Digital System Design
Team Project Final Report

Department	**Electrical & Electronic Engineering**
Group #	**6조**
Name	**2015142213 류성민**
	2014142122 정세영
	2014142141 최원영

목차

1. Verilog code of the floating point operator with explanation

(1) Explanation of the FPU module composition(fpu.v).

```
module fpu(
    input clk, input rst, input [31:0] a, input [31:0] b,input [1:0] sel,
    output reg err, output reg overflow, output reg [31:0] y);
```

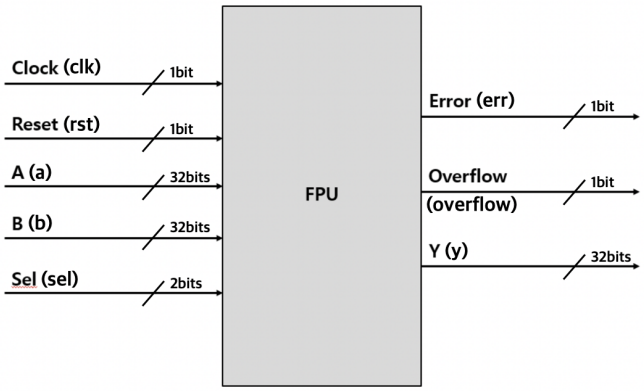


Figure 1.1 Design of our FPU

- Figure 1.1와 같이 FPU의 input은 각각 Clock(clk), Reset(rst), A(a), B(b), Select(sel)값으로 총 68-bit를 받고 output은 Overflow(overflow) bit, Error(error) bit와 32-bit의 Y(y)로 총 34-bit를 출력하도록 설계되었다. Figure 1.2는 input/output을 제외한 중간 변수들을 선언한 것이다.

```
/* Temperary variables */
// stage 0
reg         [1:0]       sel_0;
reg                     a_s, b_s;
reg         [7:0]       a_e, b_e;
reg         [23:0]      a_m;
reg         [23:0]      b_m;

// stage 1
reg         [1:0]       sel_1;
reg                     tmp_s0;
reg signed  [9:0]       tmp_e0;
reg         [47:0]      tmp_m0;
reg                     tmp_a_s0;
reg         [7:0]       tmp_a_e0;
reg         [23:0]      tmp_a_m0;
reg                     tmp_b_s0;
reg         [7:0]       tmp_b_e0;
reg         [23:0]      tmp_b_m0;

// stage 2
reg         [1:0]       sel_2;
reg                     tmp_s1;
reg signed  [9:0]       tmp_e1;
reg         [47:0]      tmp_m1;
reg                     tmp_a_s1;
reg         [7:0]       tmp_a_e1;
reg         [23:0]      tmp_a_m1;
reg                     tmp_b_s1;
reg         [7:0]       tmp_b_e1;
reg         [23:0]      tmp_b_m1;
```

Figure 1.2 Registers of our FPU

- Figure 1.3과 같이 기본 동작은 pipeline을 통해 이루어지며, 총 4개의 stage를 구성하였고 각각의 stage에 해당하는 register가 존재한다.

- 매 clock의 posedge마다 해당 stage는 이전 stage에서 넘겨받은 register값을 통해 동작을 수행하게 된다.

- 다음 stage로 넘어가는 register 값은 초기 input으로 받은 sel, a, b, 그리고 exponent와 mantissa의 연산 결과이다.

- sel bit의 경우 stage마다 해당 연산을 위해 항상 register으로 넘겨주도록 설계했다.

- sel 이외에 stage 0에 사용되는 register는 input a, b의 sign, exponent, mantissa를 저장하는 register이다.

- stage 1~2에서 사용되는 register는 tmp_x_y 또는 tmp_y의 형태로 표현되며 x는 input register의 구분, y는 floating point의 성분을 나타낸다.

- stage 3의 경우 output 출력 단계이므로 추가적인 register는 존재하지 않는다.

- overflow check를 위하여 temp register의 exponent의 bit는 1-bit 높게 설정되었다.

- stage1, 2의 tmp_m0, 1은 24bit의 multiple 연산을 위해 48-bit로 설정하였다.

