Practice 4

How to use IP catalog & Synthesize

2017-19428 서준원

1. Introduction

이번 실습에서는 IP Catalog와 Synthesis를 사용하는 방법에 대해 실습을 진행하였다. 또한 Verilog 문법 연습의 일부로, 수업시간에 배운 generate 문을 직접 사용해보는 실습을 진행하였다.

IP (Intellectual Property) Catalog는 많이 사용하는 로직들을 미리 구현해 놓아 블랙박스 형태로 사용할 수 있는 모듈이다. Xillinx에서는 FPGA에서 다양한 IP들을 사용할 수 있으며, 구성한 회로를 Synthesis 해볼 수 있다.

이번 실습에서는 IP Catalog 중 Floating Point Multipy-Adder와 Integer Multiply-Adder를 사용해보고 시뮬레이션을 해보았다. 또한 지난 실습에서 만든 Unsigned Integer Adder에 대해 for-generate 문을 사용하여 동시에 여러 개의 모듈을 만드는 방법을 실습해보았다. 이 Array-Integer를 시뮬레이션하는 스크립트를 작성하여 결과를 확인해보았다.

2. Implementation

(1) 32bit floating point Multiply-Adder

```
module tb();

    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<32; 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 fp(
        .aclk(clk),
        .aresetn(~rst),
        .s_axis_a_tdata(ain),
        .s_axis_b_tdata(bin),
        .s_axis_c_tdata(cin),
        .s_axis_a_tvalid(1'b1),
        .s_axis_b_tvalid(1'b1),
        .s_axis_c_tvalid(1'b1),
        .m_axis_result_tvalid(dvalid),
        .m_axis_result_tdata(res)
        );
endmodule
```

위는 32bit floating point multiply adder를 시뮬레이션 하기 위한 코드이다. 해당 IP에는 다양한 Input과 Output이 정의가 되어 있어 내가 만든 레지스터를 적당하게 채워주기만 하면 된다. 총 32개의 난수에 대해 시뮬레이션을 진행하였다.

Input은 s_axis_a_tdata, s_axis_b_tdata, s_axis_c_tdata에 넣어주었고, m_axis_result_tdata에 a*b + c의 결과가 출력된다. 만약 Invalid 일 경우에는 dvalid 비트가 0으로 세팅된다.

(2) 32bit integer Multiply-Adder

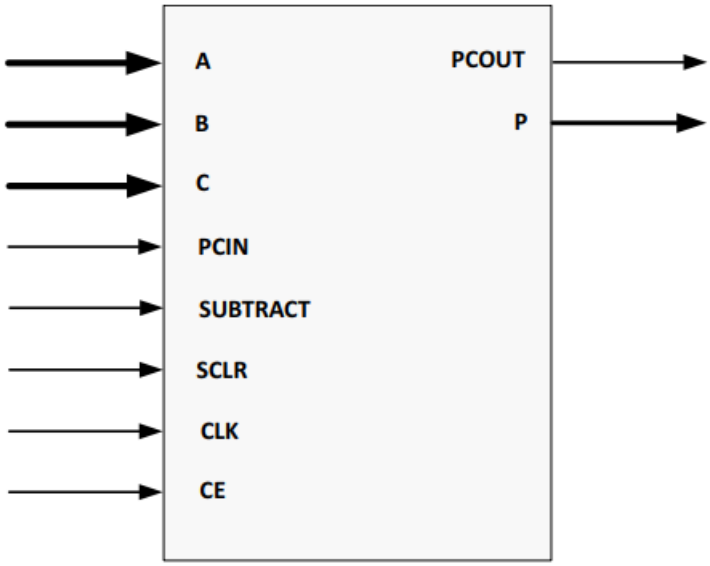


Figure 2-1: **Core Symbol**

Table 2-1: **Core Signal Pinout**

Name	Direction	Description
A[N:0]	Input	A Input bus (multiplier operand 1)
B[M:0]	Input	B Input bus (multiplier operand 2)
C[L:0][a]	Input	C Input bus (operand 1 of add/sub operation)
PCIN[b]	Input	Cascade Input
SUBTRACT	Input	Controls Add/Subtract operation (High = subtraction, Low = addition)
CE	Input	Clock Enable (active-High)
CLK	Input	Clock (rising edge)
SCLR	Input	Synchronous Clear (active-High)
PCOUT[(2)]	Output	Cascade Output
P[Q:0]	Output	Output bus

