하드웨어 시스템 및 설계

LAB-04

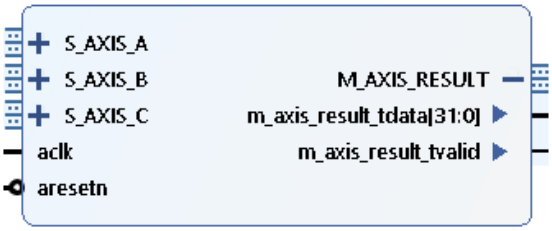
이다운

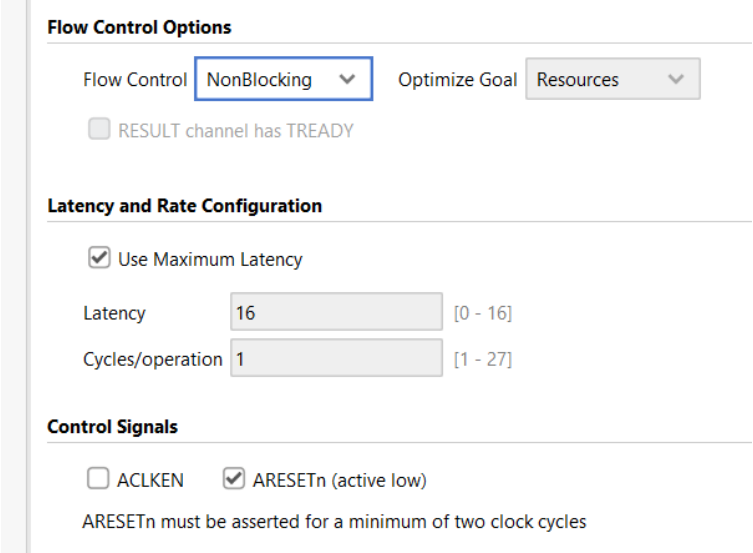
2016-13919

Floating point Multiply Adder

1. IP catalog & Testbench

floating point multiply adder는 Floating-point (7.1) IP catalog로 구현하였다.





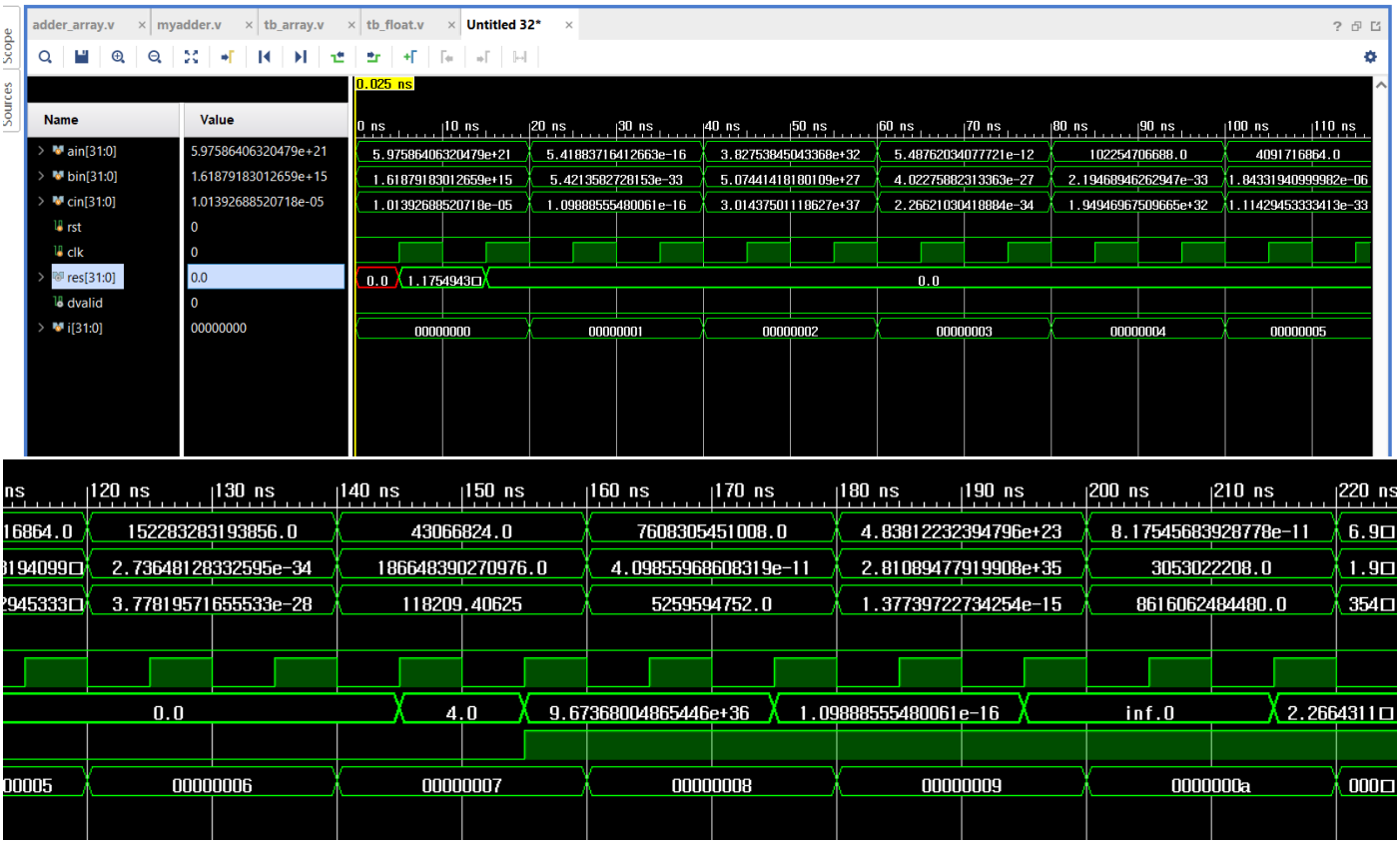
이 모듈의 input으로는 입력데이터가 될 S_AXIS_A, S_AXIS_B, S_AXIS_C 와 CLOCK인 aclk, 마지막으로 Active-Low synchronous clear 신호인 aresetn을 가진다. OUTPUT으로 결과 값인 M_AXIS_RESULT를 가진다. 이 계산을 result = a*b +c 가 된다.