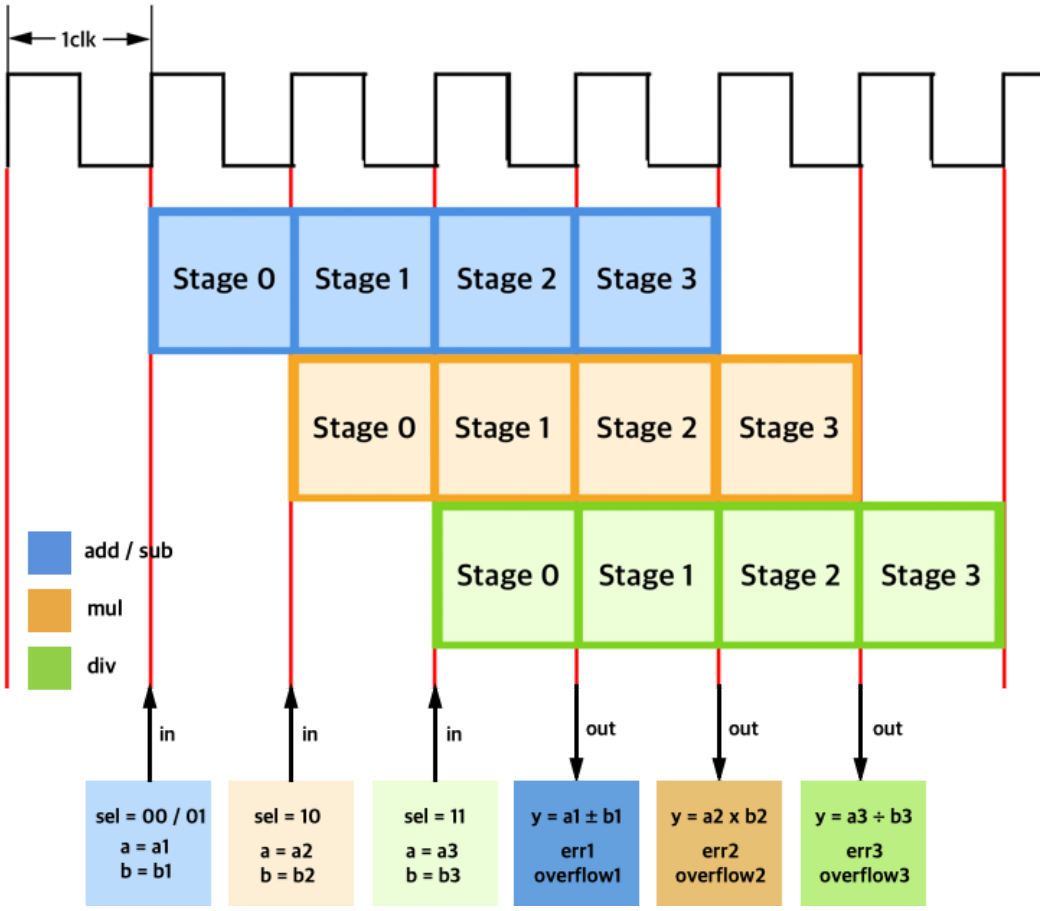


Figure 1.3 Stages of our FPU

- stage는 총 4개로 구성되며, input 입력 후 3 clock 뒤에 output이 출력된다.

- input은 1 clock마다 입력받을 수 있으며, 각각의 output은 3 clock 뒤에 출력된다.

- 각 stage에 대한 전체적인 기능은 다음과 같다.

< Stage 0>

- 입력 a,b를 sign, exponent, mantissa를 구분하여 register에 할당해준다.

< Stage 1>

1. Plus / Minus

- 입력의 부호와 sel 값를 판별하여 output의 부호와 연산을 결정한다.

- Exponent와 mantissa의 대소를 비교하여 절대값이 큰 값에서 작은 값을 연산해준다.

- 연산 결과를 register에 저장하고 stage2로 넘어간다.

2. Multiple / Divide

- Multiple의 경우 exponent를 더하고, Divide의 경우 빼준다.

- mantissa는 그대로 연산하여 결과들을 register에 저장하고 stage 2로 넘어간다.

< Stage 2>

- Stage1의 결과를 normalization하여 tmp register에 저장하고 stage3로 넘어간다.

< Stage 3>

- 입력값에 대한 exception을 확인한다.

- 결과값에 대한 overflow 및 error을 확인하여 exception을 확인한다.

- mantissa의 LSB(multiple의 경우 tmp_m1[23], 나머지의 경우 tmp_m1[0])에서 round half up을 시행하고 이로 인한 overflow를 확인한다.

- output y에 정리된 결과를 저장하여 출력한다.

(2) Explanation of the exception handling

- 모든 계산은 exception이 발생하지 않았을 경우, IEEE754의 규칙을 따라 연산하도록 한다.

- output이 각각 NaN, INF, zero에 대해 Table 1.1과 같이 err와 overflow를 표현했다.

- INF에 대해서는 +INF와 -INF가 존재하며 해당 error 발생 시 sign bit에 0 or 1, 지수부의 모든 bit를 1로, 가수부의 모든 bit를 0으로 채우고 부호에 따라 양과 음을 구분한다.

- Underflow의 경우 지수부를 전부 0으로 채우고 overflow bit와 반환하도록 한다.

- NaN의 경우 error을1, overflow를 0, 지수부와 가수부의 모든 bit를 1로 채우고 반환하도록 한다.

- 계산 결과가 0이되는 operating results 0의 경우, 경우에 따라 +0와 -0을 구분하도록 한다.

- 0과 0의 계산에서, 0의 부호에 따라 exception handler를 발생시키고 따로 정해진 규칙에 따라 연산처리하도록 한다.

output	err	overflow
NaN	1	0
inf	0	1
zero	0	1

Table 1.1 Representation of output bits

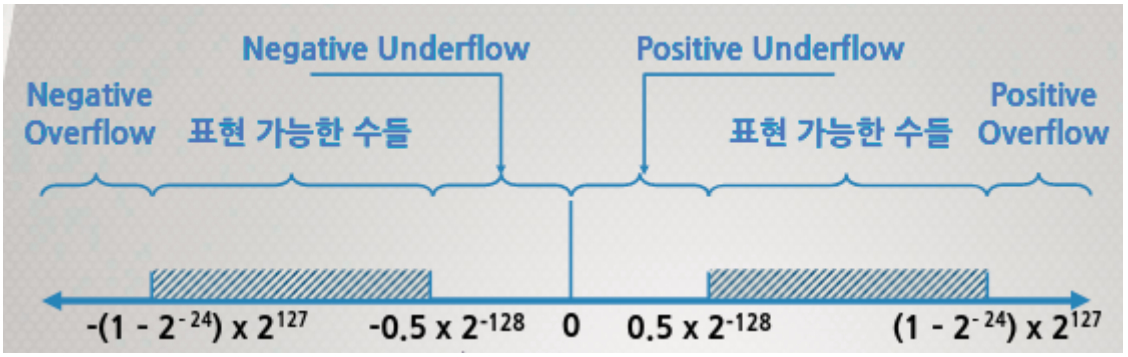


Figure 1.4 32bits floating point expression

- Table 1.2는 모든 exception 경우에 따른 표이다. Testbench 결과로 확인이 가능하다.

	Input		Output
	a	b	y
Plus / Minus	A	NaN	NaN
	NaN	B	
	±inf	±inf	
	±inf	B	±inf
	A	±inf	
	±0	B	B
	A	±0	A
Plus	A	-A	0
Minus	A	A	
Multiply	A	NaN	NaN
	NaN	B	
	±0	±inf	
	±inf	±0	
	A	±inf	±inf
	±inf	B	
	A	±0	±0
	±0	B	
Divide	A	NaN	NaN
	NaN	B	
	±inf	±inf	
	A	±0	
	±inf	B	±inf
	A	±inf	±0
	±0	B	

