4190.301A Hardware System Design

Spring 2020

Hardware System Design Lab4 Report

Kim Bochang

2014-16757

1. <lab4> Introduce

Lab 4의 목적은 xilinx가 지원하는 IP catalog의 모듈(32bit floating point, integer multiply-adder)을 불러와 사용하는 법을 익히고, Lab 3에서 구현한 adder를 이용해 adder array를 구현하는 것이다.

2. Implementation

2.1 floating point multiply-adder

베릴로그의 IP catalog를 이용하여 32bit floating point multiply-adder를 생성하는 과정은 lab4 pdf와 똑같이 진행 하였고, 테스트 벤치 역시 똑같이 사용하였으므로, 따로 서술하지 않겠다.

굳이 말하자면, IP catalog의 Floating-point를 이용하여 fused_multiply 기능을 선택하고, nonblocking 하도록 기능을 수정하고, arestn을 active low로 설정하고 maximum latency를 가지도록 모듈을 만들었다.

위에서 만든 모듈은 다음과 같은 포트를 가진다.

input

aclk : clk를 입력으로 받음. posedge에서 작동.

areset : 1을 받으면 내부 레지스터를 리셋함.

s_axis_x_tvaild : 각 x가 valid한지를 입력해주는 포트. (x: a,b,c) valid하면 1이 입력된다.

s_axis_x_tdata : 각 x의 입력값. single precision floating point. (x: a,b,c)

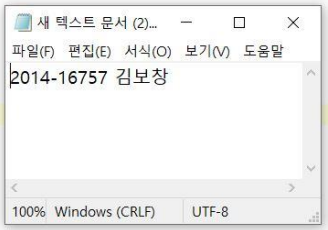
output

m_axix_result_tvalid : 출력이 valid한지를 나타내주는 값. valid하면 1이 출력된다.

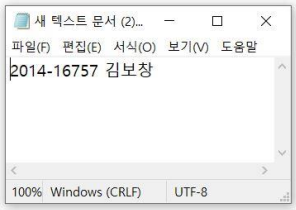
m_axix_result_tdata : a*b + c의 값

이를 test하는 test-bench 코드는 다음과 같다.

```
module tb_lab4();

    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;

    //for test
    integer i;
    //random test vector generation
    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);
```




```
            cin = $urandom%(2**31);
            #20;
        end
    end

    always #5 clk = ~clk;

    floating_point_MAC 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
```



<lab4\lab4.srcs\sources_1\new\tb.lab4.v>

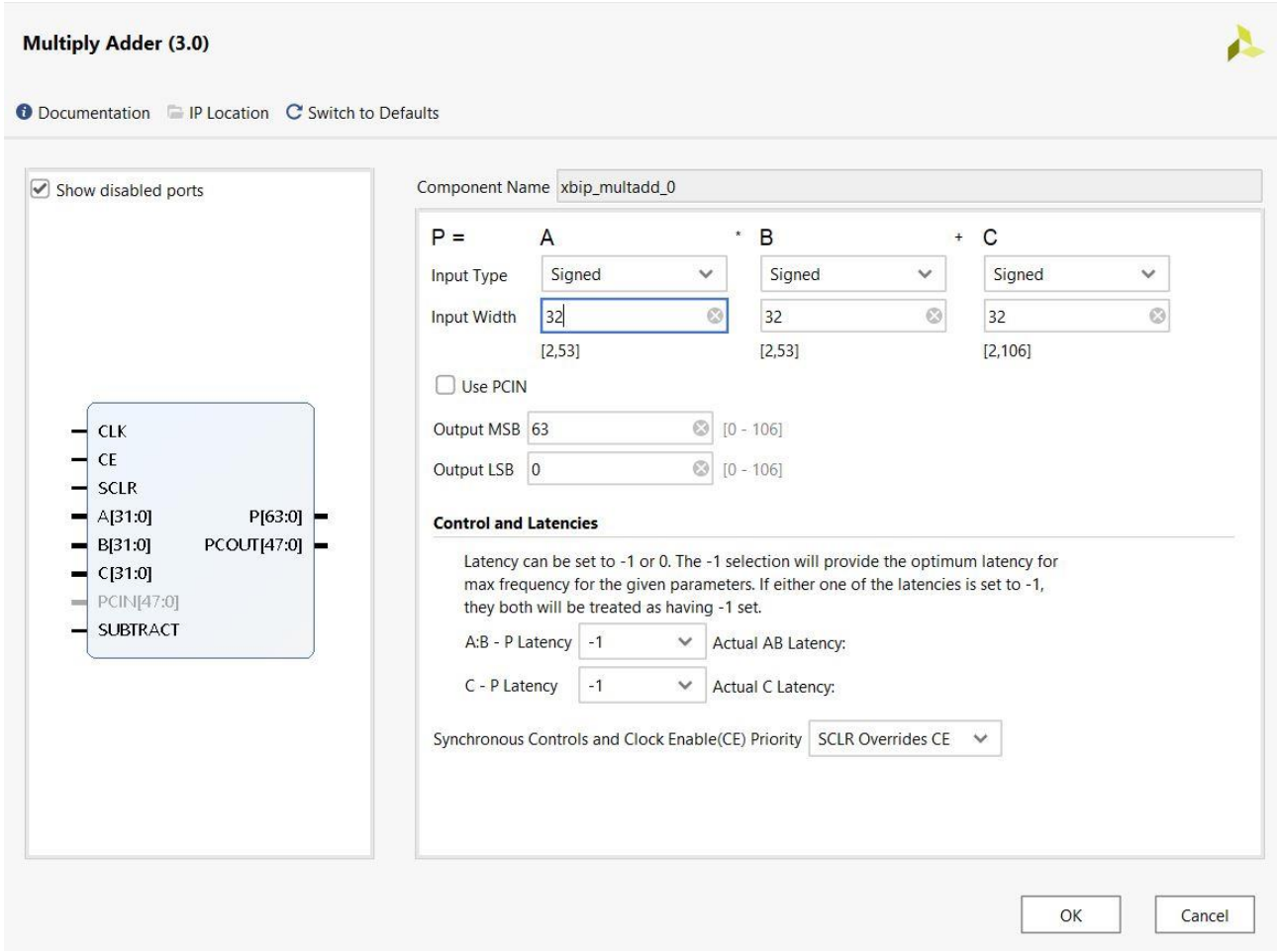
32개의 random vector를 만들어서 주기마다 위에서 만든 모듈에 집어넣고,

클럭을 계속해서 생성해준다.

2.2 integer multiply-adder

32bit integer multiply-adder를 구현하기 위해서는, Vivado의 IP Catalog에서 Multiply Adder를 이용하면 된다.

Multiply Adder의 설정은 다음과 같이 진행하였다.



32bit signed integer의 곱과 합을 계산해야 하므로,

nbit signed integer의 표현범위는 -2^(n-1) ~ 2^(n-1) – 1 이기 때문에,

두 32bit integer의 곱은 대략 -2^(62) ~ 2^(62)정도의 범위를 가지게 되고, 여기에 32bit integer를 더하거나 빼면 약간 범위가 더 커지므로, 64bit integer를 사용해야 모든 결과를 커버할 수 있다.