위 사진은 공식 다큐멘테이션에서 찾은 Integer multipy-adder의 그림이다. A와 B, 그리고 C에 임의의 길이의 정수를 넣어 주면 A*B + C의 결과를 얻을 수 있다. SCLR은 Clear를 위한 비트여서 0으로 항상 세팅했으며, 덧셈을 테스트해보고 싶었기 때문에 Subtract 비트는 0으로 세팅했다. 만약 이를 1로 세팅하면 A*B – C의 결과가 출력된다고 한다.

```
`timescale 1ns / 1ps


module tb_integer();
    reg[32-1:0] ain;
    reg[32-1:0] bin;
    reg[32-1:0] cin;
    reg clk;
    wire [64-1:0] res;
```

```
    wire [48-1:0] pcout;

    integer i;
    initial begin
        clk <=0;
        for(i=0; i<32; i=i+1) begin
            ain = $urandom%(2**31);
            bin = $urandom%(2**31);
            cin = $urandom%(2**31);
            #20;
        end
    end

    always #5 clk = ~clk;

    xbip_multadd_0 ma(
        .A(ain),
        .B(bin),
        .C(cin),
        .CLK(clk),
        .CE(1'b1),
        .SCLR(1'b0),
        .SUBTRACT(1'b0),
        .P(res),
        .PCOUT(pcout)
     );
endmodule
```

위는 Integer-Multiply-Adder IP를 테스트하기 위한 테스트 코드이다. 다른 부분은 모두 동일하고, 인풋과 아웃풋을 적절한 와이어에 연결을 해 주었다. 또한 다른 세팅을 위한 비트도 1 또는 0으로 적절하게 세팅을 해 주었다.

(3) Adder-array

```
module ADDER_ARRAY( cmd, ain0, ain1, ain2, ain3, bin0, bin1, bin2, bin3, dout0
, dout1, dout2, dout3, overflow);
       Input 과 Output 선언
    input [2:0] cmd;
    input [31:0] ain0, ain1, ain2, ain3;
    input [31:0] bin0, bin1, bin2, bin3;
    output [31:0] dout0, dout1, dout2, dout3;
    output [3:0] overflow;

    genvar i;    for-generate 를 위한 변수, 그리고 for-generate 문에서 인덱스를
통해 변수를 사용하기 위한 Memory 선언
```

```
    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};

// generate for 문의 사용. Adder 모듈을 각각 불려준다.
    generate for(i=0; i<4; i=i+1) begin:bitnum
        ADDER #(32) my_adder(
            .ain(ain[i]),
            .bin(bin[i]),
            .dout(dout[i]),
            .overflow(overflow[i])
           );
    end endgenerate
    reg [31:0] tmp [3:0];
   always @(*) begin
       case(cmd[2:0]) // cmd 값에 따라 출력을 다르게 하기 위해서 case 문을
사용하였다. 또한 조건문을 사용하기 위해 always 블록을 사용했고, 그 안에는 wire 가
들어갈 수 없어 임시로 레지스터 (tmp)를 선언하였다.
            3'b000 : {tmp[0], tmp[1], tmp[2], tmp[3]} = {dout[0], 32'b0, 32'b0
, 32'b0};
            3'b001 : {tmp[0], tmp[1], tmp[2], tmp[3]} =  {32'b0, dout[1], 32'b
0, 32'b0};
            3'b010 : {tmp[0], tmp[1], tmp[2], tmp[3]} =  {32'b0, 32'b0, dout[2
], 32'b0};
            3'b011 : {tmp[0], tmp[1], tmp[2], tmp[3]} =  {32'b0, 32'b0, 32'b0,
 dout[3]};
            3'b100 : {tmp[0], tmp[1], tmp[2], tmp[3]} =  {dout[0], dout[1], do
ut[2], dout[3]};
        endcase

    end
// 레지스터의 값을 out 에 넣어준다.
    assign {dout0, dout1, dout2, dout3} = {tmp[0], tmp[1], tmp[2], tmp[3]};
endmodule
```