Table 1.2 Exception handling chart

(3) Explanation of the round half up

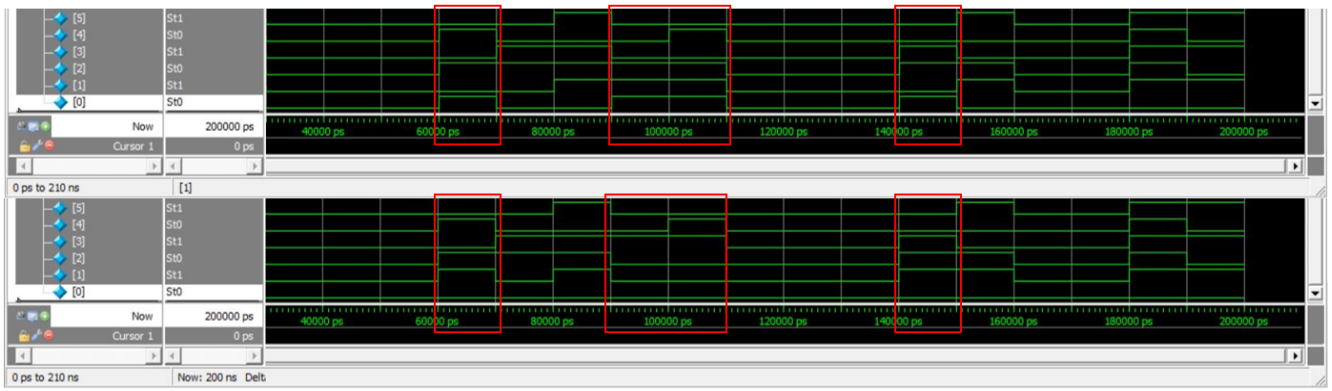


Figure 1.5 Comparison between non-rounding & rounding results

- 우리의 디자인에서 round half up은 mantissa의 LSB에서 이루어진다. 즉 output의 LSB는 항상 0이다.

- LSB가 0일 경우 그대로 출력하며, 1일 경우에는 다음 bit로 올림을 시행한다.

- Figure 1.5는 위부터 각각 rounding을 거치지 않았을 경우와 round half up을 시행한 결과를 비교한 것이다. 이를 통해 성공적으로 rounding이 진행되고 있음을 알 수 있다.

- rounding은 모든 연산이 끝나고 출력 직전에 시행하였다. rounding의 시기를 normalize이전으로 잡느냐 이후로 잡느냐에 따른 차이가 발생한다.

  예를 들어 exponent와 mantissa의 총 bit수가 3bit인 임의의 floating point를 가정할 때, stage 1의 연산에 의한 mantissa의 결과가 1110, 1111로 다른 두 수 x, y에 대해 rounding을 진행해보자(연산 결과 sign은 0, exponent도 0으로 가정한다.)

  1. rounding의 시기를 normalize 이전으로 잡게 되면, 각각 x = 1110, y = 10000가 된다.
  2. rounding의 시기를 normalize 이후로 잡게 되면, 각각 x = 1000, y = 1000이 된다.

  이 경우 1의 x는 normalize 과정에서 shift가 1번, y는 2번 일어나게 된다.

  2의 경우에서는 normalize 과정에서 먼저 shift가 각각 1번 일어나고 rounding 이후 각각 1번으로 총 2번이 일어난다.

  결과를 표로 정리하면 다음과 같다.

연산 결과	x		y	
	Exponent	Mantissa	Exponent	Mantissa
1의 경우	001	111	010	100
2의 경우	010	100	010	100

Table 1.3 operation results according to rounding timing

우리의 디자인에서는 간편한 연산과 clock의 감소를 위해 2번의 방법을 채택하였다. 1과 달리 2의 방법을 택할 경우 한 번 더 overflow의 체크를 해야한다는 점에서 area와 trade-off가 존재한다.

2. Verilog code of the testbench with explanation

(1) Summary of our testbench code(fpu_tb.v)

```
`timescale 1ns/100ps


module test; reg clk; reg rst; reg [31:0] a; reg [31:0] b; reg [1:0] sel; wire err;
             wire overflow; wire [31:0] y;
fpu fpu_test(clk, rst, a, b, sel, err, overflow, y);


   always // clock generation
   begin
       #0.5 clk = ~clk;
   end


   initial
   begin
       clk <= 1; rst <= 0; a <= 0; b <= 0; sel <= 0;
/*************** sel = 0: add ***************/
// all exception case
       #0.9
       // y =  nan
       a <= 32'b0_11111111_00000000000000000000000;  // a =  inf
       b <= 32'b0_11111111_00000000000000000000000;  // b =  inf
       #1
       // y =  -nan
       a <= 32'b1_11111111_00000000000000000000000;  // a =  -inf
       b <= 32'b0_11111111_00000000000000000000000;  // b =  inf
       ...(생략)...
/*************** sel = 1: sub ***************/
       ...(생략)...
/*************** sel = 2: mul ***************/
       ...(생략)...
/*************** sel = 3: div ***************/
       ...(생략)...
   end
endmodule
```

(2) Explanation of the the code

Testbench에는 설계한 FPU의 verification을 위한 방법으로 code coverage를 사용하였다.

각 operator에 대한 입력의 exception을 전부 test 해봤으며, 입력이 general하더라도 출력이 exception에 해당되는 부분 또한 확인하였다.

기본적인 규칙은 IEEE 754에 따라 sign, exponent, mantissa를 결정한다.

<Add(00) & Sub(01)>

1. sel[0]과 input a, b의 sign의 bit를 통해 두 수를 더할지 뺄지 결정하게 된다.
2. +연산의 경우 같은 sign인 경우 더하고 다른 sign인 경우 빼준다. 반대로 -연산의 경우 다른 sign인 경우 더하고 같은 sign인 경우 빼준다.

   (다른 방법으로는 bit가 1인 수가 짝수면 두 수를 더하고, 홀수면 두 수를 빼주며 XOR을 통해 판별 가능하다.)
3. exponent가 같은 수의 연산은 mantissa끼리 더하거나 빼고 그대로 exponent를 가져온다.

   exponent가 다른 수의 연산은 exponent가 작은 수를 큰수에 맞추고 mantissa를 shift하여 연산한다.
4. add 연산의 경우 sign bit는 a와 b중 절대값이 큰 쪽의 부호를 따라간다.

   sub 연산의 경우 sign bit는 a와 ~b중 절대값이 큰 쪽의 부호를 따라간다.
5. add와 sub 연산이 나머지 mul과 div와 다른 점 중 하나는 mantissa 연산 후 shift 과정이 까다롭다는 것이다.
   a와 b의 연산 후 mantissa의 MSB부터 낮추며 bit를 확인하며 0이 아닐때까지 왼쪽으로 shift하고 exponent를 빼준다. 만일 MSB가 통상보다 높은 자리의 bit라면 mantissa를 오른쪽으로 shift하고 exponent를 더한다.

