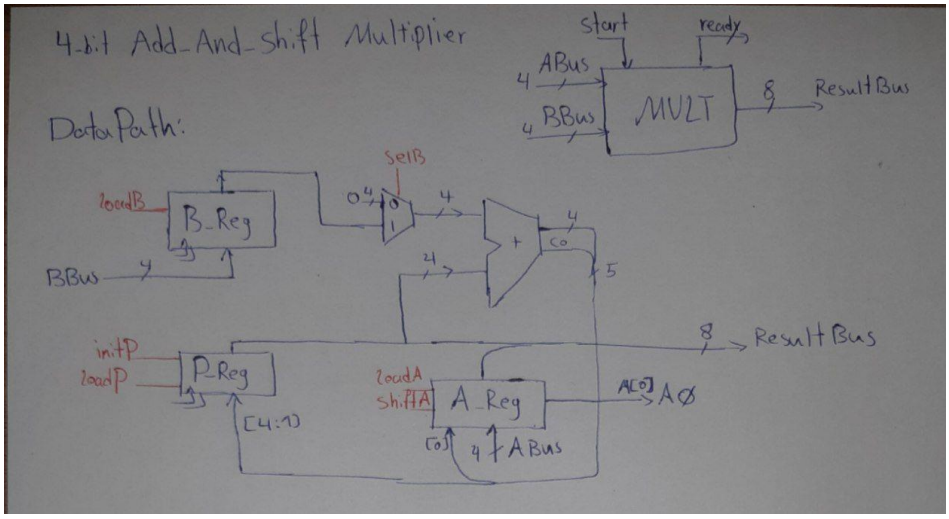
Object Oriented Modeling of Electronic Circuits – Spring 1402-03
Computer Assignment 1

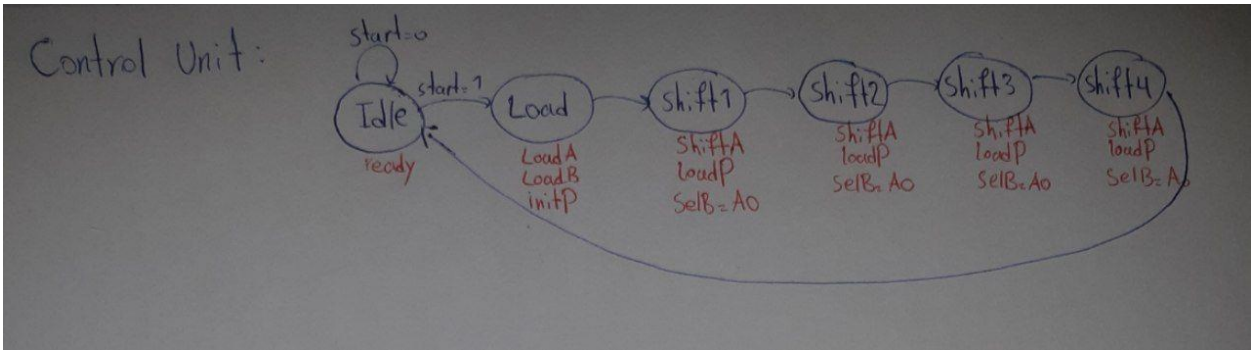
Amir Hosein Samoudi – 810100108

Part 1:

We have designed a 4 bit Add and Shift Multiplier in RTL level and described it in Verilog.



1. datapath design



2- Control Unit FSM design

Data path description is as below:

```
module Add_Shift_DP(
    input clk, rst, initP, loadP, loadA, shiftA, loadB, selB,
    input [3:0] ABus, BBus,
    input start,
```

```
    output [7:0] resultBus,
    output A0
);

    reg [3:0] Areg, Breg, Preg;
    wire [4:0] Add_result;
    wire [3:0] adder_input1;
    assign adder_input1 = selB ? Breg : 4'b0;
    assign Add_result = adder_input1 + Preg;
    assign resultBus = {Preg, Areg};
    assign A0 = Areg[0];
    always @(posedge clk, posedge rst) begin
        if (rst == 1'b1)
            Areg <= 4'b0;
        else begin
            if (loadA == 1'b1)
                Areg <= ABus;
            else if (shiftA == 1'b1)
                Areg <= {Add_result[0], Areg[3:1]};
        end
    end
    always @(posedge clk, posedge rst) begin
        if (rst == 1'b1)
            Breg <= 4'b0;
        else if (loadB == 1'b1)
            Breg <= BBus;
    end
    always @(posedge clk, posedge rst) begin
        if (rst == 1'b1)
            Preg <= 4'b0;
        else begin
            if (initP == 1'b1)
                Preg <= 4'b0;
            else if (loadP == 1'b1)
                Preg <= Add_result[4:1];
        end
    end
endmodule
```