따라서 A,B,C의 input bit로 32비트를 줬기 때문에,

output의 bit로 64bit를 사용하였다. (MSB = 63, LSB = 0)

IP catalog에서의 Multiply Adder의 각 포트의 역할은 IP documentation – View Datasheet를 통해 알 수 있다.

여기에서 포트들이 다음 역할을 함을 알 수 있었다.

input

CLK : clock. posedge에서 작동.

CE : clock enable.

SCLR : clear. 1을 주면 내부 레지스터를 0으로 초기화한다.

A,B,C : input vector

SUBTRACT : C를 A*B의 결과에 더할지, 뺄지를 결정한다. 0이면 더하고, 1이면 뺀다.

P : output vector

PCOUT : 지금까지 계산값을 더한것. PCIN을 disable했기때문에 사용하지 않는다.

위에서 만든 adder를 test하기 위한 코드는 다음과 같다.

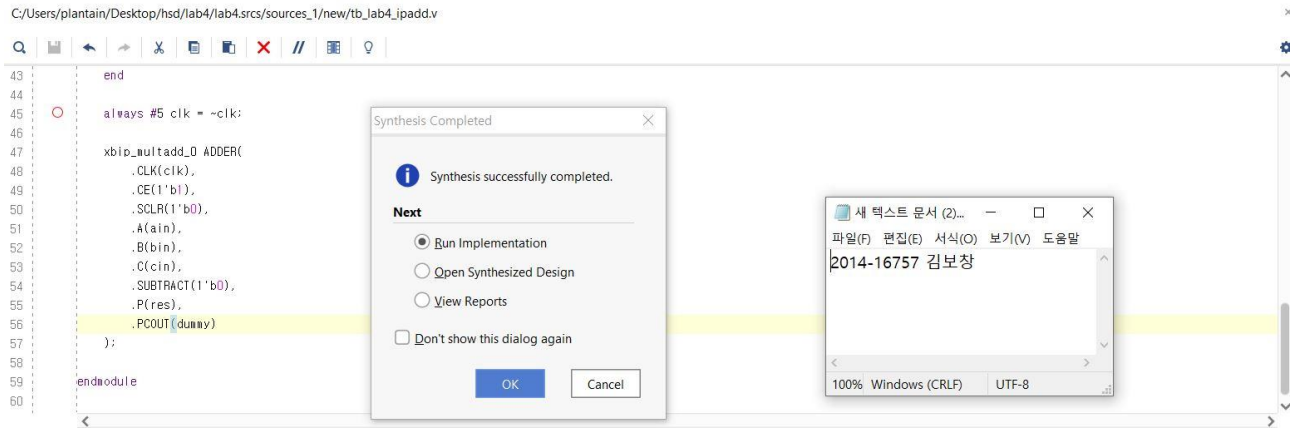
```
module tb_lab4_ipadd();

    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] dummy;

    //for test
    integer i;
    //random test vector generation
    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 ADDER(
        .CLK(clk),
        .CE(1'b1),
        .SCLR(1'b0),
        .A(ain),
        .B(bin),
        .C(cin),
        .SUBTRACT(1'b0),
        .P(res),
        .PCOUT(dummy)
    );

endmodule
```

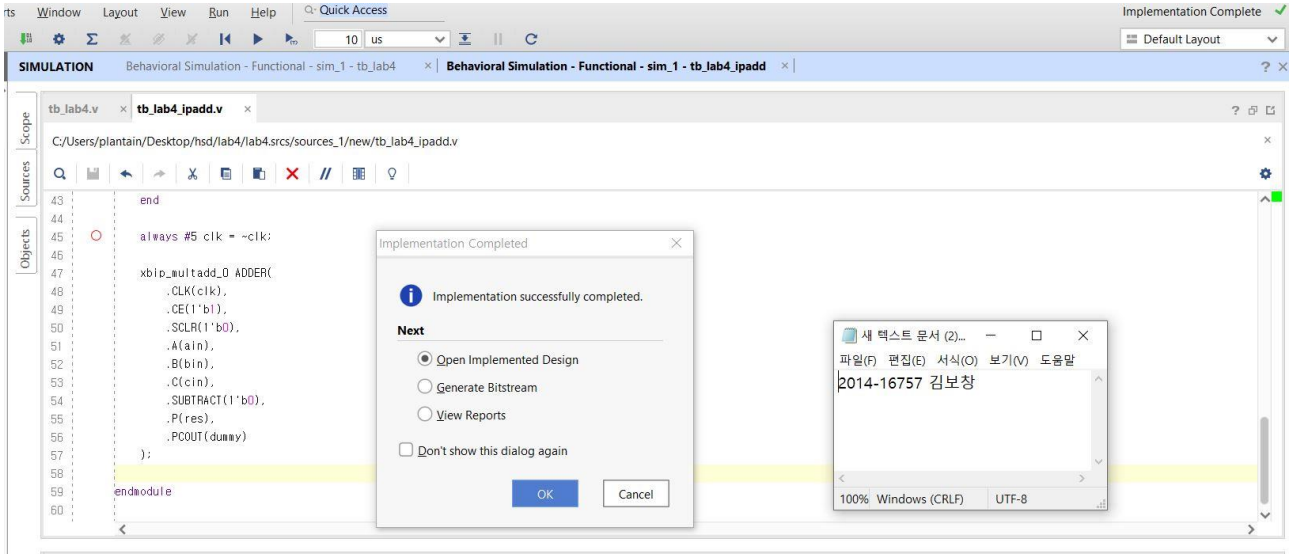
<lab4\lab4.srcs\sources_1\new\tb.lab4.v>

32개의 random vector를 만들어서 주기마다 위에서 만든 모듈에 집어넣고,

클럭을 계속해서 생성해준다.

여기까지 한 후, synthesis와 implement를 진행하였고, 다음과 같이 정상적으로 진행된 것을 확인할 수 있었다.


C:/Users/plantain/Desktop/hsd/lab4/lab4.srcs/sources_1/new/tb_lab4_ipadd.v
end
always #5 clk = ~clk;
xbip_multadd_0 ADDER(
.CLK(clk),
.CE(1'b1),
.SCLR(1'b0),
.A(ain),
.B(bin),
.C(cin),
.SUBTRACT(1'b0),
.P(res),
.PCOUT(dummy)
);
endmodule
Synthesis Completed
Synthesis successfully completed.
Next
Run Implementation
Open Synthesized Design
View Reports
Don't show this dialog again
OK
Cancel
새 텍스트 문서 (2)...
파일(F) 편집(E) 서식(O) 보기(V) 도움말
2014-16757 김보창
100% Windows (CRLF) UTF-8


Window Layout View Run Help
Quick Access
Implementation Complete
Default Layout
SIMULATION
Behavioral Simulation - Functional - sim_1 - tb_lab4
Behavioral Simulation - Functional - sim_1 - tb_lab4_ipadd
tb_lab4.v
tb_lab4_ipadd.v
Scope
Sources
Objects
C:/Users/plantain/Desktop/hsd/lab4/lab4.srcs/sources_1/new/tb_lab4_ipadd.v
Implementation Completed
Implementation successfully completed.
Next
Open Implemented Design
Generate Bitstream
View Reports
Don't show this dialog again
OK
Cancel
새 텍스트 문서 (2)...
2014-16757 김보창
100% Windows (CRLF) UTF-8

2.3 adder-array

마지막으로, adder-array를 구현하면 lab이 마무리된다.