   회로적 낭비를 줄이기 위해 shift 판별은 mantissa bit를 절반씩 쪼개가며 확인하였다.
6. 마지막 stage에서 overflow나 Nan 등 exception을 확인하고 이상이 없을 시 rounding 하여 연산 결과를 출력한다.

<Mul(10)>

1. output의 sign은 input a와 b의 XOR 연산을 통해 결정한다.
2. 두 수의 exponent는 더하고, mantissa는 곱한다.
3. tmp_m0[47] bit가 1일 시 자릿수 초과이므로 exponent에 1을 더해주고 mantissa는 오른쪽으로 shift한다.
4. 마지막 stage에서 overflow나 Nan 등 exception을 확인하고 이상이 없을 시 rounding 하여 연산 결과를 출력한다.

<Div(11)>

1. output의 sign은 input a와 b의 XOR 연산을 통해 결정한다.
2. 두 수의 exponent는 빼고, mantissa는 a의 뒤에 23'd0을 붙이고 나누어준다. 이는 output의 mantissa를 충분히 보장하기 위함이다.
3. tmp_m0[23] bit가 0일 시 자릿수 미만이므로 exponent에 1을 빼주고 mantissa는 왼쪽으로 shift한다.
4. 마지막 stage에서 overflow나 Nan 등 exception을 확인하고 이상이 없을 시 rounding 하여 연산 결과를 출력한다.

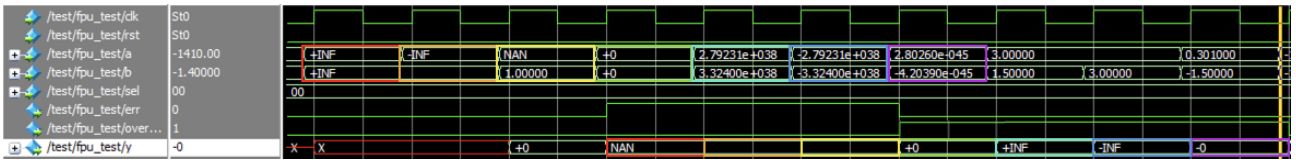
3. Waveform simulation results of the floating point operator with explanation

posedge에서 받은 input을 통해 매 edge마다 다음 stage로 넘겨주며 연산하도록 짜여졌다.

Code coverage를 위하여 exception case의 숫자들은 exception code 부분의 모든 사항을 test 하였으며, general case의 숫자들은 code를 한번씩 전부 지날 수 있도록 정하였다.

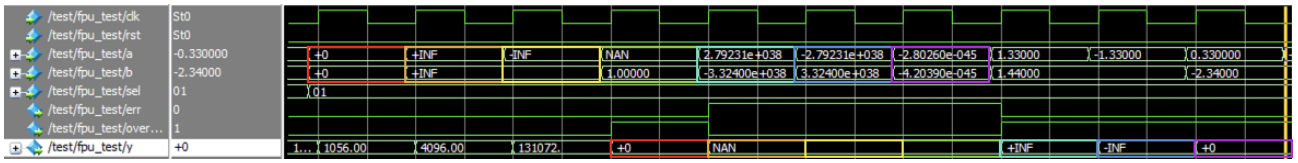
(1) Results of the exception cases.

<Add(00) exception>



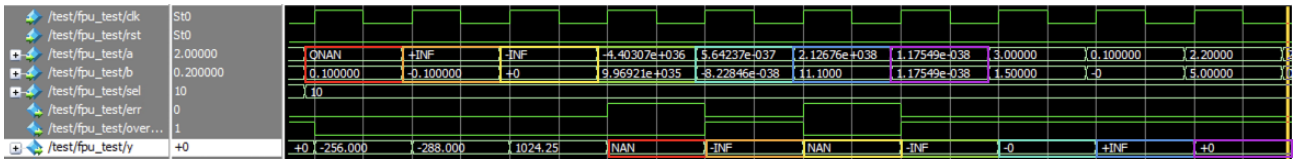
- 연산에서 input으로 NAN이 존재하거나 두 input이 INF일 경우 NAN을 출력하도록 설계하였는데 결과는 각각 빨, 주, 노의 색상에서 확인 가능하다.
- 큰 input의 계산으로 인해 표현범위를 초과하면 overflow = 1을 주고 INF를 출력한다. 결과는 푸른색에서 각각 확인 가능하다.
- 표현 가능한 작은 수의 범위를 넘어서면 underflow이며, 우리의 fpu 디자인에서는 overflow = 1을 주고 0을 출력한다. 결과는 보라색에서 확인 가능하다.
- 0의 출력의 경우 일괄적으로 overflow에 1을 할당한다.

<Minus(01) exception>



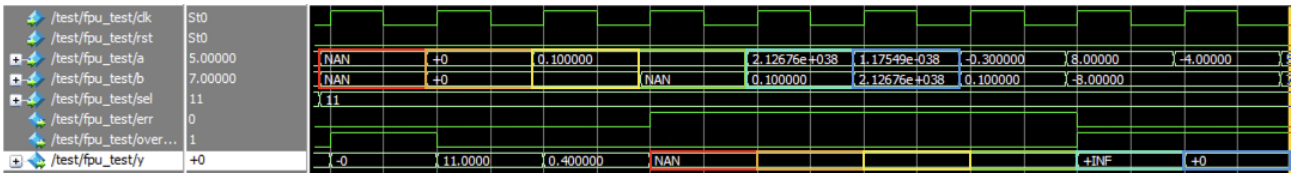
- 위의 표에 따라 NaN일때는 err = 1이며 ±INF 일때는 overflow = 1이다. 주황, 노랑, 초록색의 결과에서 확인 가능하다.
- add 연산과 마찬가지로 minus도 같은 원리로 exception을 처리한다.