controller description is as below:

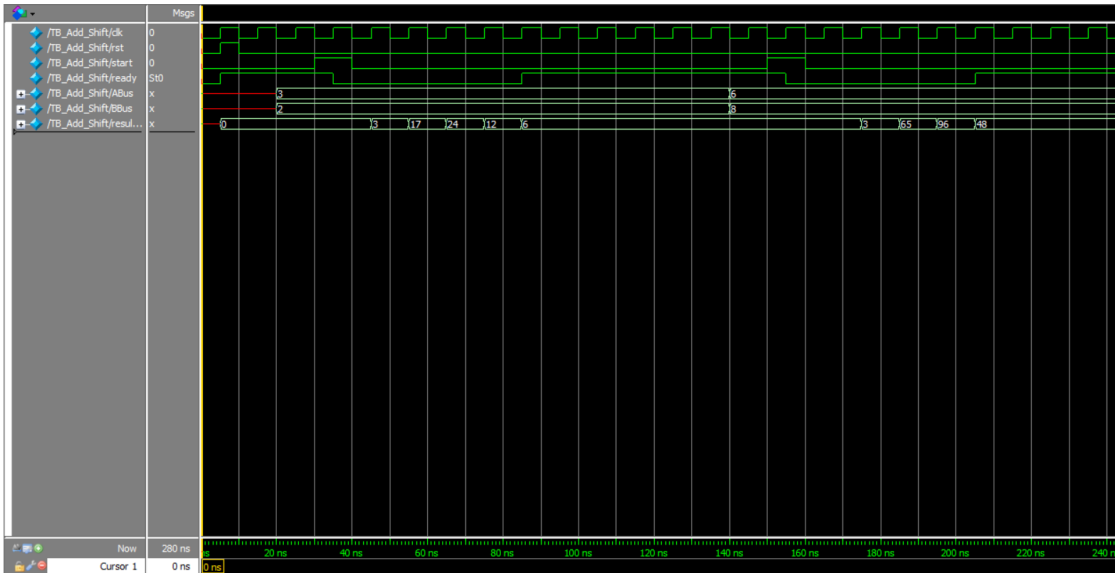
```
module Add_Shift_CU(
    input clk, rst, start, A0,
    output reg initP, loadP, loadA, shiftA, loadB, selB, ready
);
```

```
    parameter [1:0]  Idle = 0, Load = 1, Shift = 3;
    reg [1:0] ps, ns;
    reg init_Cnt, cen_Cnt;
    wire Co;
    reg [1:0] PO;
    always @(ps, start, A0, Co) begin
        ns = 2'b0;
        {initP, loadP, loadA, shiftA, loadB, selB, ready, init_Cnt, cen_Cnt} = 9'b0;
        case (ps)
           Idle : begin
                    ns = start ? Load : Idle;
                    ready = 1'b1;
                end
           Load : begin
                    ns = Shift;
                    {loadA, loadB, initP, init_Cnt} = 4'b1111;
                end
           Shift : begin
                    ns = Co ? Idle : Shift;
                    {shiftA, loadP, cen_Cnt} = 3'b111;
                    selB = A0;
                end
            default: ns = Idle;
        endcase
    end
    always @(posedge clk, posedge rst) begin
        if (rst)
            ps <= Idle;
        else
            ps <= ns;
    end

    always @(posedge clk, posedge rst) begin
        if (rst)
            PO <= 2'b0;
        else begin
            if (init_Cnt)
                PO <= 2'b0;
            else if (cen_Cnt)
                PO <= PO + 1;
        end
    end
    assign Co = &(PO);
endmodule
```

Now to get an integrated top-module we have merged the data-path and controller descriptions in one top-module (called “Add_Shift_Mult.v”)

Then we simulated the design for 2 different set of inputs.