오른쪽 사진처럼 이 모듈의 세부상황을 정했는데 Non_blocking 방법과 지연도는 최대값인 16사이클로 설정하였고 input A,B,C 와 output 모두 32bit one-precision방식으로 설정하였다

이 모듈이 잘 작동하는지 확인하기 위해서는 테스트벤치를 만들고 시뮬레이션을 돌려봐야한다. 테스트벤치는 아래와 같다.

```
module tb_float();

    reg [32-1:0] ain;
    reg [32-1:0] bin;
    reg [32-1:0] cin;
    reg rst;
    reg clk;
    wire[32-1:0] res;
    wire dvalid;

    integer i;

    initial begin
        clk<=0;
        rst<=0;
        for(i=0; i<50; i=i+1)begin
          ain = $urandom%(2**31);
          bin = $urandom%(2**31);
          cin = $urandom%(2**31);
          #20;
        end
    end

    always #5 clk=~clk;

floating_point_0 UUT(
    .aclk(clk),
    .aresetn(~rst),
    .s_axis_a_tvalid(1'b1),
    .s_axis_b_tvalid(1'b1),
    .s_axis_c_tvalid(1'b1),
    .s_axis_a_tdata(ain),
    .s_axis_b_tdata(bin),
    .s_axis_c_tdata(cin),
    .m_axis_result_tvalid (dvalid),
    .m_axis_result_tdata (res)
);

endmodule
```