위 코드는 Adder-Array를 구현한 모듈이다. 코드가 길기 때문에, 주석을 통해 설명을 대체하였다. 간단하게 요약하자면, for-generate를 사용하여 Array Size인 4개의 Adder 모듈을 사용하였고, input인 cmd의 값에 따라 결과를 다르게 하여 출력하였다.

(4) Test bench for Adder-array

```
module test_adderarray();
    reg [2:0] cmd;
```

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

    reg clk;

    always #5 clk = ~clk;
    integer i, j, k;
    initial begin
        clk <=0;
        #30;

        for(i=0; i<=4; i=i+1) begin
            cmd = i;
            for(j=0; j<=8; j=j+1) begin
                for(k=0; k<4; k=k+1) begin
                    ain[k] = $urandom%(2**32-1);
                    bin[k] = $urandom%(2**32-1);
                end

                #10;
            end
        end
    end

    ADDER_ARRAY test(
    .cmd(cmd),
    .ain0(ain[0]),
    .ain1(ain[1]),
    .ain2(ain[2]),
    .ain3(ain[3]),
    .bin0(bin[0]),
    .bin1(bin[1]),
    .bin2(bin[2]),
    .bin3(bin[3]),
    .dout0(dout[0]),
    .dout1(dout[1]),
    .dout2(dout[2]),
    .dout3(dout[3]),
    .overflow(overflow)
    );
endmodule
```

Adder-Array 모듈을 테스트하기 위한 테스트 벤치이다. 3중 for문을 사용하여 cmd, ain, bin 값을 다르게 설정하였고, overflow 테스트를 위해 ain과 bin은 2**32-1의 랜덤을 사용하였다. 한 개의

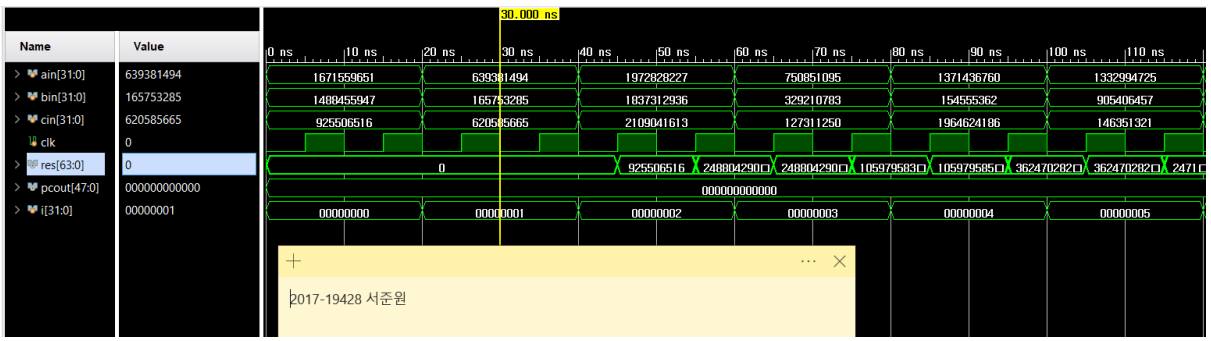
cmd 값에 대해 총 8번의 다른 값을 테스트하였다. Ain과 bin Array 값이 바뀔 때마다 클럭을 바꾸어주었다.

3. Results

(1) 32bit floating point Multiply-Adder

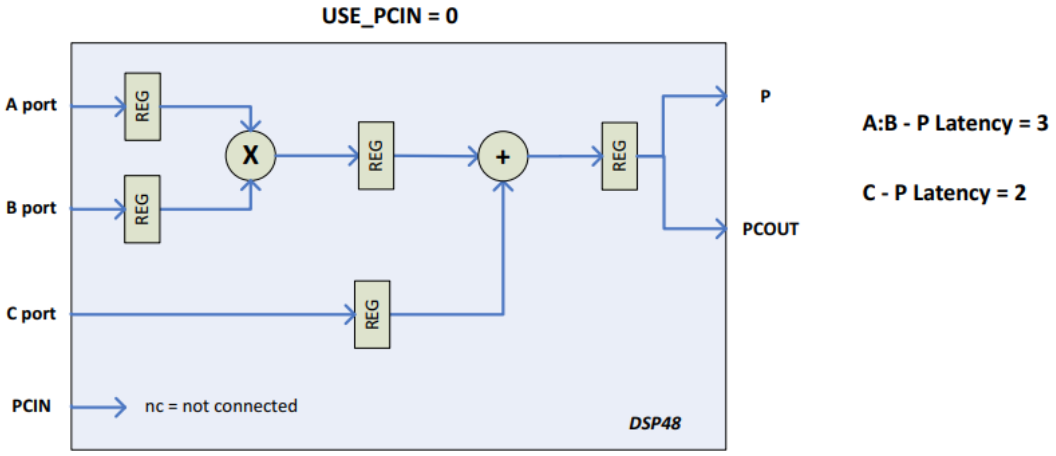
아래 사진은 32bit floating point Multiply-Adder 의 waveform 결과이다. 결과가 원하는 데로 잘 나온 것을 확인할 수있다. Dvalid가 0이 아닐 때의 결과만 유의미하게 체크할 수 있다.

(2) 32bit integer Multiply-Adder

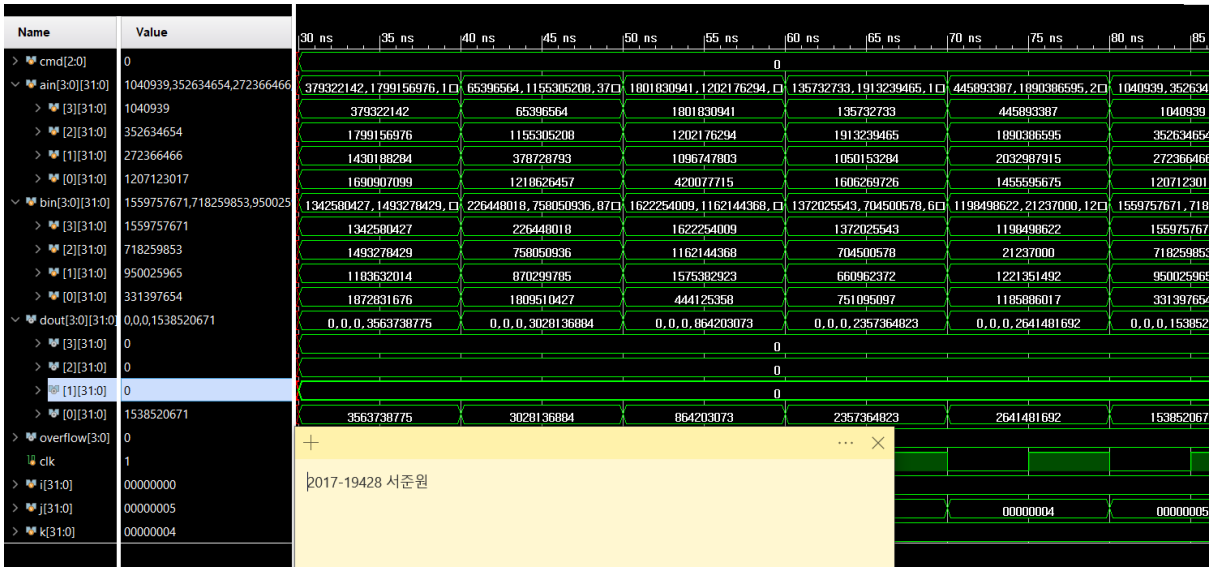