3. Simulation results

As we can see it is working properly.

Part 2:

In this part we have synthesized the RTL design by Yosys using only the AND, OR, NOT gate (gates that we used in Assignment 1) and Flip-Flops for the sequential parts.

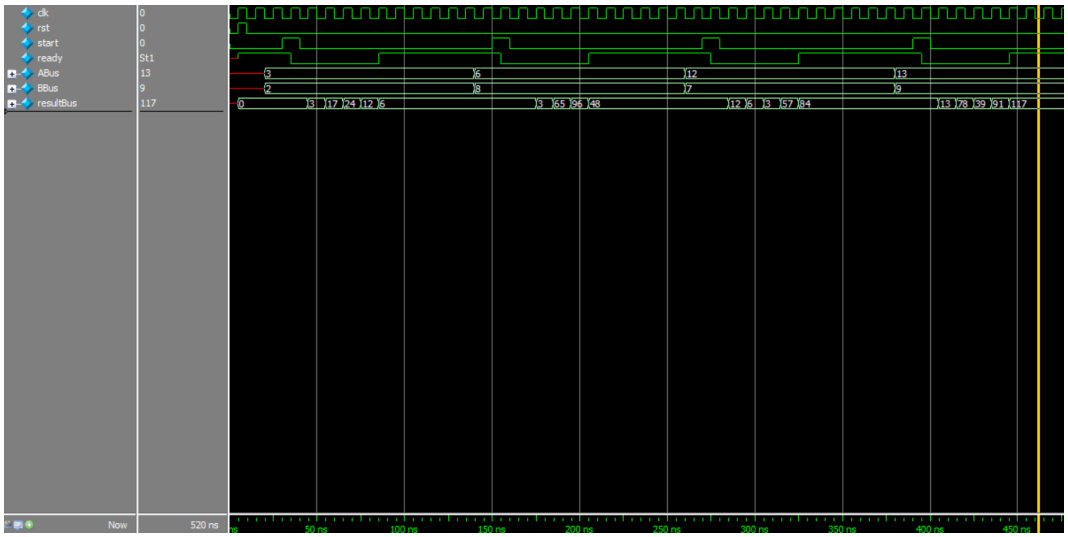
```
3.1.2. Re-integrating ABC results.
ABC RESULTS:               AND cells:       55
ABC RESULTS:               NOT cells:       14
ABC RESULTS:                OR cells:       45
ABC RESULTS:        internal signals:       46
ABC RESULTS:           input signals:       27
ABC RESULTS:          output signals:       14
Removing temp directory.

yosys>
```

4- Yosys' post-synthesis report

Part 3:

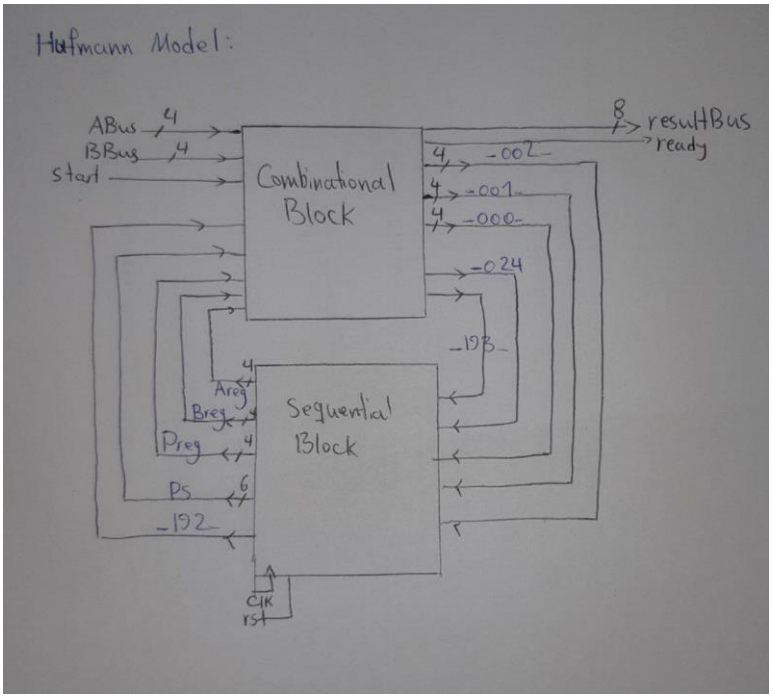
We tested the post-syntheses circuit in ModelSim .