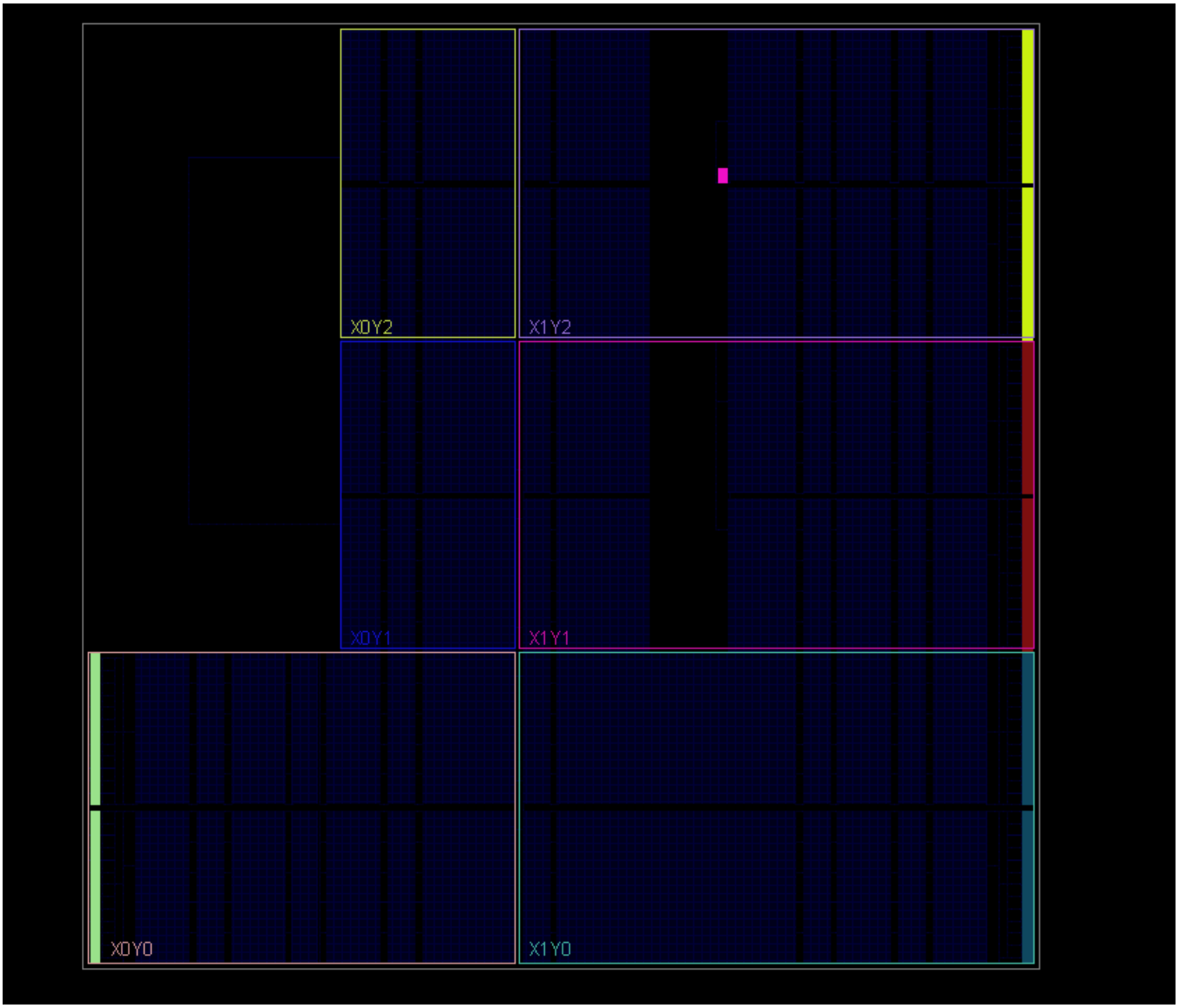
Testbench에 대해 간단히 설명하면 먼저 모듈 Testbench모듈 (tb_float) 선언 및 input, output으로 사용할 변수를 선언해 준다. 여기서 32bit를 사용하므로 [32-1:0] 을 붙여준다. 그리고 인풋값에 random한 값이 돌아가면서 들어올 수 있도록 for문을 돌리면서 urandom함수를 이용하여 대입한다. 이때 최대값을 정해주기 위해 %연산자를 사용하였다. 그리고 clk값이 주기적으로 바뀌게 하기위해 always문 안에 #5 clk= ~clk을 쓴다. 그리고 testbench와 ipcatalog을 연결할 수 있게 모듈을 작성한다.



testbench를 simulation을 돌리면 위와 같다. Radix를 single-precision으로 설정하면 e notation 방식으로 숫자가 표시된다.

2. Systhesis

Floating-point-adder를 top module로 설정하고 Run Systhesis하면 결과는 아래 사진과 같다.



3. Discussion

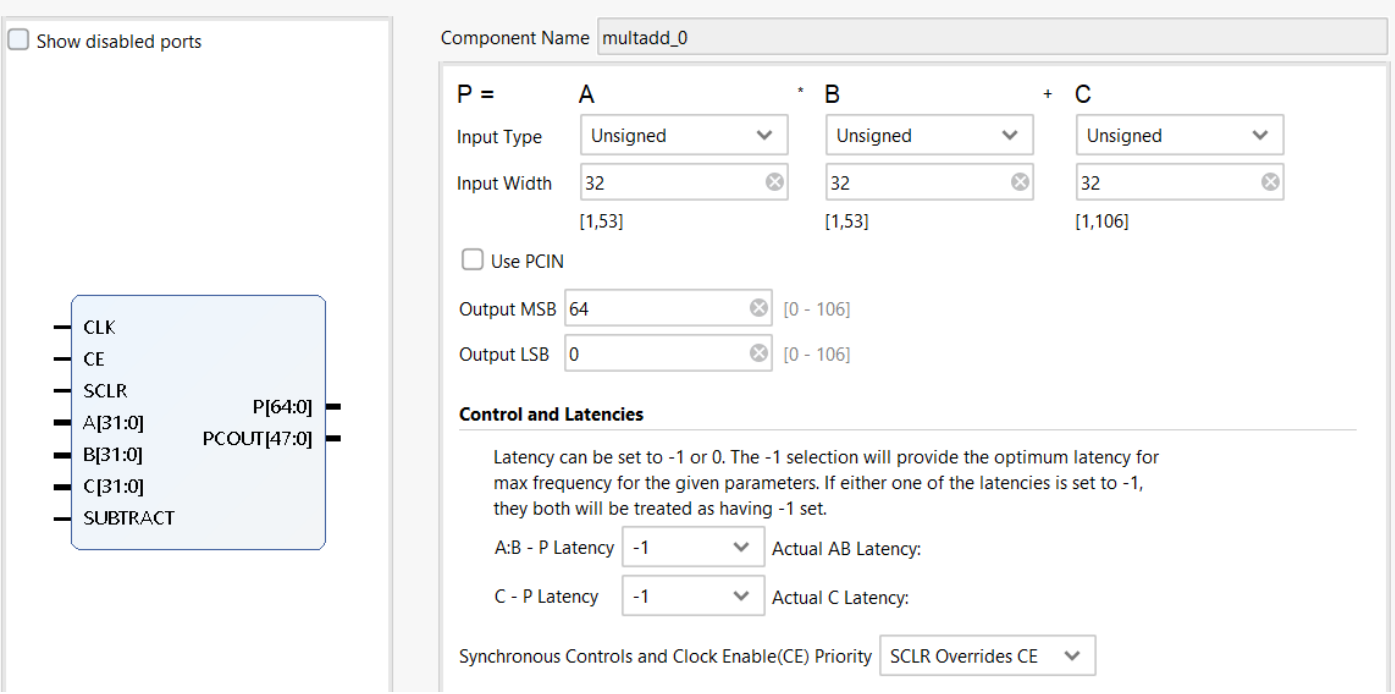
Simulation 결과 사진을 보면 input값이 들어오면 바로 결과가 나오지 않는다. 첫번째 input으로 들어온 값도 16사이클 뒤인 155ns에 output으로 계산 값이 나온다. 이 이유는 위에서 ip-catalog 설정에서 16사이클 Latency를 설정해 주었기 때문이다.

