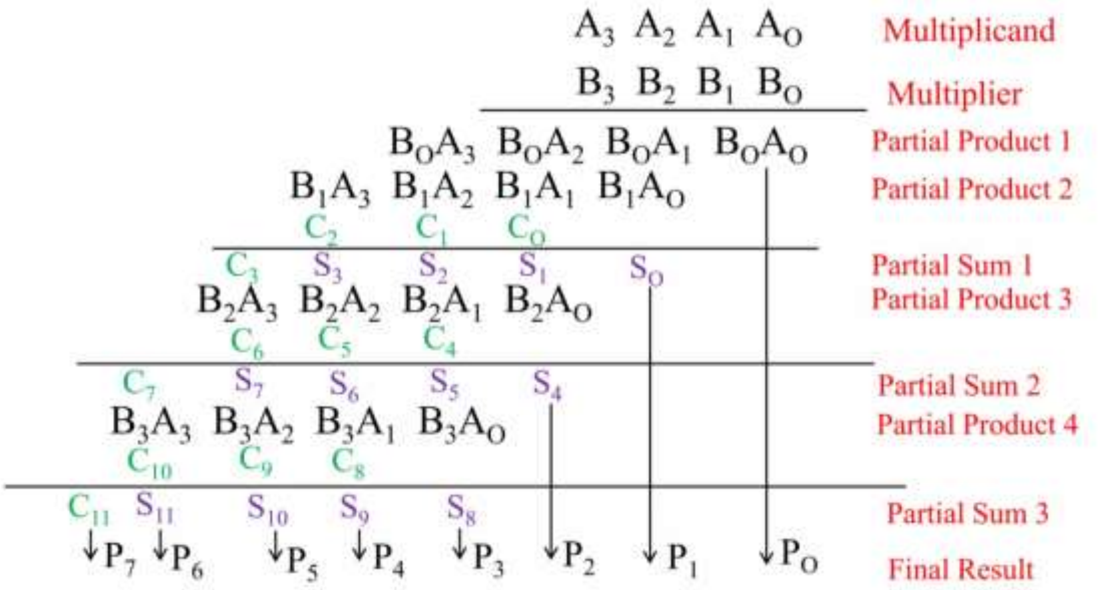
Parallel Multiplier:

A parallel multiplier is a combinational logic circuit used in digital systems to perform the multiplication of two binary numbers. Binary multiplication is similar to the decimal multiplication. In the first step of the process of the parallel multiplier, the partial product terms are obtained by the bit by bit multiplication, which is equal to the ANDing of two binary numbers. In the next step, all the partial product terms of each column are added together to get the final binary product output.

The logic circuit design of this multiplier varies with bit size and its complexity increases with an increase in the multiplier's bit size. It is governed by the AND gate functions when the two bits, which are to be multiplied, are fed as inputs with various bit sizes.

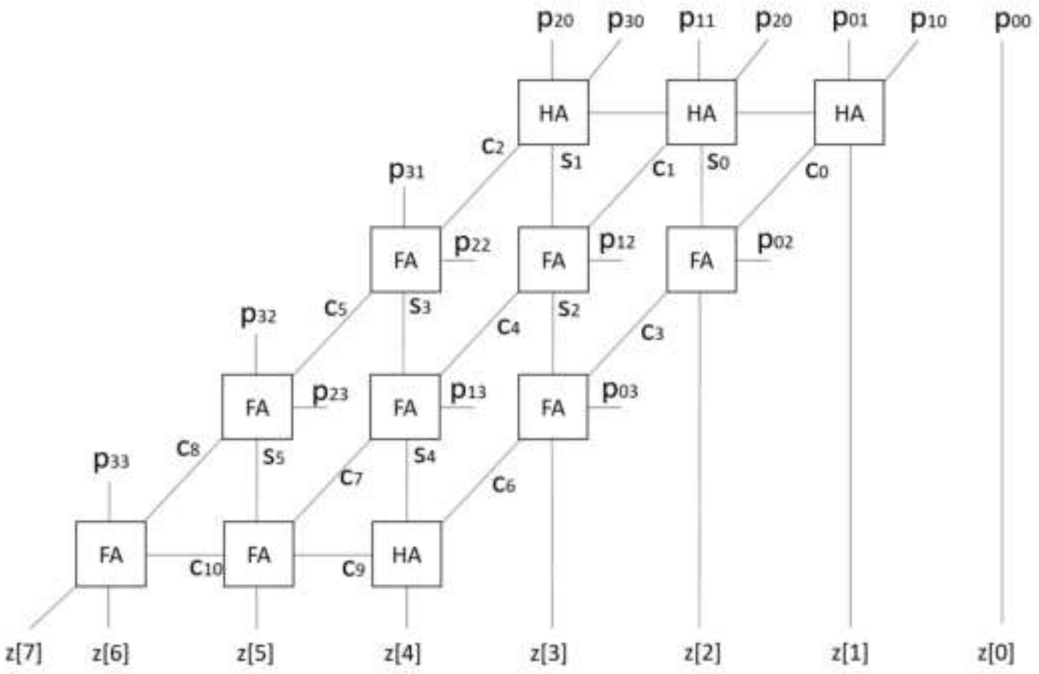
The main function of this multiplier is to do binary multiplication of 2 binary numbers with various bit sizes and reduce the calculation time in electronic digital systems such as computers. However, since n*n-1 full adders will be used during the addition stage, there will be a considerable increase in the delay of the circuit with increase in the sizes of the input binary numbers.