이 adder-array를 만들 때는 Lab3에서 구현한 my adder를 사용하였다.

adder array의 각 포트의 역할은 다음과 같다.

input

cmd : 3비트 입력. cmd값에따라 어떤 값을 출력할지가 달라진다.

ainx : 32비트 입력. 내부에서 adder의 input으로 들어감. (x = 0,1,2,3)

binx : 32비트 입력. 내부에서 adder의 input으로 들어감. (x = 0,1,2,3)

output

doutx : 32 bit 출력. cmd의 값에 따라 a_inx + b_inx의 결과를 가지거나, 0의 결과를 가진다.

overflow : 4bit 출력. cmd의 값에 따라 각 adder에서 overflow가 일어났는지 여부를 출력하거나, 0을 출력한다.

작동은,

cmd가 0~3일때는 dout0~3과 overflow의 0~3번째 비트에 ain0~3,bin0~3의 덧셈결과와 오버플로우 여부를 출력,

그 이외의 값에 대해서는 0을 출력하도록 하였고,

cmd가 4일때는 모든 dout과 overflow에 모든 ain과 bin의 덧셈결과와 오버플로우 여부를 출력하도록,

cmd가 5,6,7일때는 모든 dout과 overflow에 0이 출력되도록 하였다.

베릴로그 코드는 다음과 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, ain1, ain2, ain3;
input [31:0] bin0, bin1, bin2, bin3;
output [31:0] dout0, dout1, dout2, dout3;
output [3:0] overflow;


wire [31:0] ain [3:0];
wire [31:0] bin [3:0];
wire [31:0] dout [3:0];
wire temp_overflow[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:adder
    my_add #(32) MY_ADDER(
       .ain(ain[i]),
       .bin(bin[i]),
       .dout(dout[i]),
       .overflow(temp_overflow[i])
    );
end endgenerate

assign dout0 = (cmd == 3'b000) ? dout[0] :
               (cmd == 3'b100) ? dout[0] :
               {(32){1'b0}};

assign dout1 = (cmd == 3'b001) ? dout[1] :
               (cmd == 3'b100) ? dout[1] :
               {(32){1'b0}};

assign dout2 = (cmd == 3'b010) ? dout[2] :
               (cmd == 3'b100) ? dout[2] :
               {(32){1'b0}};
```

```
assign dout3 = (cmd == 3'b011) ? dout[3] :
              (cmd == 3'b100) ? dout[3] :
              {(32){1'b0}};

assign overflow = (cmd == 3'b000) ? {1'b0, 1'b0, 1'b0, temp_overflow[0]} :
                  (cmd == 3'b001) ? {1'b0, 1'b0,  temp_overflow[1], 1'b0} :
                  (cmd == 3'b010) ? {1'b0, temp_overflow[2], 1'b0, 1'b0} :
                  (cmd == 3'b011) ? {temp_overflow[3], 1'b0, 1'b0, 1'b0} :
                  (cmd == 3'b100) ? {temp_overflow[3], temp_overflow[2], temp_overflow[1], temp_overflow[0]} :
                  {(4){1'b0}};

endmodule
```

<lab4\lab4.srcs\sources_1\new\adder_array.v>

generate for문을 이용해, my_adder를 4개 생성하였고, 생성한 adder에 각각 알맞는 input을 넣어주어 계산이 진행되도록 하였다.

그 후, cmd의 값에 따라 계산된 결과값과 overflow 여부를 그대로 리턴하거나, 0으로 리턴하도록 하였다.

여기서 사용된 my_add는 lab3에서 만들었던 것으로, 다음과 같은 코드이다.

```
module my_add #(parameter BITWIDTH = 32)
(
input [BITWIDTH-1:0] ain,
input [BITWIDTH-1:0] bin,
output [BITWIDTH-1:0] dout,
output overflow
);
/* IMPLEMENT HERE! */

assign {overflow, dout} = ain + bin;

endmodule
```

<lab4\lab4.srcs\sources_1\imports\new\my_add.v>

그리고, 이를 테스트하기 위한 테스트 벤치 코드를 다음과 같이 작성하였다.

```
module tb_adder_array();

    reg [2:0] cmd;
    reg [32-1:0] ain[3:0];
    reg [32-1:0] bin[3:0];

    wire [32-1:0] dout0, dout1, dout2, dout3;
    wire [3:0] overflow;

    //for test
    integer i,j,k;
    //random test vector generation
    initial begin
        for(i=0; i<8; i=i+1) begin
            cmd = i;
            for(j=0; j<4; j=j+1) begin
                for(k=0; k<4; k=k+1) begin
                    ain[k] = $urandom%(2**31);
                    bin[k] = $urandom%(2**31);
                end
                #20;
            end  // for each cmd, make random input
        end

        #100;
        cmd = 4;
        ain[0] = {(32){1'b1}};
        ain[1] = {(32){1'b0}};
        ain[2] = {(32){1'b1}};
        ain[3] = {(32){1'b0}};
        bin[0] = {(32){1'b1}};
        bin[1] = {(32){1'b1}};
        bin[2] = {(32){1'b1}};
        bin[3] = {(32){1'b1}};

    end
```

```
    adder_array ADDARRAY(
        .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(dout0),
        .dout1(dout1),
        .dout2(dout2),
        .dout3(dout3),
        .overflow(overflow)
    );