5. Post-synthesis simulation

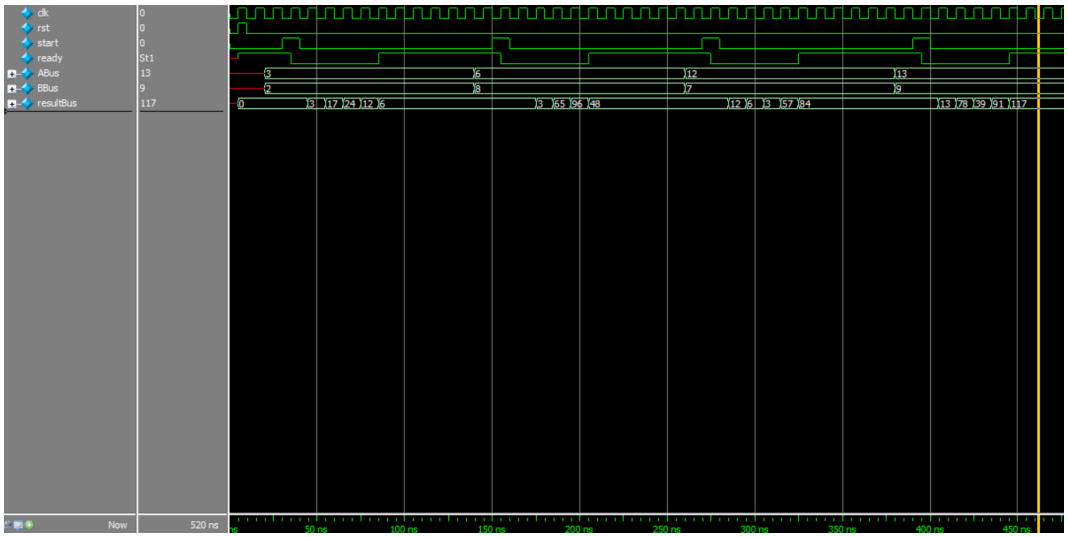
As we can see it is matching the results of part 1.

Now to perform a better simulation in c++ for the upcoming parts, we have break our design into the Combinational and Sequential parts (Huffman model).



6- Huffman model of Add-Shift Multiplier

Now we have gotten instantiations of this two modules and performed a simulation to verify it.



7- Huffman model simulation

It fully matches the previous simulations.

Part 4:

Our goal is to write a program that gets a list of the Yosys generated gate (which don't necessary have an order in simulation and are concurrent) and sort it in a way that we would be able to evaluate each gate separately and somehow resolve the fact that we don't have concurrency in C++.

After sorting the gates, we have run the simulation once again and it matched our expectations.

Part 5:

In this section we used the library that includes below gates:

```
void AND(char a, char b, char& w)
{
    if ((a == '0') || (b == '0'))
        w = '0';
    else if ((a == '1') && (b == '1'))
        w = '1';
    else
        w = 'X';
}
void OR(char a, char b, char& w)
{
    if ((a == '1') || (b == '1'))
        w = '1';
    else if ((a == '0') && (b == '0'))
        w = '0';
    else
        w = 'X';
}
```

```
void NOT(char a, char& w)
{
    if (a == '1')
        w = '0';
    else if (a == '0')
        w = '1';
    else
        w = 'X';
}
```

Then we have modified the Verilog code and converted all the Yosys' output file gates to the above gate models and wrote two functions one for the combinational part and one for sequential part.

Part 6:

We have developed a test-bench to verify our C++ model of Add-Shift multiplier.

Inputs are extracted from a csv format file with the other of inputs as "clk,rst,start,ABus,BBus".