<Multiple(10) exception>



- 곱셈 연산의 경우도 input이 NaN인 경우에 대해서 output은 NaN이다. 덧셈, 뺄셈과 마찬가지로 overflow와 underflow, INF를 출력하며 위의 그림에서 각각 확인 가능하다.

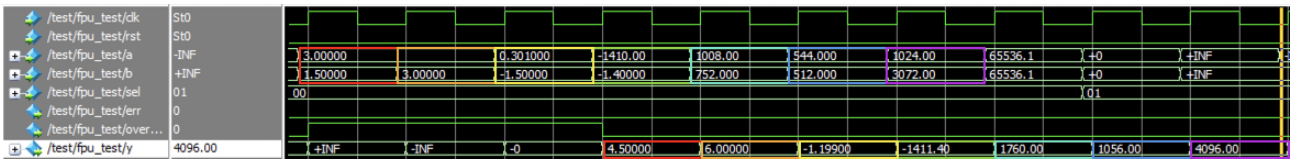
<Divide(11) exception>



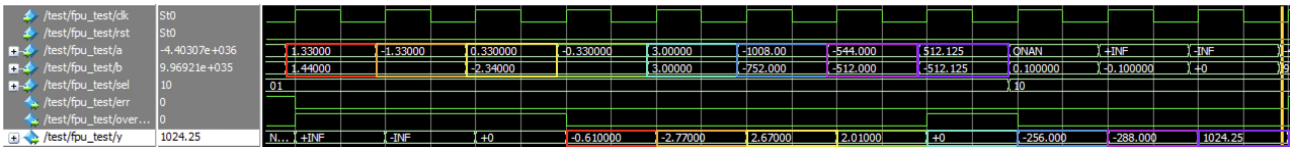
- 다른 연산과 달리 나눗셈 연산에는 divide by zero가 존재한다. 위의 그림에서 주황색에 해당하며 NaN을 출력하고 err = 1을 출력한다.
- 나머지 exception에 대해서는 마찬가지로 Nan 또는 overflow, underflow 등이 존재하며 표현은 동일하다.