endmodule
```

<lab4\lab4.srcs\sim_1\new\tb_adder_array.v>

테스트벤치 코드가 하는일은 다음과 같다.

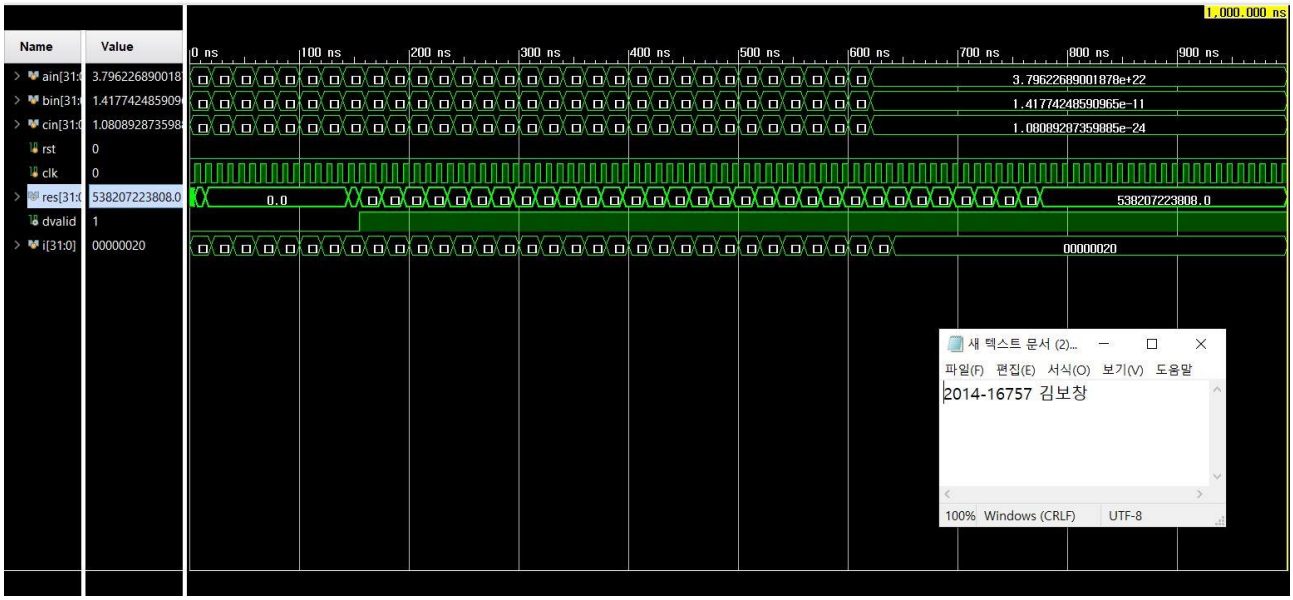
cmd를 0~7까지 생성하고, 각 cmd에 맞춰 ain0~3, bin0~3에 32비트 랜덤 벡터를 생성하여 대입하는 것을 4번 반복한다.

그 후, 약간의 딜레이를 두고 오버플로우가 발생했을때 제대로 작동하는지 체크하기 위해,

cmd를 4로 두어 모든 adder의 결과가 출력되도록 한뒤, 오버플로우가 발생하도록 인풋을 만들어준다.

3. Result & Discussion

3.1 floating point multiply-adder

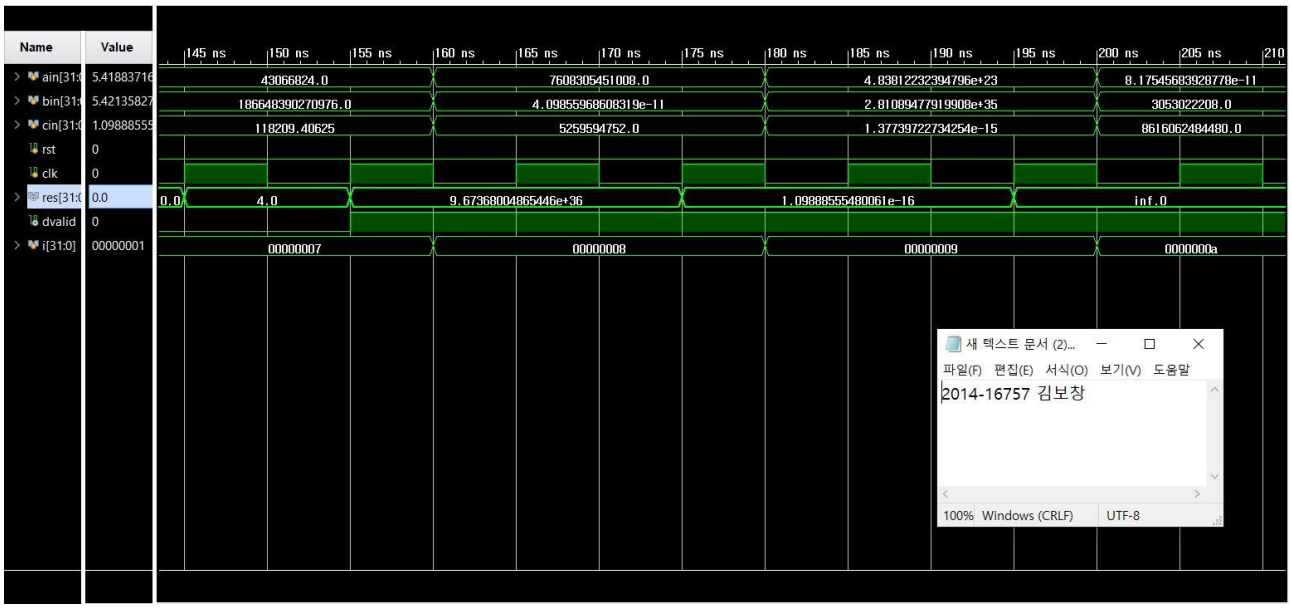


<전체 wave form>

<waveform의 첫부분을 확대한것>

첫부분에서는, input이 들어간 뒤 result가 출력되기까지 16사이클이 걸리기 때문에

res의 값으로 정상적인 값이 나오지 않고있다. (dvalid값이 0으로, 사용할 수 없는값)



<처음부분으로부터 약 16사이클 뒤>

결과를 보면, dvalid가 1이 된 순간부터 처음 입력에 대한 계산결과가 나오고 있다.

처음 나오는 result 는 약 9.67 * 10^36으로, 첫 ain 이 5.97 * 10^21, bin 이 1.61 * 10^15, cin 이 1.01 * 10^-5였던것을 감안하면, result = a*b + c 의 값이 제대로 출력되는 것을 알 수 있다.

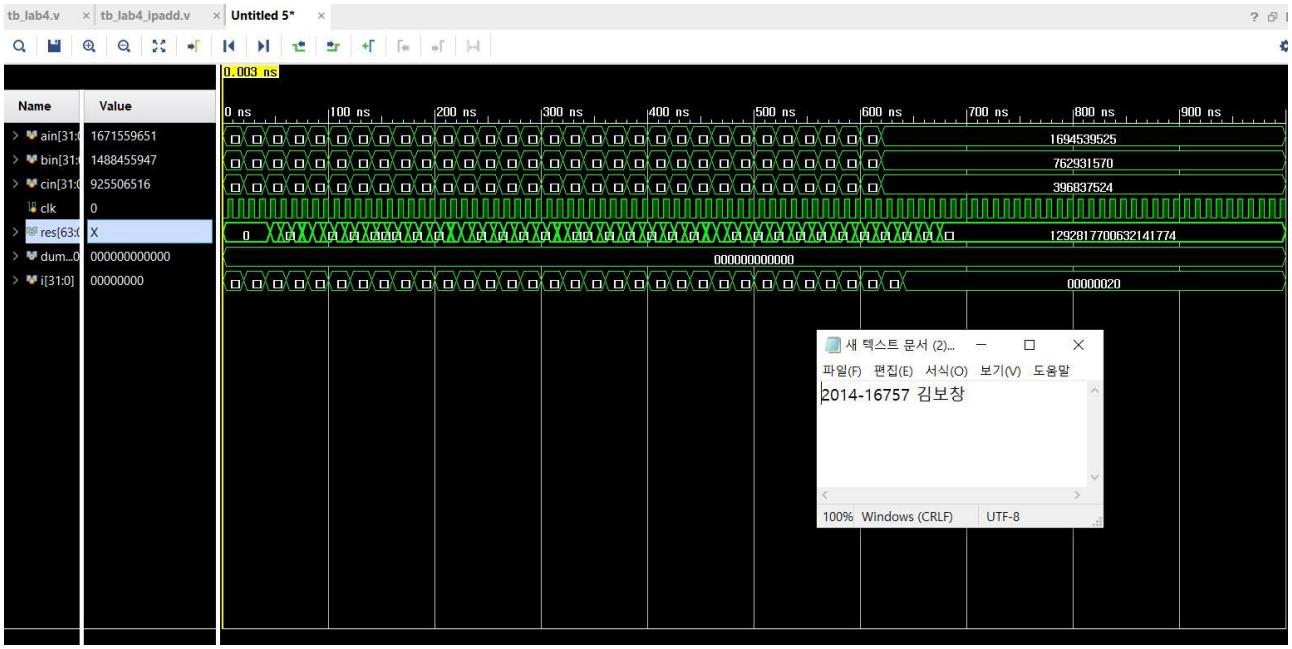
두번째로 나오는 result 역시 ain=5.41 * 10^-16, bin=5.42 * 10^-33, cin = 1.09 * 10-16이었던것을 감안하면, ain * bin 의 값이 매우 작으므로, result 로 cin 의 값이 나오는 것을 확인할 수 있다.