We have tested the model for various input datasets.

```
1   C.C = 1 : Clock = P  ,Reset = 0  ,start = 0  ,Mult Result = XXXXXXXX,  ready = X
2   C.C = 2 : Clock = 1  ,Reset = 1  ,start = 0  ,Mult Result = XXXXXXXX,  ready = X
3   C.C = 3 : Clock = 0  ,Reset = 0  ,start = 1  ,Mult Result = 00000000,  ready = 1
4   C.C = 4 : Clock = P  ,Reset = 0  ,start = 1  ,Mult Result = 00000000,  ready = 1
5   C.C = 5 : Clock = 1  ,Reset = 0  ,start = 0  ,Mult Result = 00000000,  ready = 0
6   C.C = 6 : Clock = 0  ,Reset = 0  ,start = 0  ,Mult Result = 00000000,  ready = 0
7   C.C = 7 : Clock = P  ,Reset = 0  ,start = 0  ,Mult Result = 00000000,  ready = 0
8   C.C = 8 : Clock = 1  ,Reset = 0  ,start = 0  ,Mult Result = 00000110,  ready = 0
9   C.C = 9 : Clock = 0  ,Reset = 0  ,start = 0  ,Mult Result = 00000110,  ready = 0
10  C.C = 10 : Clock = P  ,Reset = 0  ,start = 0  ,Mult Result = 00000110,  ready = 0
11  C.C = 11 : Clock = 1  ,Reset = 0  ,start = 0  ,Mult Result = 00000011,  ready = 0
12  C.C = 12 : Clock = 0  ,Reset = 0  ,start = 0  ,Mult Result = 00000011,  ready = 0
13  C.C = 13 : Clock = P  ,Reset = 0  ,start = 0  ,Mult Result = 00000011,  ready = 0
14  C.C = 14 : Clock = 1  ,Reset = 0  ,start = 0  ,Mult Result = 01000001,  ready = 0
15  C.C = 15 : Clock = 0  ,Reset = 0  ,start = 0  ,Mult Result = 01000001,  ready = 0
16  C.C = 16 : Clock = P  ,Reset = 0  ,start = 0  ,Mult Result = 01000001,  ready = 0
17  C.C = 17 : Clock = 1  ,Reset = 0  ,start = 0  ,Mult Result = 01100000,  ready = 0
18  C.C = 18 : Clock = 0  ,Reset = 0  ,start = 0  ,Mult Result = 01100000,  ready = 0
19  C.C = 19 : Clock = P  ,Reset = 0  ,start = 0  ,Mult Result = 01100000,  ready = 0
20  C.C = 20 : Clock = 1  ,Reset = 0  ,start = 0  ,Mult Result = 00110000,  ready = 1
21  C.C = 21 : Clock = 0  ,Reset = 0  ,start = 0  ,Mult Result = 00110000,  ready = 1
22  C.C = 22 : Clock = P  ,Reset = 0  ,start = 0  ,Mult Result = 00110000,  ready = 1
23  C.C = 23 : Clock = 1  ,Reset = 0  ,start = 0  ,Mult Result = 00110000,  ready = 1
24  C.C = 24 : Clock = 0  ,Reset = 0  ,start = 0  ,Mult Result = 00110000,  ready = 1
```