Simulation output res 값을 보면 inf.0이 있다. 이는 무한대를 의미한다. 숫자 표기 방법이 single-precision이므로 표현 할 수 있는값을 넘은 overflow 즉 지수부분이 1인 상태이다.

Integer Multiply Adder

1. IP catalog & Testbench

Integer Multiply Adder 모듈은 Multiply-adder 3.0 IP-catalog로 구현 하였다.

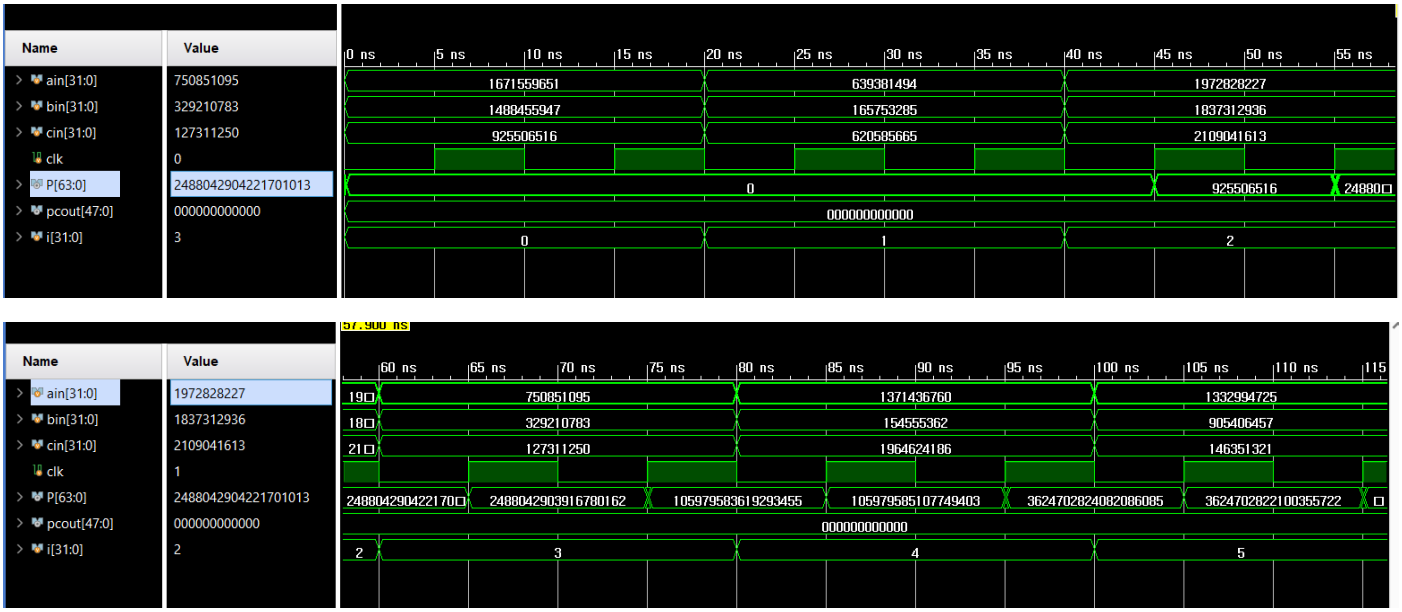


input A,B,C 모두 32bit unsigned integer이고 output P는 64bit이다.

latency가 존재 하도록 A:B-P 와 C-P에 모두 Latency를 설정 하였다.