이는 single precision 의 표현 가능한 가장 작은 양수가 1.401 × 10−45.정도이므로, 이 이하의 양수는 0으로 취급되게 되어 ain*bin 의 값이 0으로 표기되어 무시된 것이다.

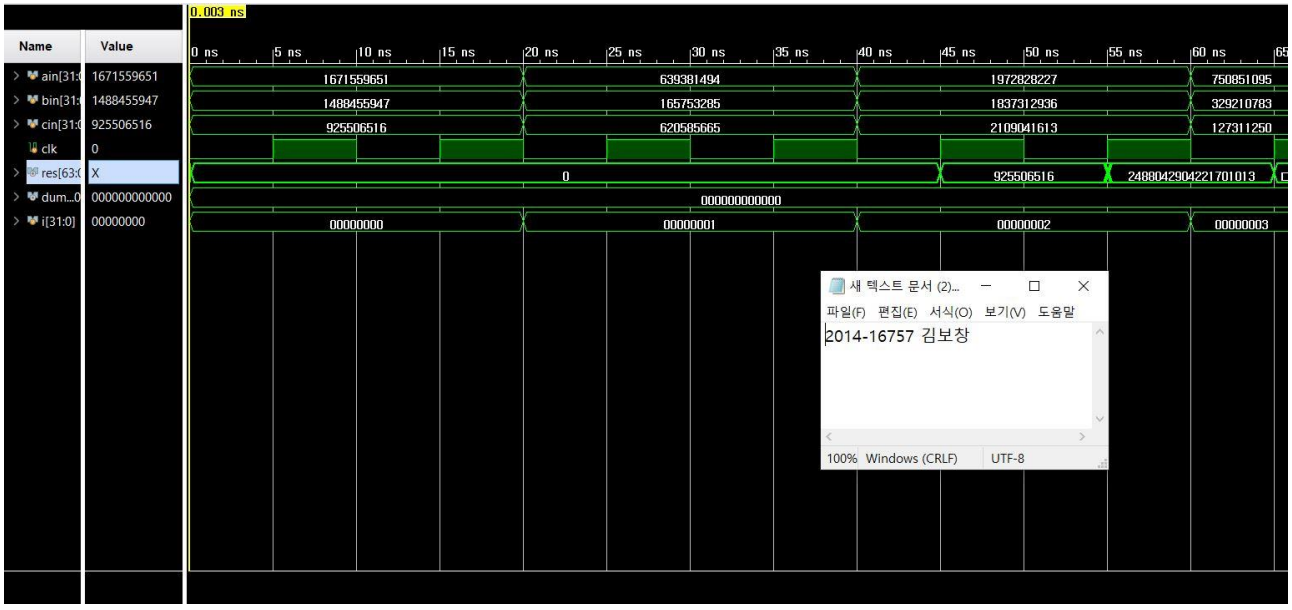
세번째로 나오는 result는 특이하게 infinity가 나오는데, 이 역시 single precision이 표현가능한 가장 큰 값은 3.4028235×10^{38} 정도로, ain = 3.82 * 10^32, bin = 5.07 * 10^27, cin = 3.01 * 37로, ain * bin이 10^39를 넘기때문에 single precision으로 표현가능한 범위를 넘어 infinity로 표기되고, 따라서 result가 infinity가 나온것이다.

결론적으로, 기대한 대로 32bit floating point 연산이 제대로 잘 작동함을 알 수 있었다.

3.2 integer multiply-adder



<전체 waveform>



<wavefrom 첫번째 부분>

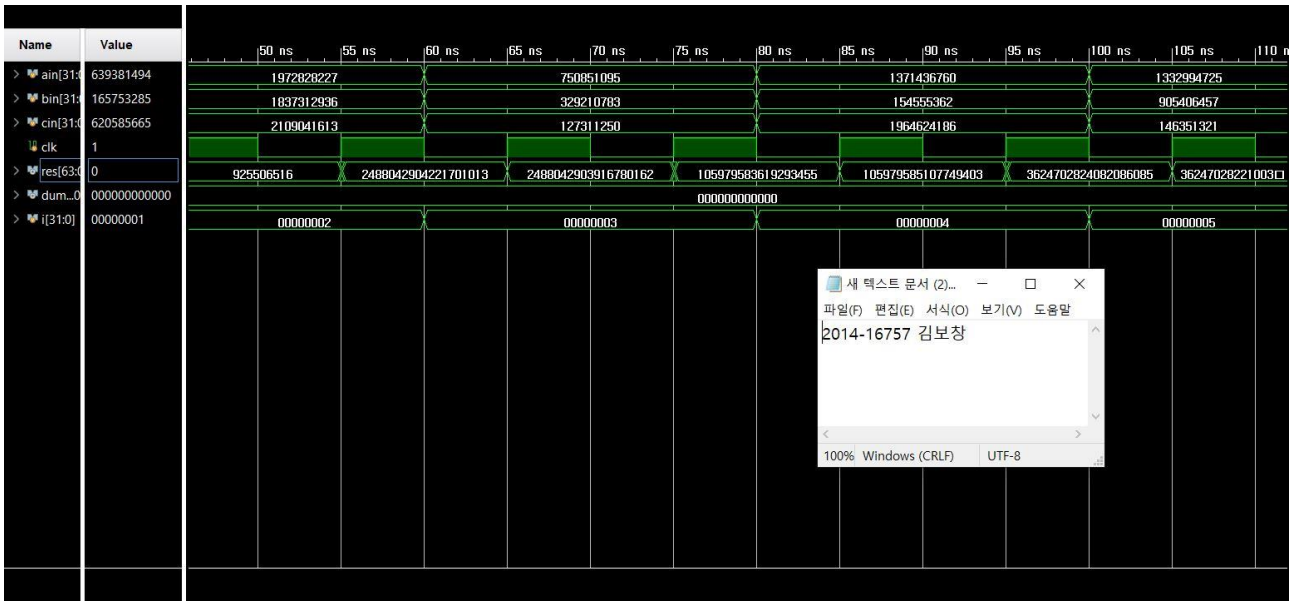
첫부분에서는, input이 들어간 뒤 result가 출력되기까지 6사이클정도가 걸리기 때문에,

그 전까지는 정상적인 출력이 나오지 않고 있다.

5~6사이클 부분의 925506516부분이 주목할 만 한데, cin의 결과가 그대로 출력되고 있다.

이는 integer multiply-adder에서 곱셈(A:B – P Latency)과 덧셈 (C-P Latency)의 fetch 타이밍이 각각 달라서,

새로운 A*B 의 결과가 계산되기 전에, 기존의 A*B의 값 (처음에는 0)이 C의 결과와 더해져 출력되기 때문이다.



<wavefrom 첫부분의 뒤쪽>

6사이클 후 (55ns~65ns)부분에서는 ain = 1671559651, bin = 1488455947, cin = 925506516일때,