8- Cpp model for inputs 6 and 8

```
C.C = 1 : Clock = P  ,Reset = 0  ,start = 0  ,Mult Result = XXXXXXXX,  ready = X
C.C = 2 : Clock = 1  ,Reset = 1  ,start = 0  ,Mult Result = XXXXXXXX,  ready = X
C.C = 3 : Clock = 0  ,Reset = 0  ,start = 1  ,Mult Result = 00000000,  ready = 1
C.C = 4 : Clock = P  ,Reset = 0  ,start = 1  ,Mult Result = 00000000,  ready = 1
C.C = 5 : Clock = 1  ,Reset = 0  ,start = 0  ,Mult Result = 00000000,  ready = 0
C.C = 6 : Clock = 0  ,Reset = 0  ,start = 0  ,Mult Result = 00000000,  ready = 0
C.C = 7 : Clock = P  ,Reset = 0  ,start = 0  ,Mult Result = 00000000,  ready = 0
C.C = 8 : Clock = 1  ,Reset = 0  ,start = 0  ,Mult Result = 00001100,  ready = 0
C.C = 9 : Clock = 0  ,Reset = 0  ,start = 0  ,Mult Result = 00001100,  ready = 0
C.C = 10 : Clock = P  ,Reset = 0  ,start = 0  ,Mult Result = 00001100,  ready = 0
C.C = 11 : Clock = 1  ,Reset = 0  ,start = 0  ,Mult Result = 00000110,  ready = 0
C.C = 12 : Clock = 0  ,Reset = 0  ,start = 0  ,Mult Result = 00000110,  ready = 0
C.C = 13 : Clock = P  ,Reset = 0  ,start = 0  ,Mult Result = 00000110,  ready = 0
C.C = 14 : Clock = 1  ,Reset = 0  ,start = 0  ,Mult Result = 00000011,  ready = 0
C.C = 15 : Clock = 0  ,Reset = 0  ,start = 0  ,Mult Result = 00000011,  ready = 0
C.C = 16 : Clock = P  ,Reset = 0  ,start = 0  ,Mult Result = 00000011,  ready = 0
C.C = 17 : Clock = 1  ,Reset = 0  ,start = 0  ,Mult Result = 00111001,  ready = 0
C.C = 18 : Clock = 0  ,Reset = 0  ,start = 0  ,Mult Result = 00111001,  ready = 0
C.C = 19 : Clock = P  ,Reset = 0  ,start = 0  ,Mult Result = 00111001,  ready = 0
C.C = 20 : Clock = 1  ,Reset = 0  ,start = 0  ,Mult Result = 01010100,  ready = 1
C.C = 21 : Clock = 0  ,Reset = 0  ,start = 0  ,Mult Result = 01010100,  ready = 1
C.C = 22 : Clock = P  ,Reset = 0  ,start = 0  ,Mult Result = 01010100,  ready = 1
C.C = 23 : Clock = 1  ,Reset = 0  ,start = 0  ,Mult Result = 01010100,  ready = 1
C.C = 24 : Clock = 0  ,Reset = 0  ,start = 0  ,Mult Result = 01010100,  ready = 1
```

9- Cpp model for inputs 7 and 12

Part 7:

In this part we have used the assignment 1 program to convert list of primitive gates (AND, OR, NOT) to the assign statement. We had to modify the code to handle assign statements (which does exists in Yosys generated code) and also handle the multi-bits wire (as both input and output).