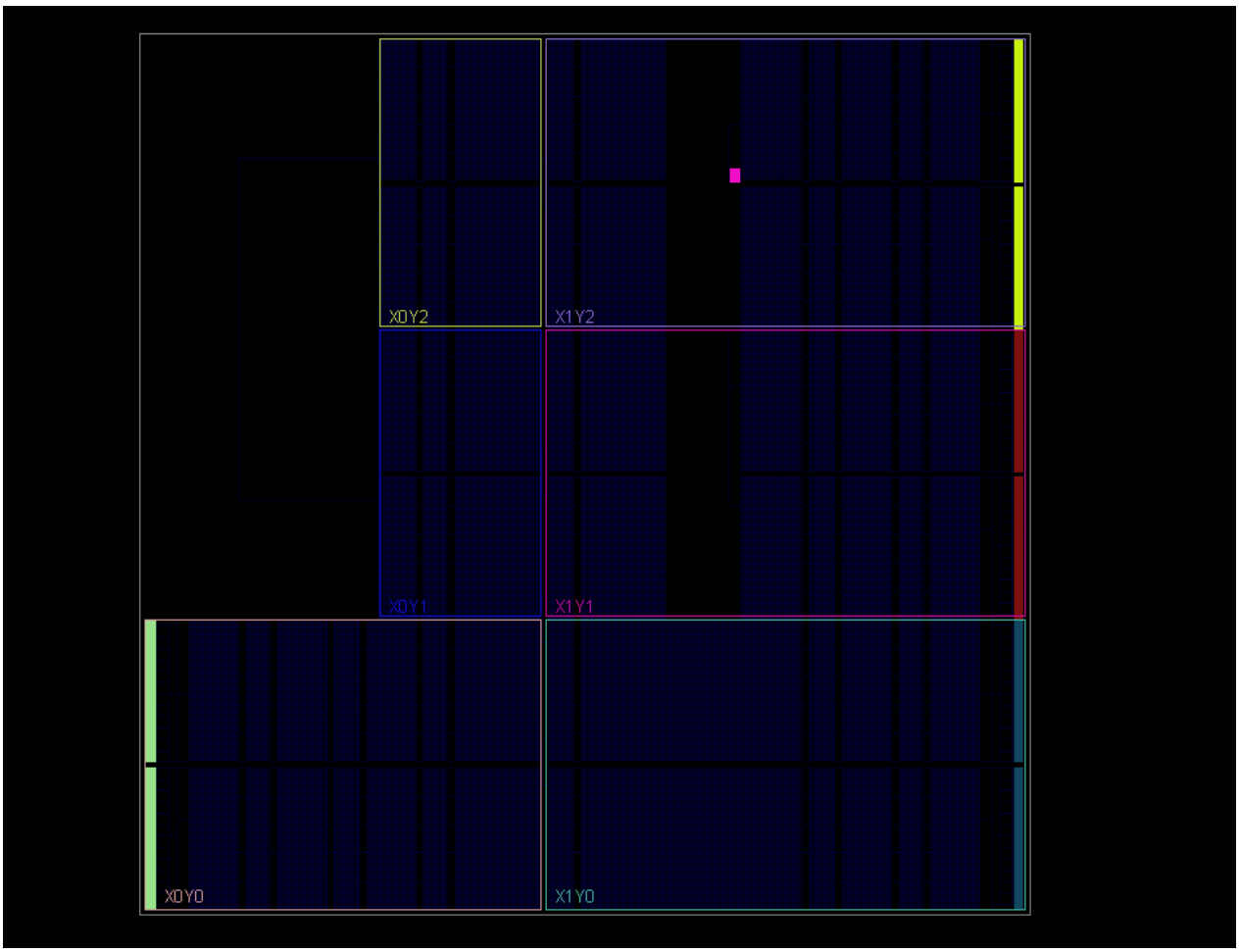
```
module tb_mul();

    reg [32-1:0] ain;
    reg [32-1:0] bin;
    reg [32-1:0] cin;
    reg clk;
    wire [64-1:0] P;
    wire [48-1:0] pcout;

    integer i;

    initial begin
        clk<=0;
        for(i=0; i<50; i=i+1)begin
          ain = $urandom%(2**31);
          bin = $urandom%(2**31);
          cin = $urandom%(2**31);
          #20;
        end
     end

     always #5 clk=~clk;

multadd_0 UUT(
    .CLK(clk),
    .SUBTRACT(1'b0),
    .SCLR(1'b0),
    .CE(1'b1),
    .A(ain),
    .B(bin),
    .C(cin),
    .P(P),
    .PCOUT(pcout)
);

endmodule
```

Testbench 코드는 floating-point multiply adder와 거의 동일한 구조로 만들었다. 이 testbech를 top 모듈로 설정한뒤 simulation을 돌리면 아래와 같다