res의 값으로 2488042904221701013으로,정상적으로 res = ain*bin + cin의 결과가 출력됨을 알 수 있다.

그 후, 65ns~75ns 부분에서는 ain * bin의 값의 fetch 타이밍이 더 늦기때문에,

20ns~40ns에 입력된 ain = 639381494, bin = 165753285, cin = 620585665로 계산된 res값이 아니라,

cin만 20ns~40ns의 것으로 바뀌고, ain과 bin은 0ns~20ns에 입력된 값으로 그대로 입력된 값이 출력됨을 확인할 수 있다.

20ns~40ns에 입력된 ain = 639381494, bin = 165753285, cin = 620585665로 계산된 res 값은

75ns~85ns때에서야 res = 105979583619293455로 정상적으로 출력됨을 확인할 수 있다.

결론적으로, 타이밍 이슈가 있긴 하지만, res의 값으로 ain*bin + cin의 값이 출력됨을 확인할 수 있다.

따라서 정상적으로 작동함을 확인할 수 있다.

3.3 integer adder array

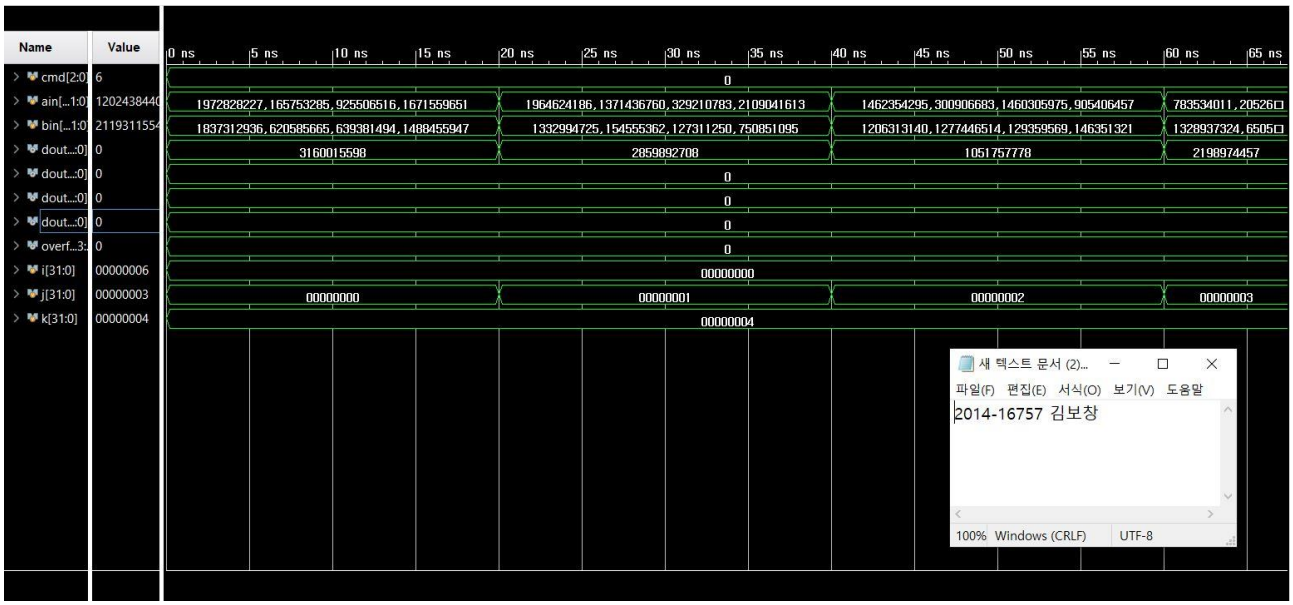
<전체 waveform 작동모습. 각 cmd에 따라 4개씩의 input들이 들어감을 알 수 있다.>

700ns이후는 테스트벤치 코드에서 700ns 이전에 랜덤생성한 인풋들이 overflow를 만들지 못하기 때문에,

임의로 overflow를 생성하는 인풋들을 넣어주고, cmd를 4로 설정하여 오버플로우 여부가 잘 출력되는지 확인한 것이다.

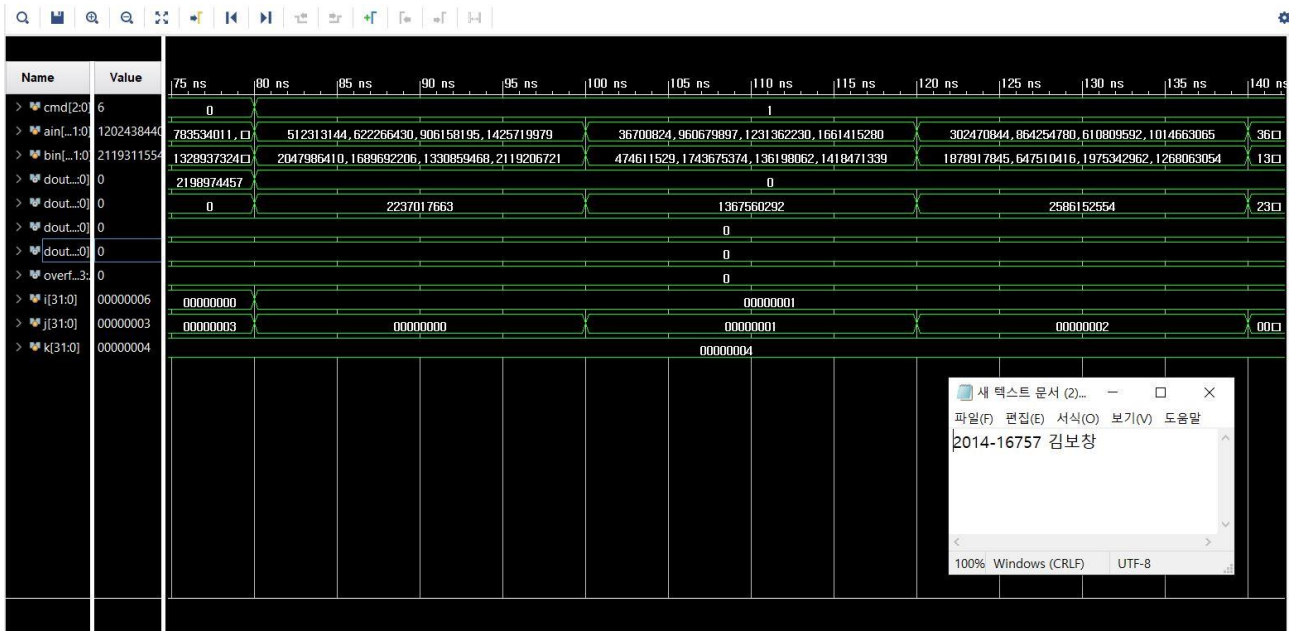
각각 0번과 2번에 해당하는 input에서 오버플로우가 발생하도록 의도하였고,

따라서 overflow의 0,2번째 비트에서 오버플로우가 검출됨을 확인할 수 있으므로 오버플로우 관련 기능은 정상적으로 작동함을 확인할 수 있다.