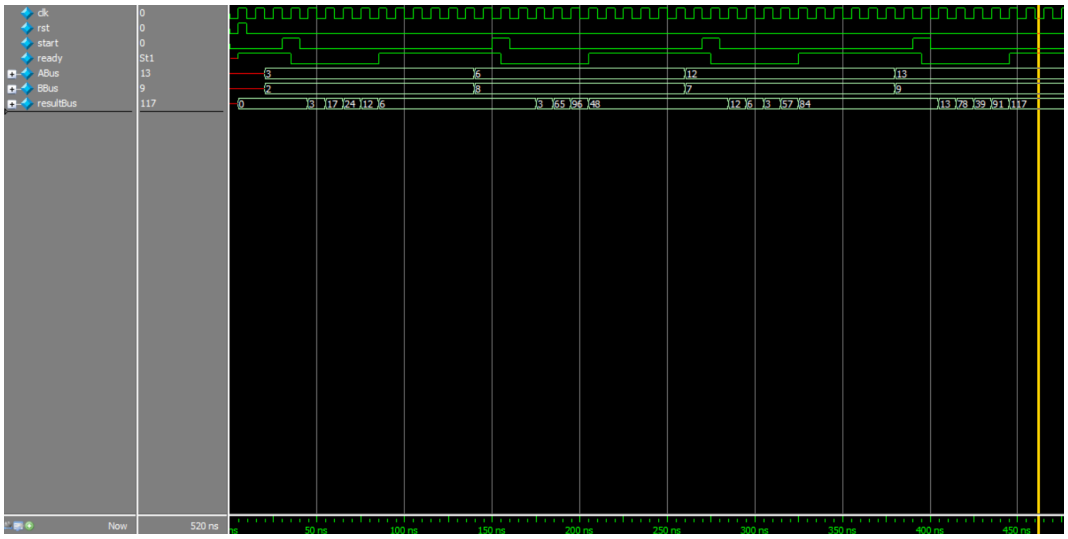
```
module Behavioral(
    input [5:0] ps,
    input _192_,
    input start,
    output _193_,
    output _024_,
    output [3:0] _000_,
    output [3:0] _001_,
    output [3:0] _002_,
    output ready,
    output [7:0] resultBus
);
    assign _193_ = ((((_192_) | (start)) & (~((ps[4])))));
    assign _024_ = (((~((_192_))) & (start)));
    assign _000_[0] = ((((ABus[0]) & (ps[3])) | ((((((ps[2]) | (ps[4])) | ((ps[1]) | (ps[5]))) & (Areg[1])) | ((((~((ps[2]))) & (~((ps[4])))) &
    assign _000_[1] = ((((ABus[1]) & (ps[3])) | ((((((ps[2]) | (ps[4])) | ((ps[1]) | (ps[5]))) & (Areg[2])) | ((((~((ps[2]))) & (~((ps[4])))) &
    assign _000_[2] = ((((ABus[2]) & (ps[3])) | ((((((ps[2]) | (ps[4])) | ((ps[1]) | (ps[5]))) & (Areg[3])) | ((((~((ps[2]))) & (~((ps[4])))) &
    assign _000_[3] = (((((((~((ps[2]))) & (~((ps[4])))) & ((~((ps[1]))) & (~((ps[5]))))) & (Areg[3])) | (ps[3])) | (((~(((((((ps[2]) | (ps[4])
    assign _001_[0] = ((((BBus[0]) | (~((ps[3])))) & ((ps[3]) | (Breg[0]))));
    assign _001_[1] = ((((BBus[1]) | (~((ps[3])))) & ((ps[3]) | (Breg[1]))));
    assign _001_[2] = ((((BBus[2]) | (~((ps[3])))) & ((Breg[2]) | (ps[3]))));
    assign _001_[3] = ((((BBus[3]) | (~((ps[3])))) & ((Breg[3]) | (ps[3]))));
    assign _002_[0] = ((((((ps[2]) | (ps[4])) | ((ps[1]) | (ps[5]))) | (Preg[0])) & (~((ps[3])))) & ((((((((~((ps[2]))) & (~((ps[4])))) & ((~
    assign _002_[1] = ((((((ps[2]) | (ps[4])) | ((ps[1]) | (ps[5]))) | (Preg[1])) & (~((ps[3])))) & ((((~(((((((ps[2]) | (ps[4])) | ((ps[1]) |
    assign _002_[2] = ((((((ps[2]) | (ps[4])) | ((ps[1]) | (ps[5]))) | (Preg[2])) & (~((ps[3])))) & ((((~(((((((ps[2]) | (ps[4])) | ((ps[1]) |
    assign _002_[3] = ((((((ps[2]) | (ps[4])) | ((ps[1]) | (ps[5]))) | (Preg[3])) & (~((ps[3])))) & (((((((ps[2]) | (ps[4])) | ((ps[1]) | (ps[
    assign ready = ((~((_192_))));
    assign resultBus[0] = (Areg[0]);
    assign resultBus[1] = (Areg[1]);
    assign resultBus[2] = (Areg[2]);
    assign resultBus[3] = (Areg[3]);
    assign resultBus[4] = (Preg[0]);
    assign resultBus[5] = (Preg[1]);
    assign resultBus[6] = (Preg[2]);
    assign resultBus[7] = (Preg[3]);

endmodule
```

10- Behavioral code of combinational part

It should be noted that some expressions are extremely long and figure 10 just show the overall format of the generated code.

As always we perform a simulation in this step to verify our work and as we expected the results are totally similar to the structural design of previous parts.



11- Behavioral circuit simulation result

Part 8:

In this part we used the code from part 6 and just change the combinational-part function definition with the new behavioral code and by doing the operator overloading we would be able to evaluate the expressions with Verilog syntax style.