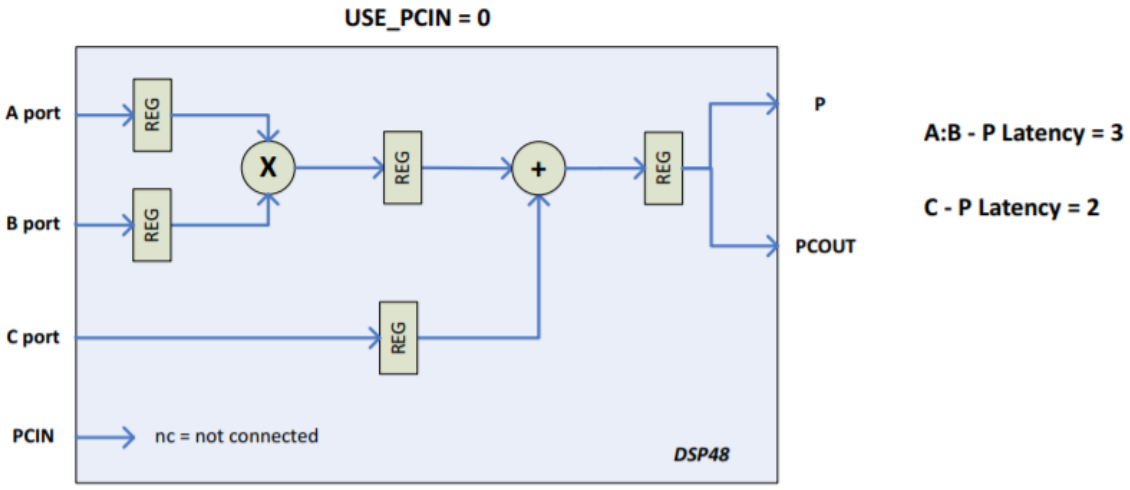
모든 변수의 radix는 unsigned 10진수로 설정 하였다. Output P의 값을 보면 input이 들어오자마자 출력이 안되는것과 한사이클을 주기로 반복적으로 올바른 output과 이상한 값이 돌아가면서 나오는 값을 알수 있다.

2. Systhesis



3.Discussion

simulation output값이 딜레이도 있고 주기적으로 올바른 결과값, 쓰레기값으로 반복적으로 나온다. 이 이유는 아래 그림을 보면



A:B-P Latency는 3인데 반면에 C-P 는 2이다. 이 둘에 Latency에 차이가 나기 때문에 먼저 그 전의 res 값에 C의 값이 더한 값이 res에 출력되고 한 사이클 뒤에 올바른 AxB+C 결과값이 res에 출력되는것이다.

Adder-Array

1.Verilog

adder-array 모듈은 verilog로 구현하였다. Verilog 코드는 아래와 같다.

```
module adder_array(cmd, Ain0, Ain1, Ain2, Ain3,
Bin0, Bin1, Bin2, Bin3,
Dout0, Dout1, Dout2, Dout3, overflow);
    input [2:0] cmd;
    input [31:0] Ain0;
    input [31:0] Ain1;
    input [31:0] Ain2;
    input [31:0] Ain3;
    input [31:0] Bin0;
    input [31:0] Bin1;
    input [31:0] Bin2;
    input [31:0] Bin3;
    output reg [31:0] Dout0;
    output reg [31:0] Dout1;
    output reg [31:0] Dout2;
    output reg [31:0] Dout3;
    output reg [3:0] overflow;

    wire [31:0] ain[3:0];
    wire [31:0] bin[3:0];
    wire [31:0] dout[3:0];
```

```
assign {ain[0], ain[1], ain[2], ain[3]} = {Ain0, Ain1, Ain2, Ain3};
assign {bin[0], bin[1], bin[2], bin[3]} = {Bin0, Bin1, Bin2, Bin3};


genvar i;

generate for(i=0; i< 4; i=i+1) begin: a_add

myadder a_a(ain[i], bin[i], dout[i], ov[i]);

end endgenerate
```

먼저 처음에는 모듈을 선언한뒤 input, output을 선언 해주었다. 모든 input,output은 unsigned integer로 설정하였다. 그리고 wire로 ain, bin, dout 배열을 선언해주고 이를 assign으로 ain[i] =Ain(i) 로 할당해 주었다. 이렇게 한 이유는 바로 다음에 나오는데, 바로 다음 generate for문을 이용하여 myadder 모듈로 더하기 과정을 해주었다. ain[i]로 할당해 주었기 때문에 따로 변수를 선언하지 않고 for문 i값에 따라 모듈 input,output으로 사용 할수 있게 하였다.

```
always@(*)begin
if(cmd == 3'd0)begin
    Dout0 <= dout[0];
    Dout1 <= 0;
    Dout2 <= 0;
    Dout3 <= 0;
    overflow[0] <= ov[0];
    overflow[1] <= 0;
    overflow[2] <= 0;
    overflow[3] <= 0;
end
else if(cmd == 3'd1)begin
    Dout0 <= 0;
    Dout1 <= dout[1];
    Dout2 <= 0;
    Dout3 <= 0;
    overflow[0] <= 0;
    overflow[1] <= ov[1];
    overflow[2] <= 0;
    overflow[3] <= 0;
end
else if(cmd == 3'd2)begin
    Dout0 <= 0;
    Dout1 <= 0;
    Dout2 <= dout[2];
    Dout3 <= 0;
    overflow[0] <= 0;
    overflow[1] <= 0;
```

```
81          overflow[3] <= 0;
82      end
83      else if(cmd ==3'd4)begin
84          Dout0 <= dout[0];
85          Dout1 <= dout[1];
86          Dout2 <= dout[2];
87          Dout3 <= dout[3];
88          overflow[0] <= ov[0];
89          overflow[1] <= ov[1];
90          overflow[2] <= ov[2];
91          overflow[3] <= ov[3];
92
93      end
94      end
95
96  endmodule
```