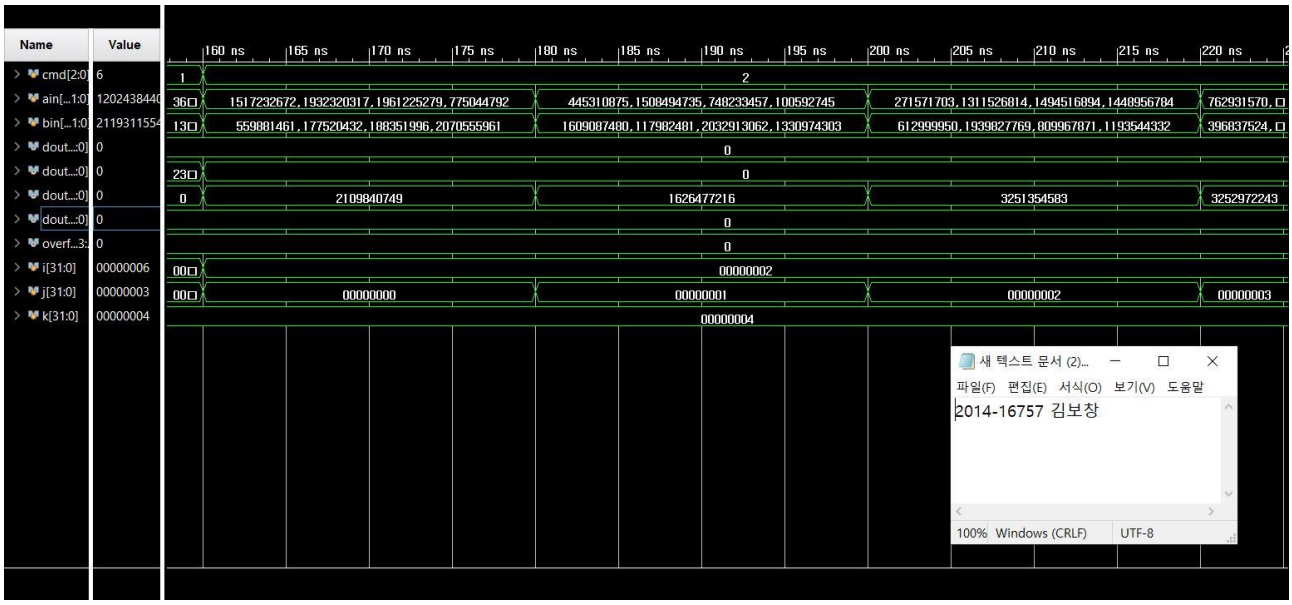


<cmd = 0일때의 작동모습>

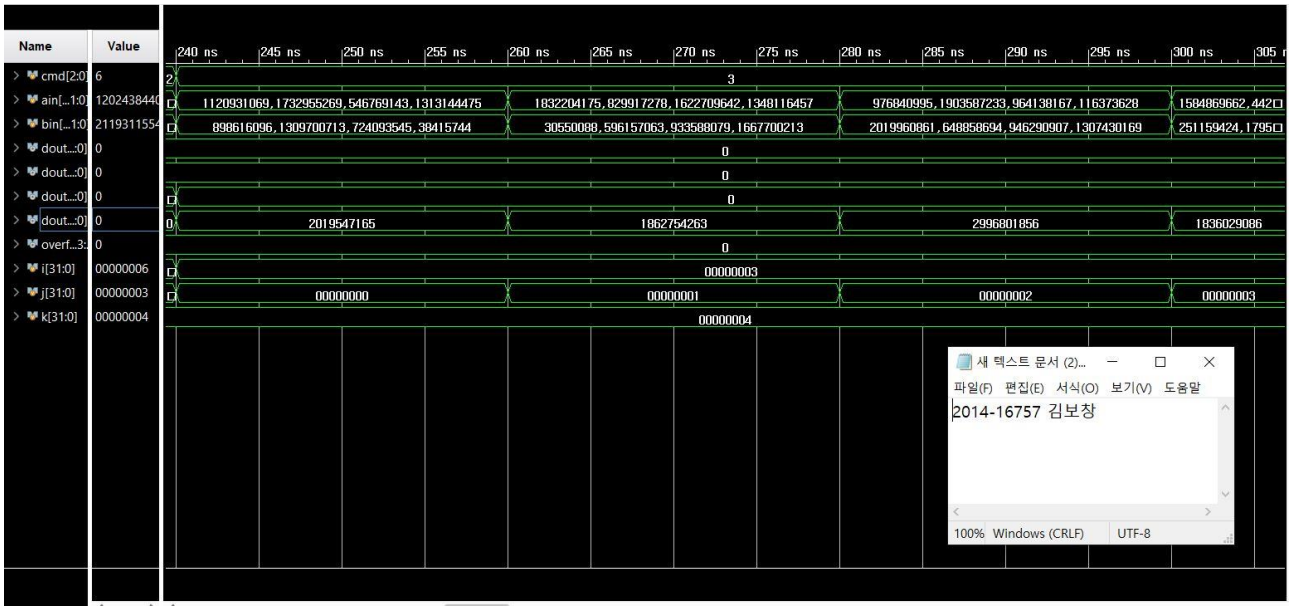
cmd가 0일때는 output으로 ain_0, bin_0의 계산 결과에 해당하는 dout0과 overflow의 0번째 비트만(LSB) 오버플로우 여부가 출력되고, 그 이외에는 정상적으로 0이 출력됨을 알 수 있다.



<cmd = 1일때의 작동모습>

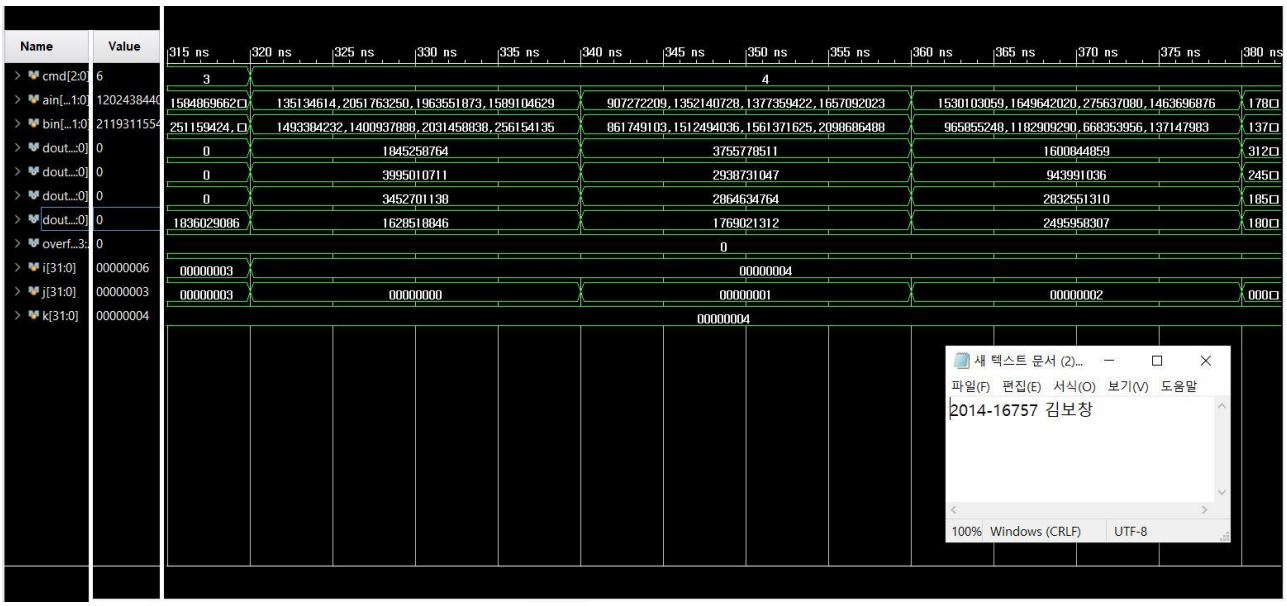


<cmd = 2일때의 작동모습>



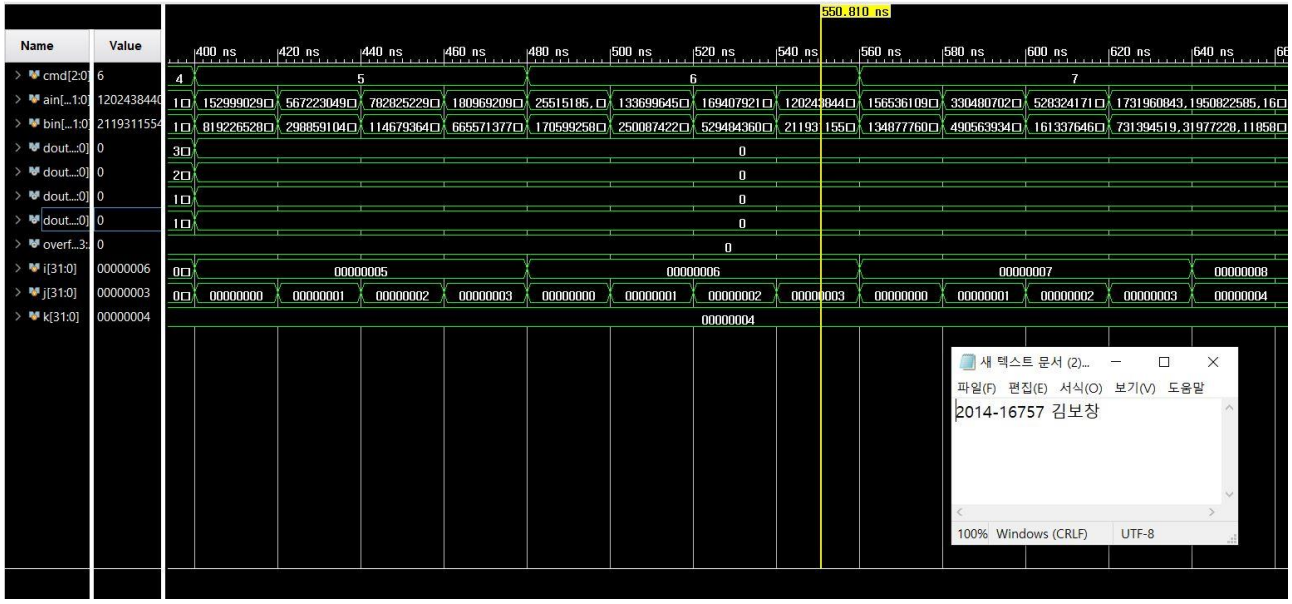
<cmd = 3일때의 작동모습>

위의 waveform 결과들을 보면, cmd가 각각 1, 2, 3일때도 결과로 해당하는 cmd 번호에 따른 ain+bin의 결과만 dout과 overflow의 값으로 출력됨을 확인할 수 있다.



<cmd = 4일때의 작동모습>

cmd가 4일때는 모든 ain_x, bin_x의 계산결과를 각각 해당하는 dout_x의 계산 결과와 overflow여부를 overflow 비트로 출력하므로, 알맞은 자리에 해당 값들이 출력됨을 알 수 있다.



<cmd = 5,6,7>

cmd가 5,6,7일때는 모든 dout과 overflow에서 0이 출력됨이 보인다.

결론적으로, 구현한 adder array가 의도대로 잘 작동함을 확인할 수 있었다.

지금까지의 구현에서, 다음과 같은 개선사항을 생각해 볼 수 있었다.

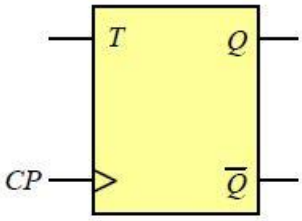
integer multiply-adder의 경우, 클럭사이클에 따라 유효하지 않은 output이 출력되는 경우가 있었는데,

이러한 output을 사용하는 경우가 생기지 않도록, 즉, 유효한 output만 사용할 수 있도록