(2) Results of the general cases

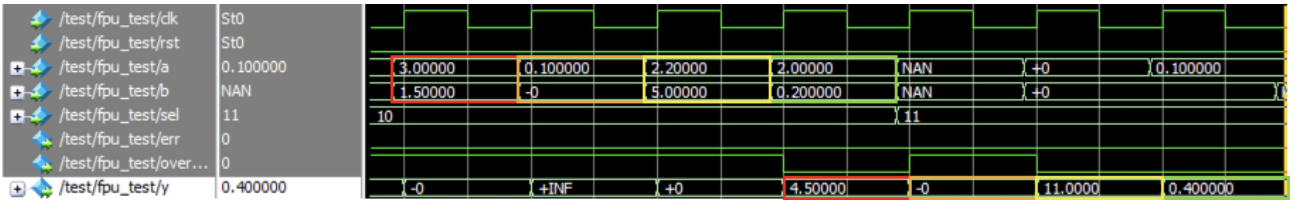
<Add(00) general>



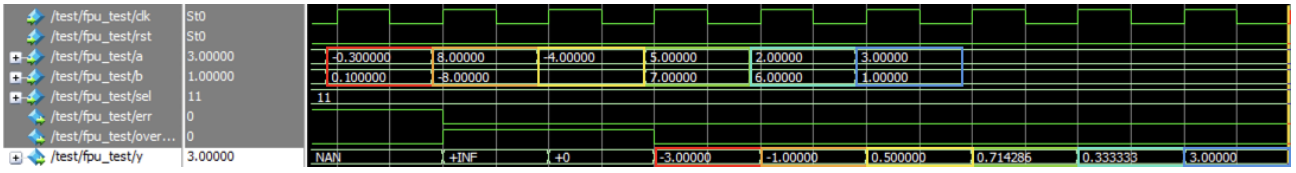
<Minus(01) general>



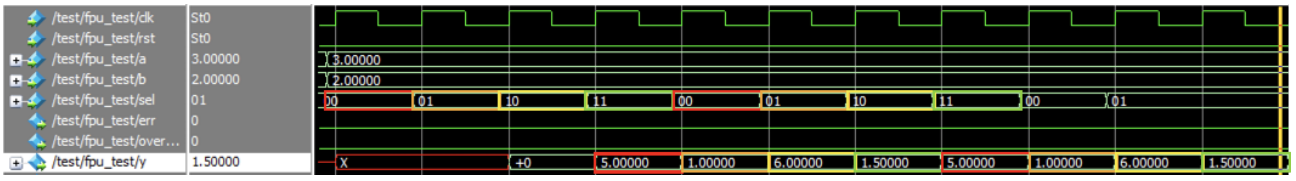
<Multiple(10) general>



<Divide(11) general>



<Mixed general cases>

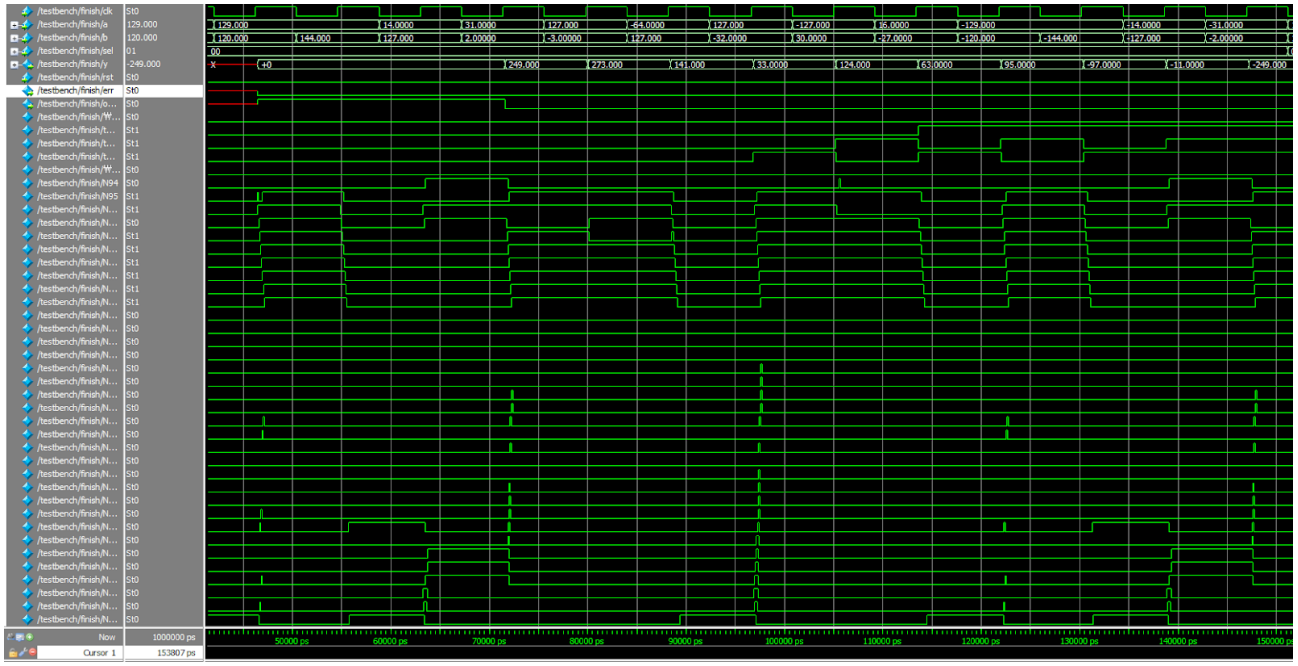


- 위의 general case들은 mantissa의 shift, exponent의 증가, round half up 등을 고려한 bit로 할당된 case들이다. 정상적인 출력값을 가지는 것으로 우리의 fpu 디자인이 성공적이었음을 확인할 수 있다.
- 모든 general case에 대해 순차적으로 들어오는 input에 따라 output이 정상적으로 출력됨을 확인할 수 있다. sel값이 변하더라도 register에서 함께 가져가기 때문에 정상적으로 pipeline이 동작할 수 있었다.

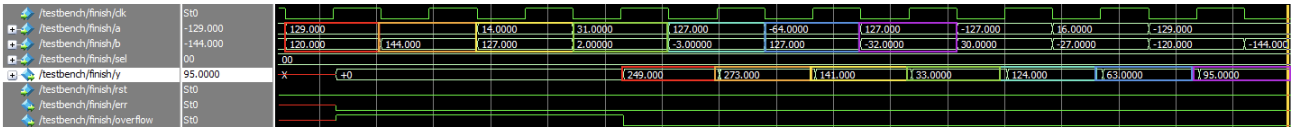
4. Waveform simulation results of the synthesized floating point operator with explanation

(1) Simulation results for using a golden testbench.

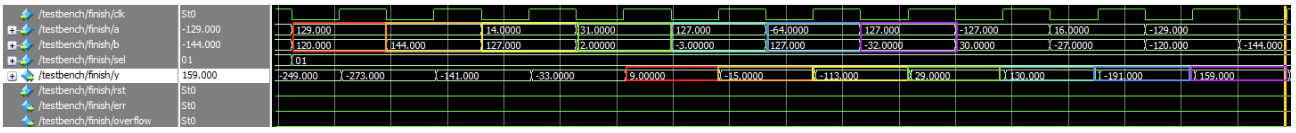
<전체 화면>



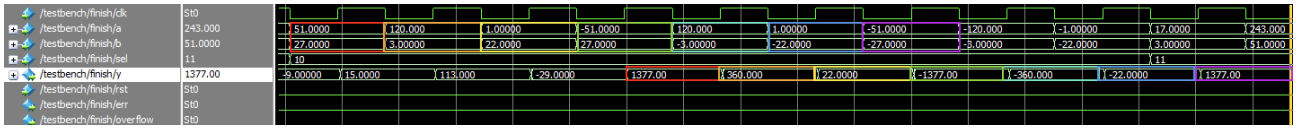
<Add(00)>



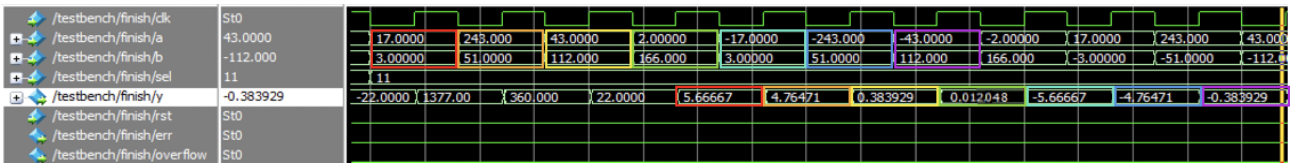
< Minus(01) >



<Multiple(10)>



<Divide(11) >



(2) Explanation of the results

$ create_clock { "clk" } -name "clk" -period 9 -waveform { 0 4.5 }
$ compile
$ write -format verilog -hier -output ../outputs/fpu_mapped.v

위 명령어로 fpu.v를 합성 후 fpu_mapped.v를 golden testbench와 함께 시뮬레이션 한 결과들이다. period를 9ns로 했기때문에 golden testbench의 STEP을 9로 바꿔주고 진행했다. 결과들은 모두 잘 나왔다.

```
1   `timescale 1ns/100ps
2
3   module testbench;
4       reg Clock, Reset;
5       reg [31:0] A, B;
6       reg [1:0] Sel;
7       wire [31:0] Y;
8       wire Overflow, Error;
9       parameter STEP=9; //It is should be modified depending on clock frequency.
10
```