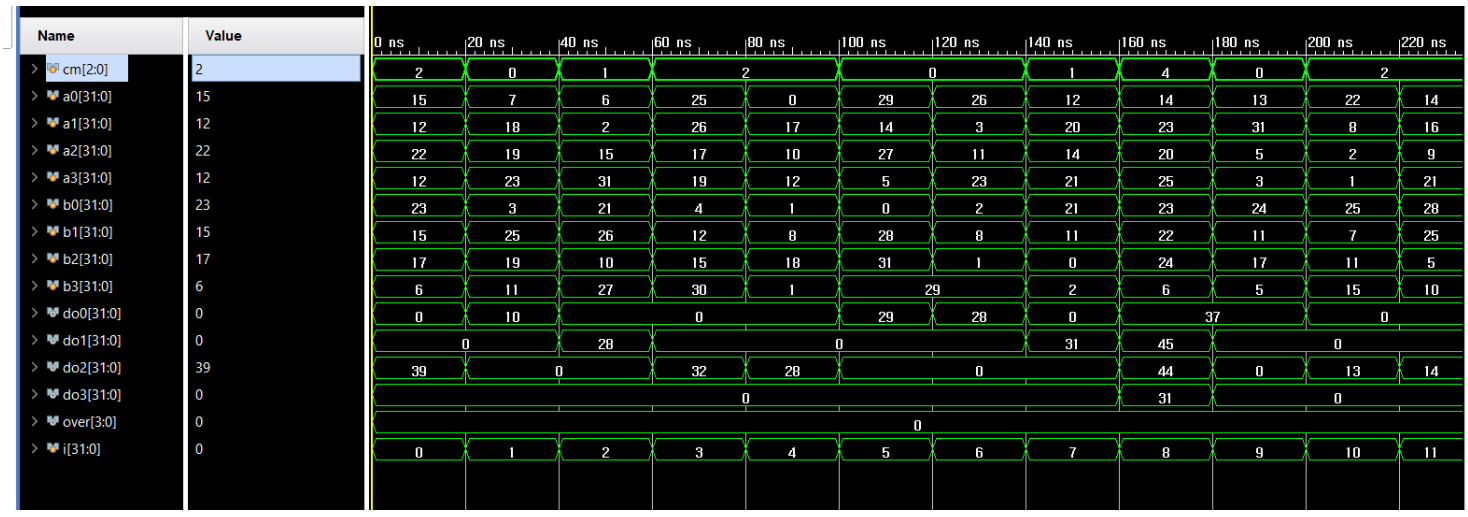
그리고 이제 input cmd 값에 따라 결과 출력값이 정해질수 있도록 always문을 사용하여 여러 if문 조건에 cmd가 오게 하였다. cmd가 0이면 Dout0과 overflow[0]만 출력되고 나머지는 0이 되도록 하고 cmd가 1,2,3일 때 마찬가지로 해당하는 output만 출력되도록하고 4일때는 모든 output이 출력되도록 하였다. 사실 always문 말고 assign문으로도 코드를 짤수 있었지만 always문을 사용하고 싶어 이렇게 코드를 만들었다.

2. Testbench & Simulation

```
module tb_array();

    reg [2:0] cm;
    reg [31:0] a0;
    reg [31:0] a1;
    reg [31:0] a2;
    reg [31:0] a3;
    reg [31:0] b0;
    reg [31:0] b1;
    reg [31:0] b2;
    reg [31:0] b3;
    wire [31:0] do0;
    wire [31:0] do1;
    wire [31:0] do2;
    wire [31:0] do3;
    wire [3:0] over;

    integer i;
```