위 사진은 32bit integer Multiply-Adder IP의 실험 결과이다. 결과가 약간 이상하게 보일 수 있다. 정확히 표현하자면, 결과가 밀려서 나오고, 한 사이클에서 결과가 변하는 것을 확인할 수 있다. 이는 IP의 구현 방법과 상관이 있다. 다큐멘테이션을 참조하면 Add와 Multiply 사이에 Latency가 다르다. 이에 따라 결과가 input이 변하는 cycle과 차이가 있다고 해석할 수 있다.

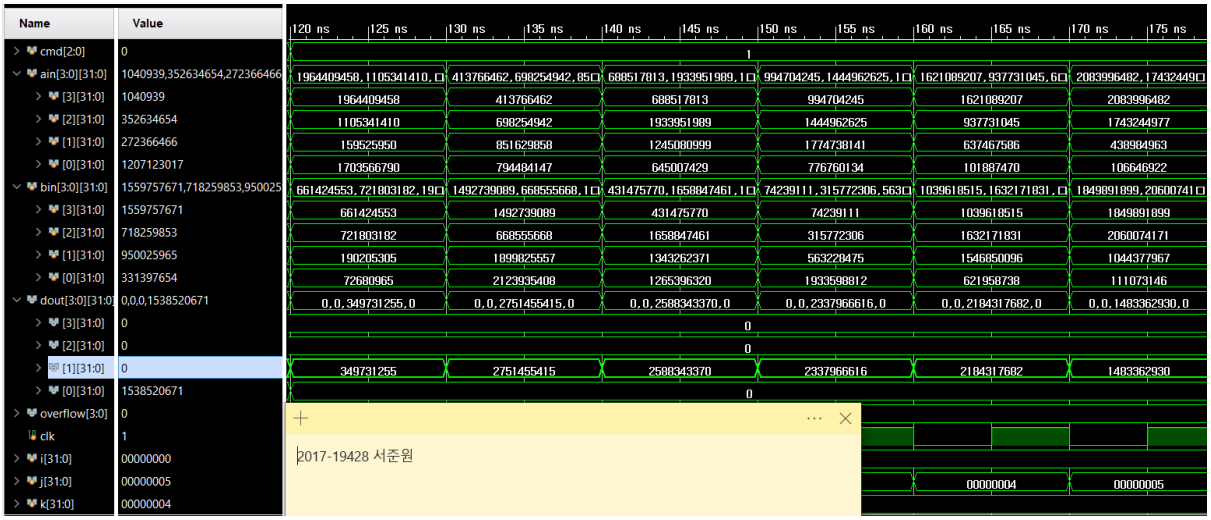
아래 사진은 공식 Documentation의 32bit integer Multiply-Adder IP 설계도이다. Latency에 대한 설명이 주어져있다.



(3) Adder-array



위 사진은 cmd가 0일 때의 wave-form 결과이다. Ain과 bin, 그리고 dout에서 0번째 member만 계산이 되어있고, 나머지는 0으로 세팅된 것을 확인할 수 있다.



위 사진은 cmd가 1일 때의 Waveform 결과이다. 마찬가지로 인풋과 output array의 1번째 멤버만이 계산되어 있다. 마찬가지로 아래 사진들은 cmd가 각각 2, 3일 때의 결과이다.

위 사진은 ip가 4일 때의 사진이다. 이 때는 위의 결과들과는 다르게 output array의 결과들이 각각 저장되어있는 것을 확인할 수 있다. 이는 코드에서 case 문을 사용하여 구현이 되었다. 또한 모든 결과에 대해 각각의 array마다의 overflow가 해당되는 자리에 세팅되어 있는 것을 확인할 수 있다.