5. Schematic view of the floating point operator with explanation

(1) Schematic views of the FPU

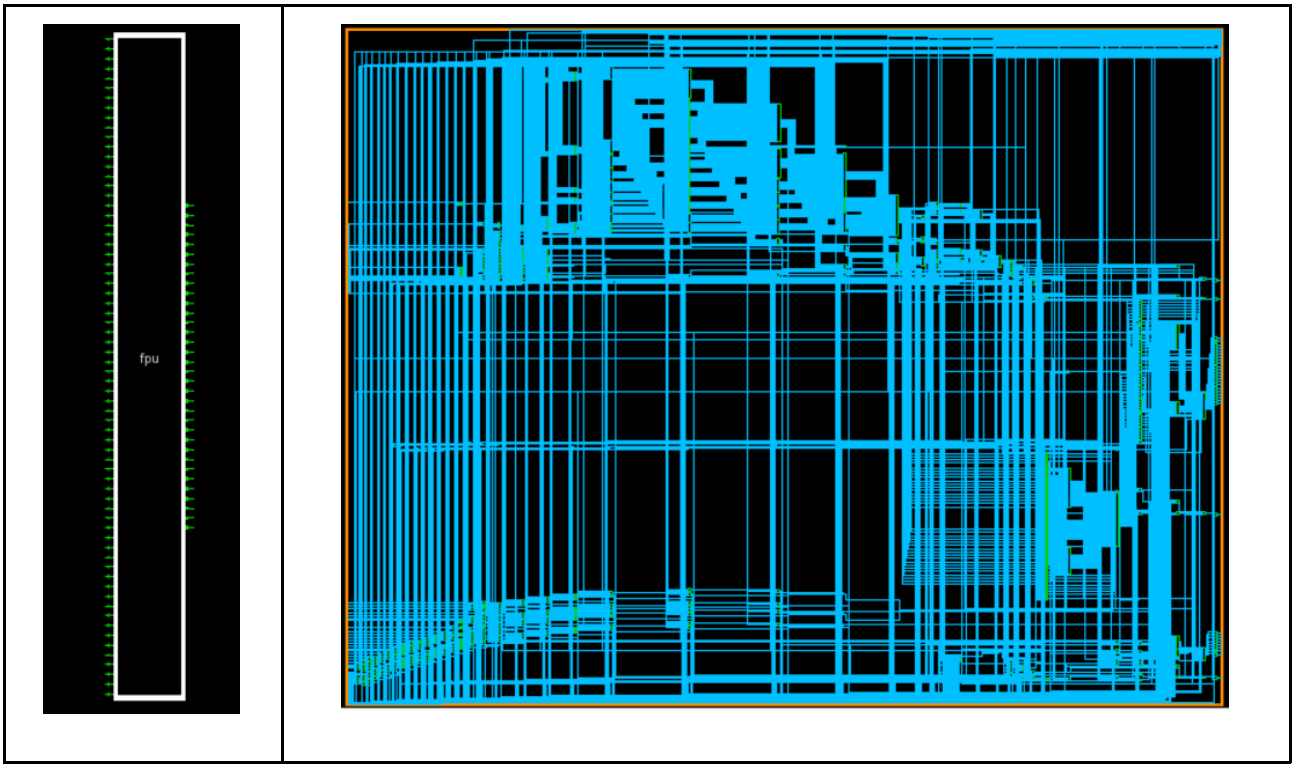


Figure 5.1 Overall schematic of our FPU

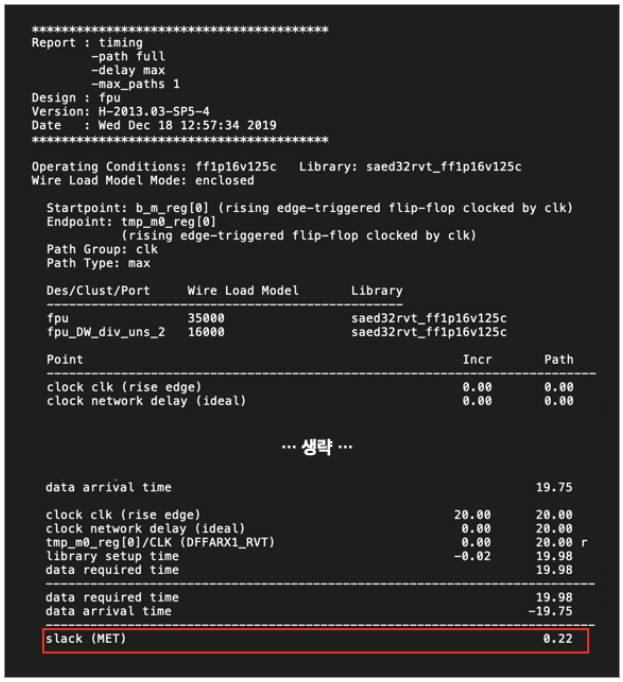
(2) Explanation of the schematic views

Figure 5.1은 RTL level로 설계된 FPU verilog file을 Design vision을 통해 확인한 결과이다. input과 output의 모든 bit를 보여주는 형태로 길게 늘여진 모습을 확인가능하다. 일련의 명령어를 통해 gate level로 synthesize된 결과는 figure 5.1의 우측 schematic에서 확인가능하다. clock, area effort에 따라 결과는 변동가능하다. 실제로 verilog로 작성된 floating point unit이 무수히 많은 gate와 wire등을 통해 합성되었다.

6. Area reports with analysis

(1) Area reports with 50MHz clock(Total area: 28194.85)

$ create_clock { "clk" } -name "clk" -period 20 -waveform { 0 10 }
$ compile
$ compile -area_effort high



```
****************************************
Report : timing
        -path full
        -delay max
        -max_paths 1
Design : fpu
Version: H-2013.03-SP5-4
Date   : Wed Dec 18 12:57:34 2019
****************************************

Operating Conditions: ff1p16v125c   Library: saed32rvt_ff1p16v125c
Wire Load Model Mode: enclosed

  Startpoint: b_m_reg[0] (rising edge-triggered flip-flop clocked by clk)
  Endpoint: tmp_m0_reg[0]
            (rising edge-triggered flip-flop clocked by clk)
  Path Group: clk
  Path Type: max

  Des/Clust/Port     Wire Load Model       Library
  ------------------------------------------------
  fpu                35000                 saed32rvt_ff1p16v125c
  fpu_DW_div_uns_2   16000                 saed32rvt_ff1p16v125c

  Point                                                   Incr       Path
  ------------------------------------------------------------------------
  clock clk (rise edge)                                   0.00       0.00
  clock network delay (ideal)                             0.00       0.00

                                  ··· 생략 ···

  data arrival time                                                 19.75

  clock clk (rise edge)                                  20.00      20.00
  clock network delay (ideal)                             0.00      20.00
  tmp_m0_reg[0]/CLK (DFFARX1_RVT)                         0.00      20.00 r
  library setup time                                     -0.02      19.98
  data required time                                                19.98
  ------------------------------------------------------------------------
  data required time                                                19.98
  data arrival time                                                -19.75
  ------------------------------------------------------------------------
  slack (MET)                                                       0.22
```