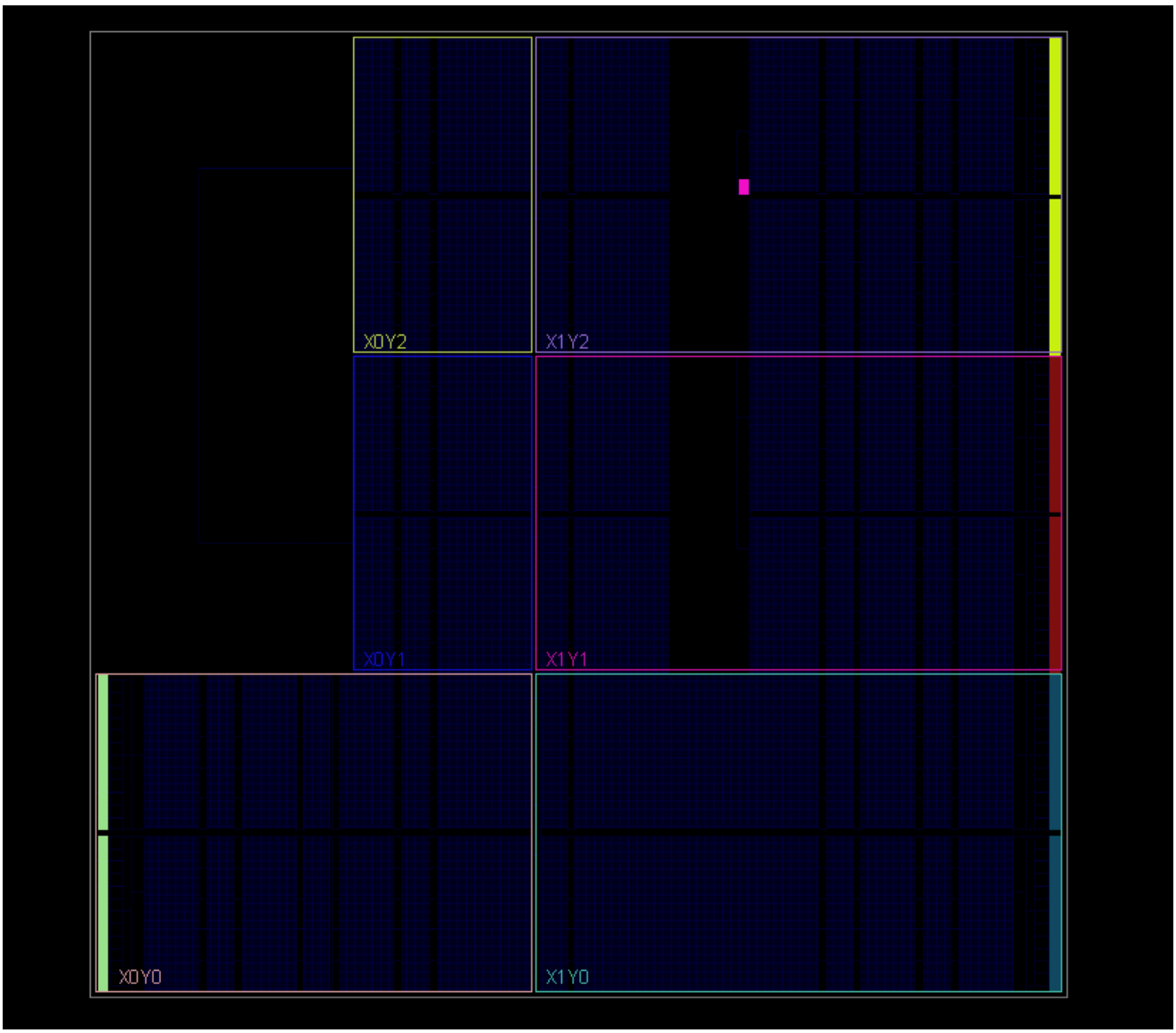
```
initial begin
    for(i=0; i<64; i=i+1)begin
        a0 <= $urandom%(2**5);
        a1 <= $urandom%(2**5);
        a2 <= $urandom%(2**5);
        a3 <= $urandom%(2**5);
        b0 <= $urandom%(2**5);
        b1 <= $urandom%(2**5);
        b2 <= $urandom%(2**5);
        b3 <= $urandom%(2**5);
        cm <= $urandom%(5);

        #20;
    end

end

adder_array UUT(
    .cmd(cm),
    .Ain0(a0),
    .Ain1(a1),
    .Ain2(a2),
    .Ain3(a3),
    .Bin0(b0),
    .Bin1(b1),
    .Bin2(b2),
    .Bin3(b3),
```

앞의 두 모듈들과 testbench문은 크게 다를게 없다. 처음에 모듈 및 input, output 들을 선언하고 for문을 돌면서 input값에 랜덤한 값이 입력되도록 urandom 함수를 사용하였다.



Testbench의 simulation wave이다. 이번에는 clk 신호를 사용하지 않고 Latency 또한 설정하지않았으므로 input이 들어오면 바로 그에 해당하는 output이 출력된다. 위 사진을 보면 cm(cmd)값에 따라 output do 1~3 이 0이 출력되거나 덧셈 값이 출력된다.

3. Systhesis



Array-adder 모듈을 Systhesis 결과값은 위와 같다.

4. Discussion

verilog에서 cmd값에 따라 output이 정해지도록 하기 위해 나는 always 및 if문을 사용하였다. 하지만 이는 assign문으로도 구현 가능한다 예를들어 dout0 하나만 코드를 작성하면

```
assign Dout0= (cmd==0) ? dout[0] : 0
```

```
assign overflow[0] = (cmd==0) ? ov[0]: 0
```

이렇게 구현할 수 있다.

이 모듈은 clock 신호를 사용하지 않은 비동기 방식으로 구현하였는데 만약 clock 신호를 사용하면 다른 모듈들과 마찬가지로 latency가 발생하였을 것이다.