Array Multiplier:

Array multiplier is similar to how we perform multiplication with pen and paper i.e. finding a partial product and adding them together. It is simple architecture for implementation. A significant difference of the Array Multiplier when compared to the Parallel Multiplier is that it is more resource efficient as it uses half adders wherever possible instead of full adders. This helps to reduce redundancy, as the use of logic resources is more optimized to the requirements of multiplication.



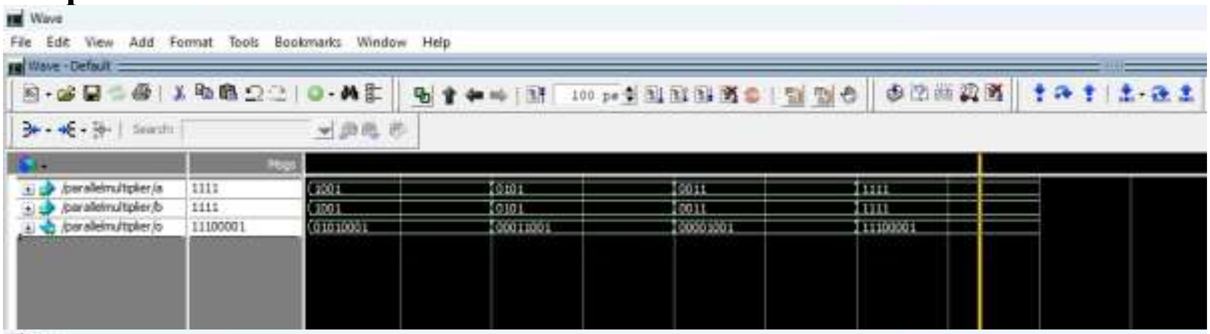
Array Multiplier

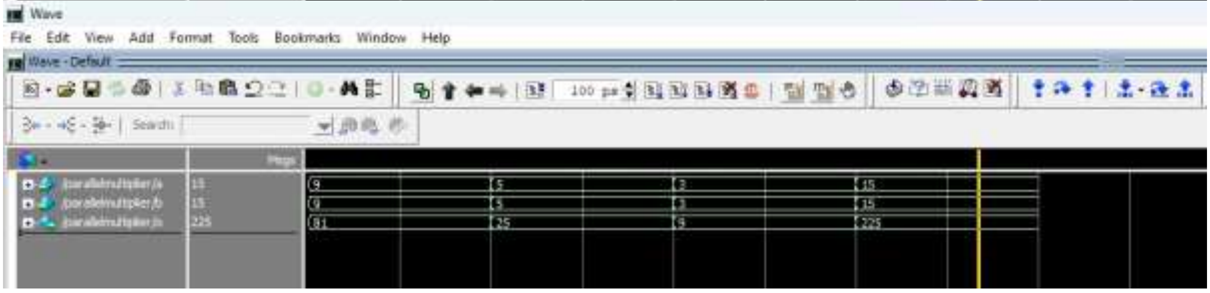
Source code and Outputs:

Parallel Multiplier:

```
module parallelmultiplier(input [3:0]a,b, output reg [7:0]o);
reg [3:0]ands[3:0];
reg [3:0]sums[2:0];
reg [4:0]cars[2:0];
integer i,j;

always @(*) begin
for(i=0; i<4; i=i+1) begin
for(j=0; j<4; j=j+1) begin
ands[i][j]=a[i]&b[j];
end
end

for(i=0; i<3; i=i+1) begin
cars[i][0]=0;
end

for(i=0; i<3; i=i+1)begin
sums[0][i]=(ands[0][i+1])^(ands[1][i])^(cars[0][i]);
cars[0][i+1]=(ands[0][i+1]&ands[1][i])|(ands[1][i]&cars[0][i])|(cars[0][i]&ands[0][i+1]);
end
sums[0][3]=(ands[1][3])^(cars[0][3]^1'b0);
cars[0][4]=(ands[1][3]&cars[0][3])|(cars[0][3]&1'b0)|(1'b0&ands[1][3]);

for(j=1; j<3; j=j+1)begin
for(i=0; i<3; i=i+1)begin
sums[j][i]=(sums[j-1][i+1])^(ands[j+1][i])^(cars[j][i]);
cars[j][i+1]=(sums[j-1][i+1]&ands[j+1][i])|(ands[j+1][i]&cars[j][i])|(cars[j][i]&sums[j-
1][i+1]);
end
sums[j][3]=(ands[j+1][3])^(cars[j][3])^(cars[j-1][4]);
cars[j][4]=(ands[j+1][3]&cars[j][3])|(cars[j][3]&cars[j-1][4])|(cars[j-1][4]&ands[j+1][3]);
end

o[7:0]={cars[2][4],sums[2][3:0],sums[1][0],sums[0][0],ands[0][0]};
end
endmodule
```

Output:



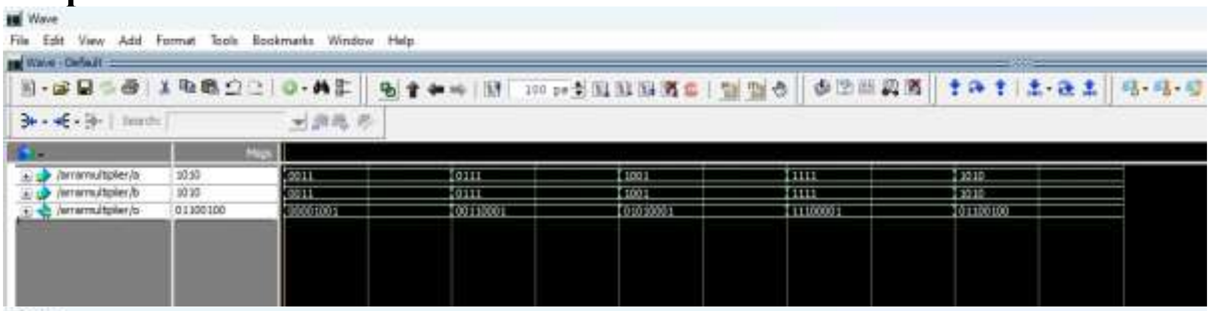


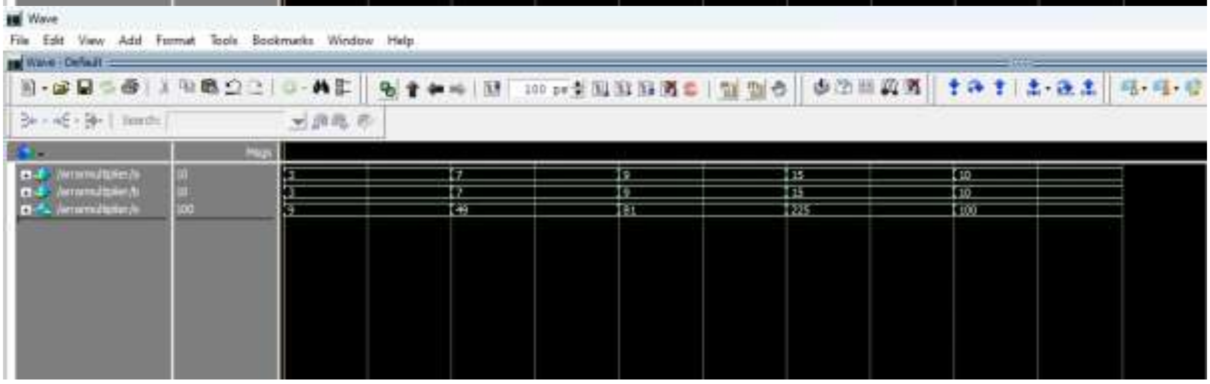
2. Array Multiplier:

```
module arrarmultiplier(input [3:0]a,b, output reg [7:0]o);
reg [3:0]ands[3:0];
reg [2:0]sums[3:0];
reg [2:0]cars[3:0];
integer i,j;

always @(*)
begin
for(i=0; i<4; i=i+1)begin
for(j=0; j<4; j=j+1)begin
ands[i][j]=a[i]&b[j];
end
end

for(i=0; i<3; i=i+1)begin //initial stage
sums[0][i]=ands[0][i+1]^ands[1][i];
cars[0][i]=ands[0][i+1]&ands[1][i];
end

for(i=1; i<3; i=i+1)begin //intermediate stages
for(j=0; j<2; j=j+1)begin
sums[i][j]=ands[i+1][j]^cars[i-1][j]^sums[i-1][j+1];
cars[i][j]=(ands[i+1][j]&cars[i-1][j])|(cars[i-1][j]&sums[i-1][j+1])|(sums[i-
1][j+1]&ands[i+1][j]);
end
sums[i][2]=ands[i+1][2]^cars[i-1][2]^ands[i][3];
cars[i][2]=(ands[i+1][2]&cars[i-1][2])|(cars[i-1][2]&ands[i][3])|(ands[i][3]&ands[i+1][2]);
end

//final stage
```

```
sums[3][0]=sums[2][1]^cars[2][0];
cars[3][0]=sums[2][1]&cars[2][0];
sums[3][1]=cars[3][0]^cars[2][1]^sums[2][2];
cars[3][1]=(cars[3][0]&cars[2][1])|(cars[2][1]&sums[2][2])|(sums[2][2]&cars[3][0]);
sums[3][2]=cars[3][1]^cars[2][2]^ands[3][3];
cars[3][2]=(cars[3][1]&cars[2][2])|(cars[2][2]&ands[3][3])|(ands[3][3]&cars[3][1]);

o[7:0]={cars[3][2],sums[3][2:0],sums[2][0],sums[1][0],sums[0][0],ands[0][0]};
end
endmodule
```

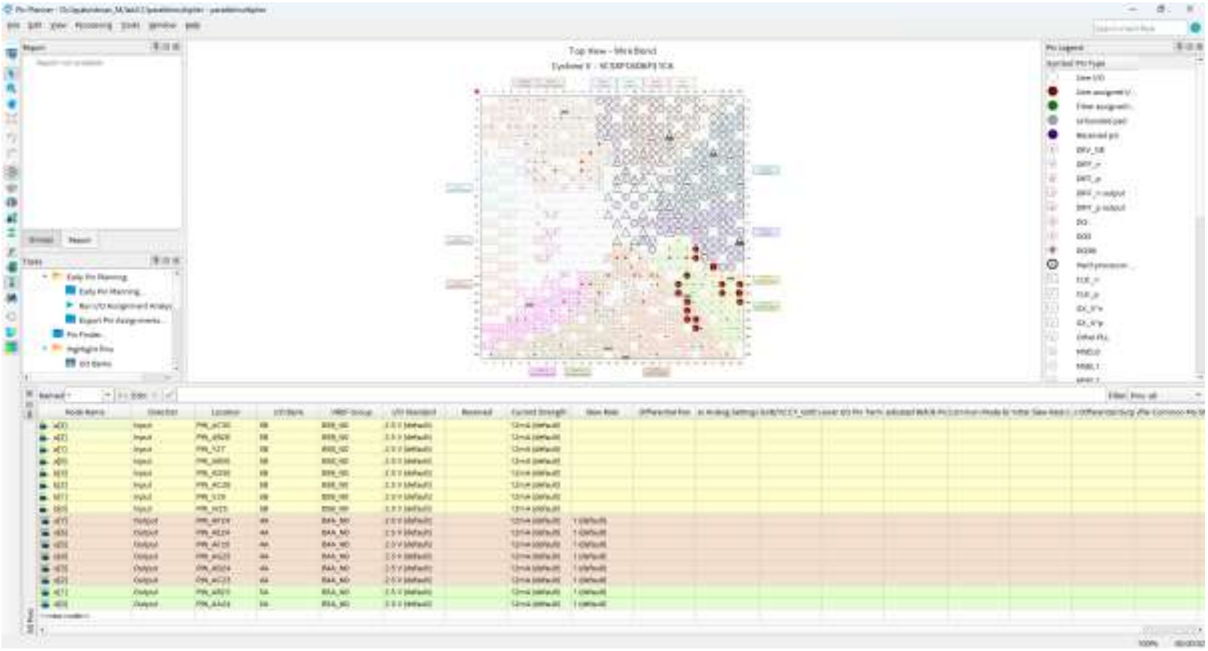
Output:



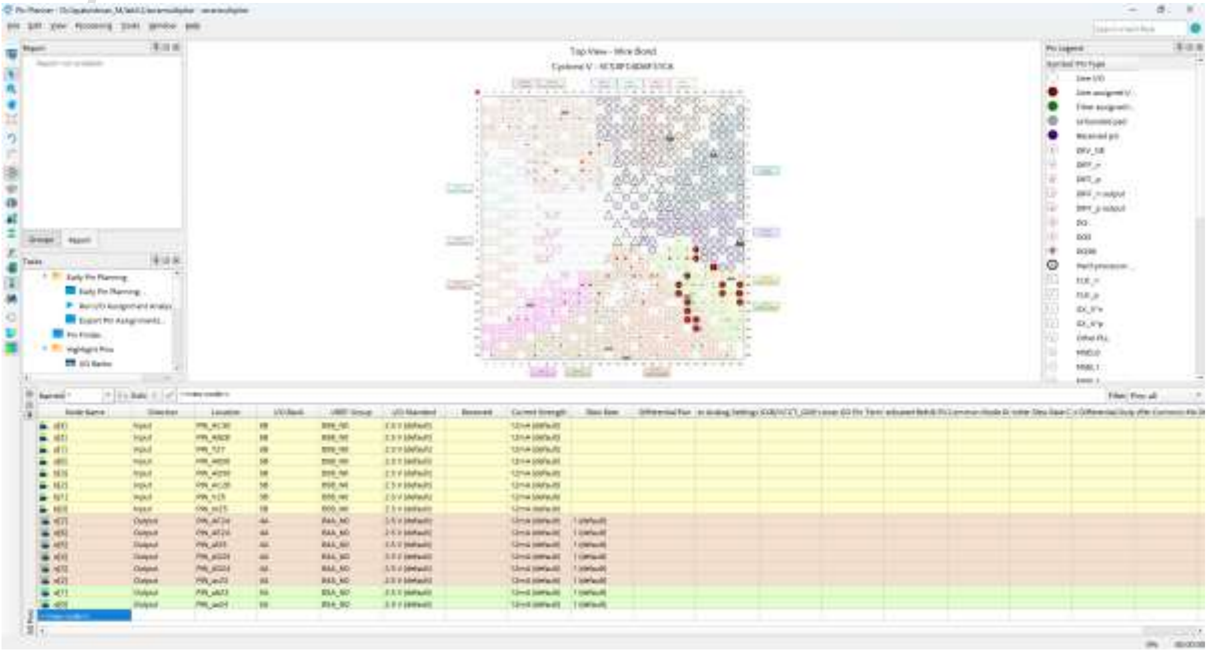


Pin Planning:

Chosen I/O for Parallel Multiplier:

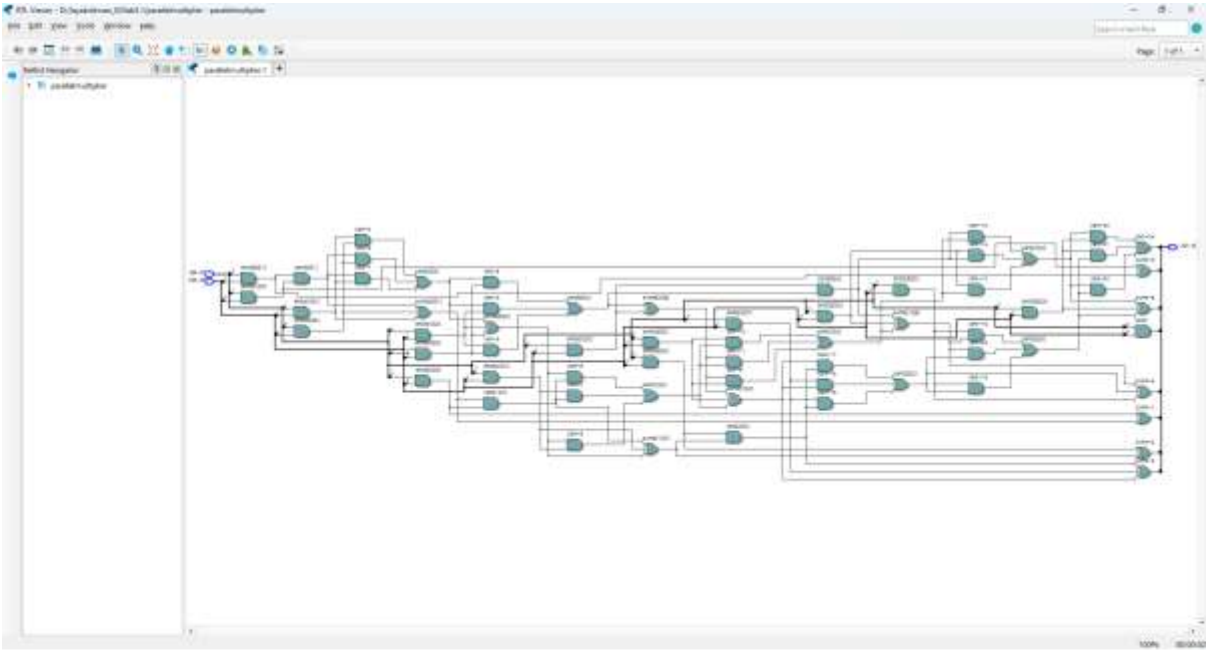


Chosen I/O for Array Multiplier:

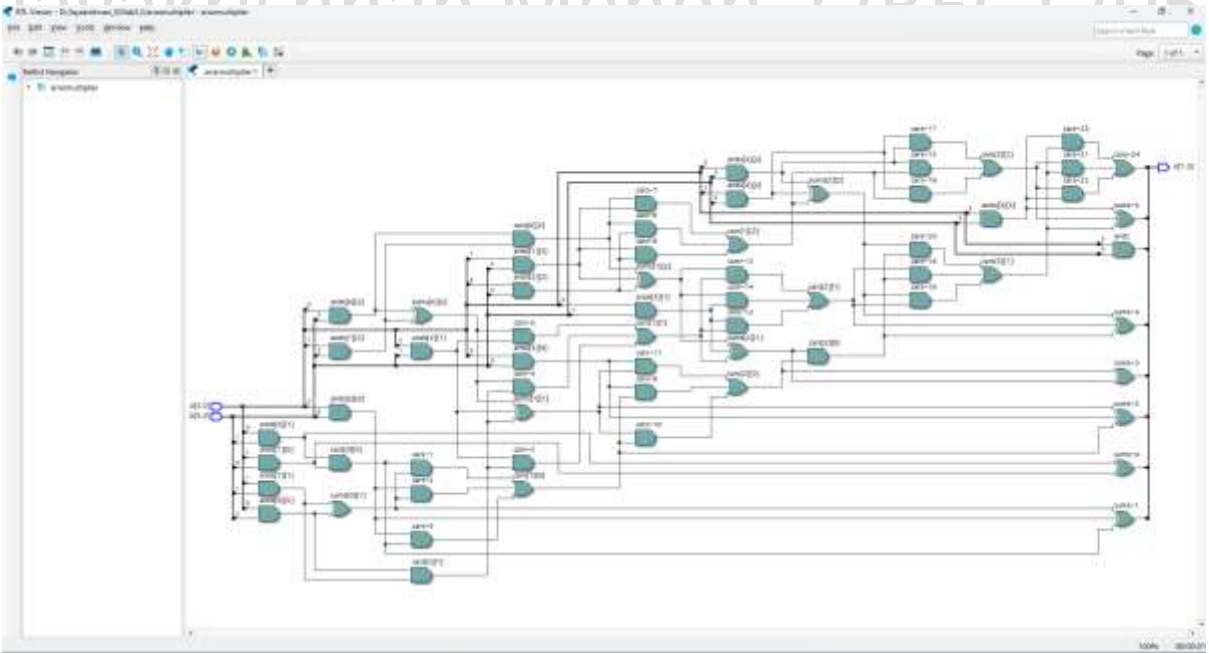


RTL Diagrams:

Parallel Multiplier:



Array Multiplier:



Hardware Implementation:

Parallel Multiplier:

Input: a=1001, b=1001

Output: c=01010001 (9x9=81)

Input: a=0101, b=0101

Output: c=00011001 (5x5=25)

Array Multiplier:

Input: a=0111, b=0111

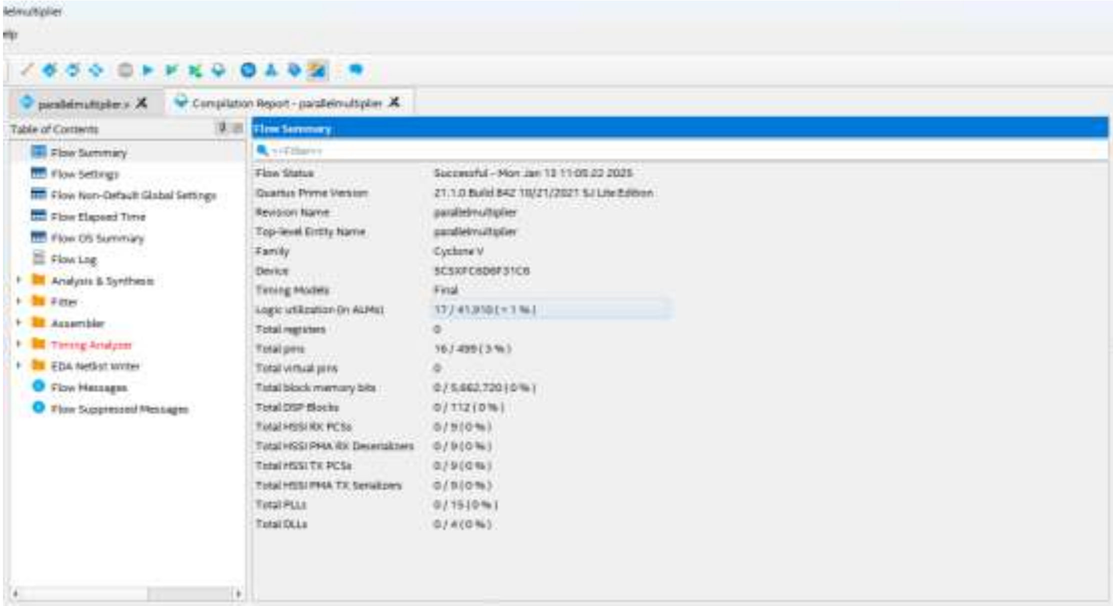
Output: c=00110001 (7x7=49)

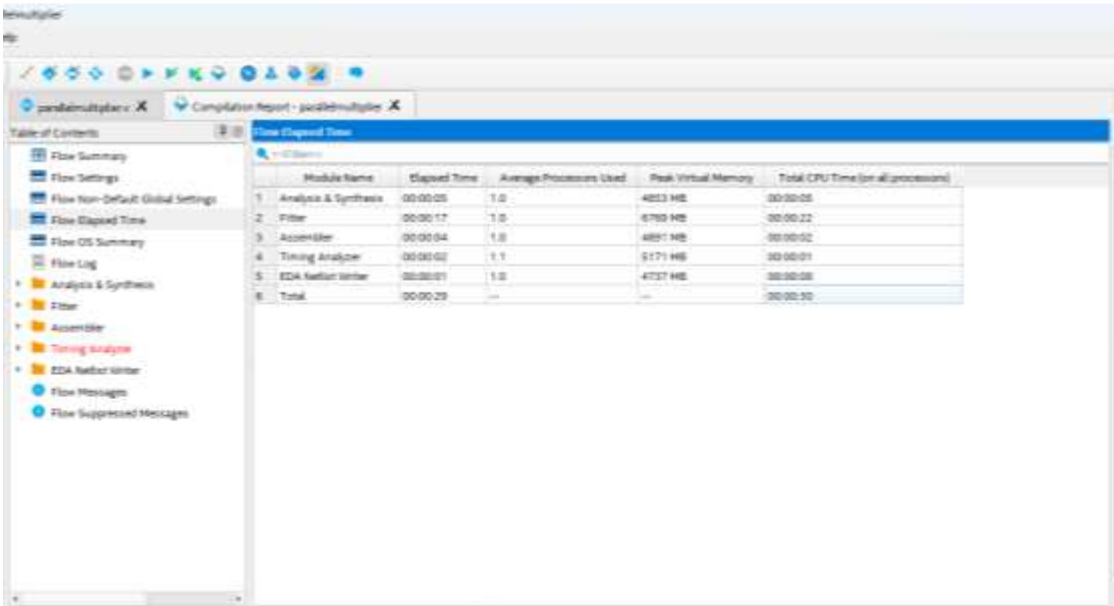
Input: a=1010, b=1010

Output: c=01100100 (10x10=100)