synchronize 하는 회로를 추가하는 방법을 생각해 볼 수 있겠다.

integer multiply adder에서, clk의 posedge에서 입력된 ain,bin,cin의 값에 해당하는 유효한 output은

값의 입력 이후 5사이클째에 정상적으로 출력되므로,



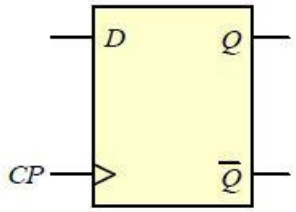
T	CP	$Q(t+1)$
0	↑	$Q(t)$
1	↑	$\overline{Q}(t)$

<posedge triggered T – flipflop>

위의 T플립플롭의 T 입력으로 1을 주면, CP가 posedge일때 기존에 저장된 값이 반전되므로,

CP에 CLK을 물렸을때, output Q의 결과로 CLK의 주기의 2배를 가지는 new_CLK가 생성된다.

CLK의 2배주기를 가지는 이 new_CLK를 잘 이용해서,



D	CP	$Q(t+1)$
0	↑	0
1	↑	1

<posedge triggered D-flipflop>

integer multiply adder의 output을 D-flipflop의 D input으로, new_CLK를 CP에 물려주면,

2사이클 주기로 값이 저장되고 출력되므로, 유효한 output일때만 D 플립플롭에 값이 저장되도록 synchronize 해준

후, D 플립플롭의 Q를 output으로 사용하는 방법도 생각해 볼 수 있다.

다만, 이러한 방법을 사용하면 회로에 포함되어야 하는 소자들이 많아지게 되고, 전체 회로의 결과의 delay가 flip

flop들의 delay만큼 증가하게 되므로, 비용적으로는 적절한 선택이라고 보기 힘들다.

4. Conclusion

Vivado의 IP catalog를 이용하여 기존에 있던 범용성 좋고 검증된 모듈들을 편하게 가져다 쓸 수 있음을 알았고, 생성한 모듈들을 synthesize, implement를 통해 실제 FPGA에 구현할 수 있다는 것을 알 수 있었던 보람찬 랩이었다.