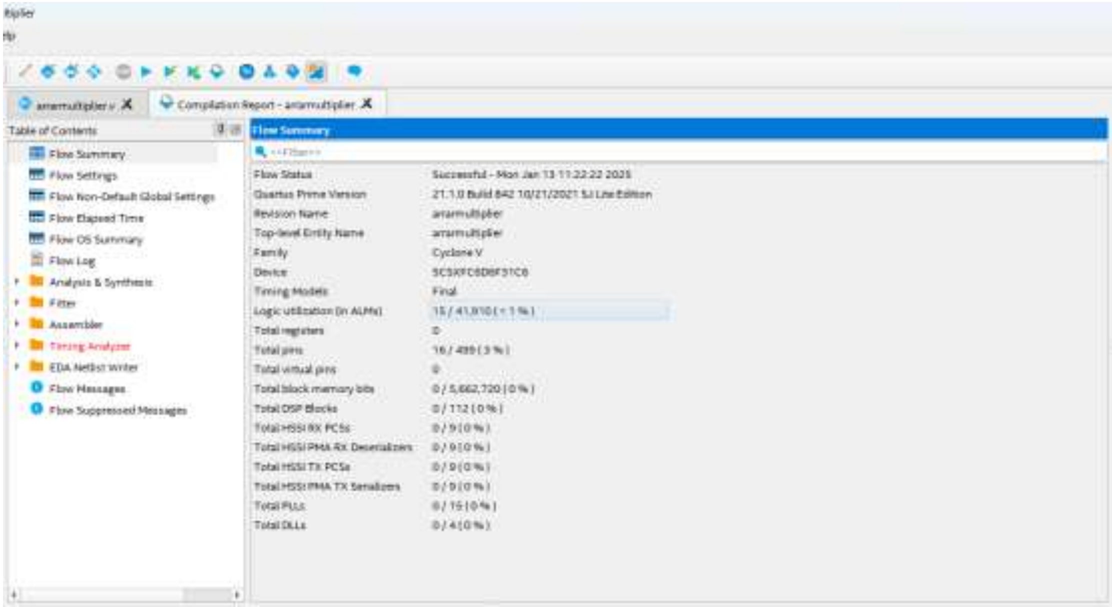
Logic utilization and Timing Analysis:

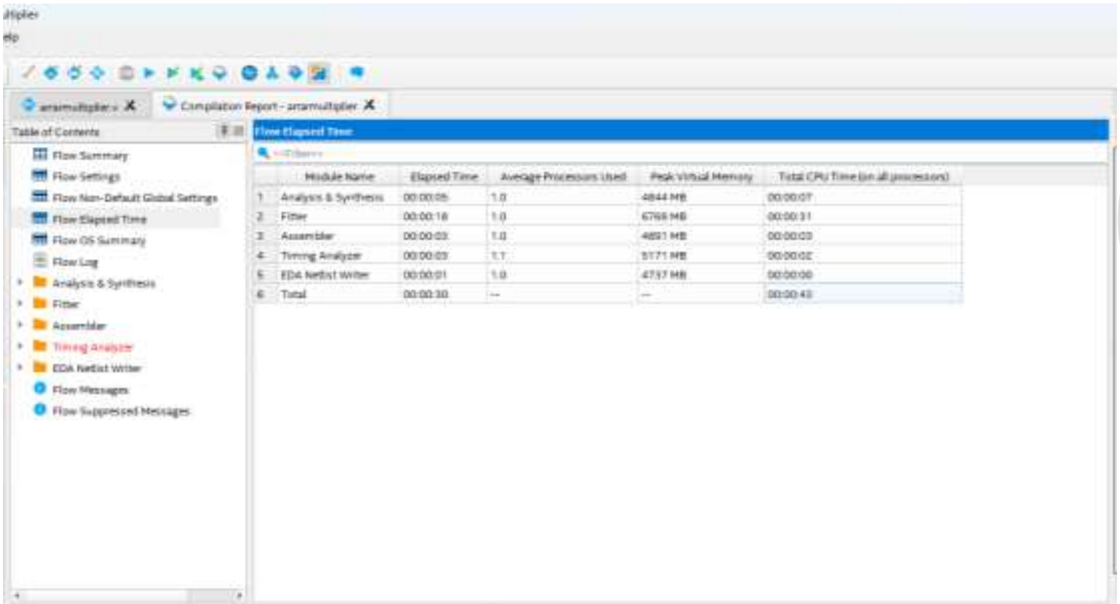
Parallel Multiplier:





Array Multiplier:





05

Result:

Hence, Verilog RTL code for a Parallel Multiplier and an Array Multiplier, were successfully verified and implemented using Quartus Prime. Logic Utilization and Timing Analysis were also observed for both of the circuits.