< area optimization X >

```
****************************************
Report : timing
        -path full
        -delay max
        -max_paths 1
Design : fpu
Version: H-2013.03-SP5-4
Date   : Wed Dec 18 13:37:14 2019
****************************************

Operating Conditions: ff1p16v125c   Library: saed32rvt_ff1p16v125c
Wire Load Model Mode: enclosed

  Startpoint: b_m_reg[15]
              (rising edge-triggered flip-flop clocked by clk)
  Endpoint: tmp_m0_reg[0]
            (rising edge-triggered flip-flop clocked by clk)
  Path Group: clk
  Path Type: max

  Des/Clust/Port     Wire Load Model       Library
  ------------------------------------------------
  fpu                35000                 saed32rvt_ff1p16v125c
  fpu_DW_div_uns_2_0_0
                     16000                 saed32rvt_ff1p16v125c

  Point                                                   Incr       Path
  ------------------------------------------------------------------------
  clock clk (rise edge)                                   0.00       0.00
  clock network delay (ideal)                             0.00       0.00

                                  ··· 생략 ···

  data arrival time                                                 19.81

  clock clk (rise edge)                                  20.00      20.00
  clock network delay (ideal)                             0.00      20.00
  tmp_m0_reg[0]/CLK (DFFARX1_RVT)                         0.00      20.00 r
  library setup time                                     -0.02      19.98
  data required time                                                19.98
  ------------------------------------------------------------------------
  data required time                                                19.98
  data arrival time                                                -19.81
  ------------------------------------------------------------------------
  slack (MET)                                                       0.17
```

< area optimization O >

Figure 6.1 Difference of slack before and after area optimization

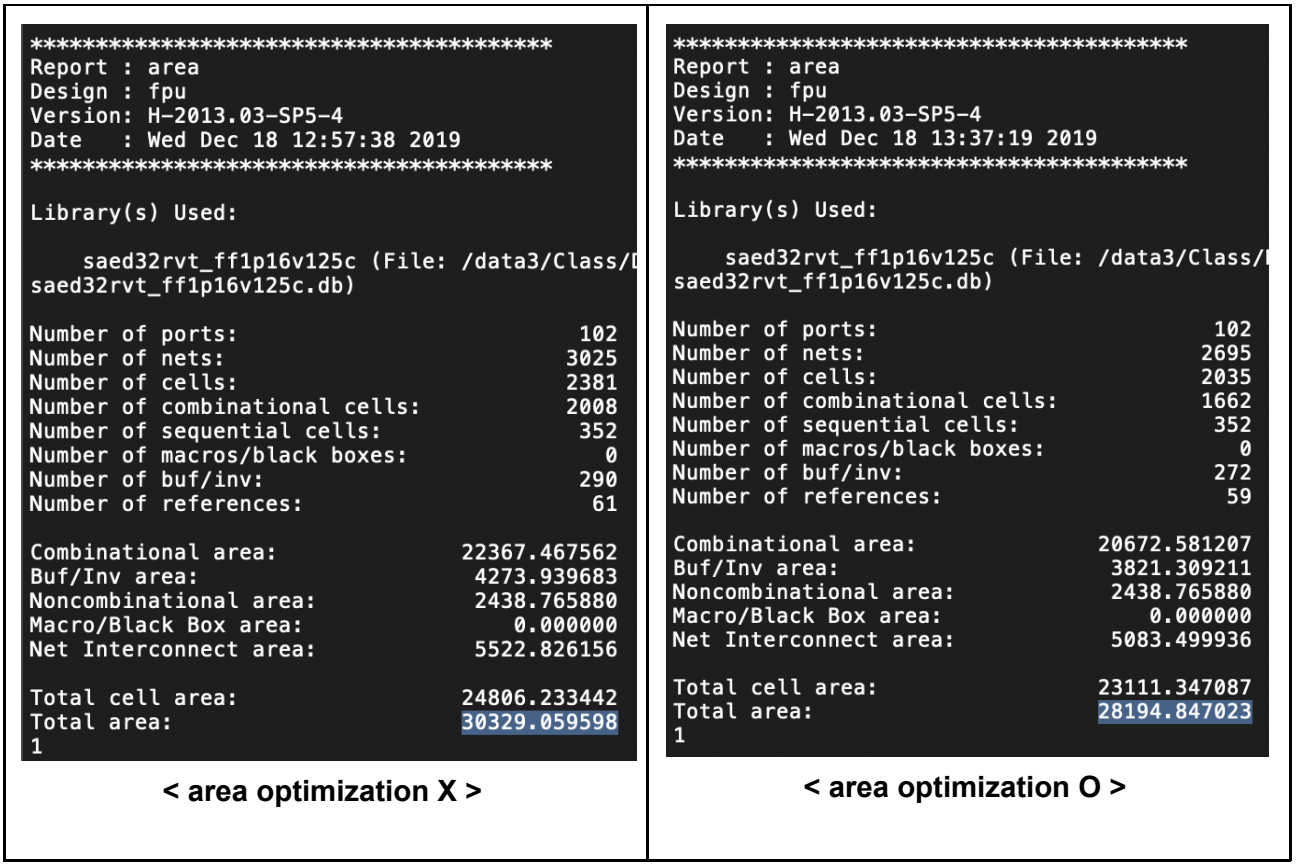


Figure 6.2 Difference of area before and after area optimization

(2) Analysis of the results

Figure 6.1과 6.2는 각각 합성된 fpu에 대한 timing과 area정보를 나타낸다. 일반적인 compile 명령어 또는 compile -area_effort high를 통해 area optimization결과와 일반 결과를 비교하였다.

요구되는 Design constraint에서 area를 맞추기 위해서는, RTL level에서 design 시 gate를 줄이도록 설계하는 것이 중요하다. design compiler는 주어진 코드에 따라 최적화된 gate logic을 합성하지만 결국 user가 설계한 코드를 기반으로 이루어진다. 따라서 fpu를 verilog를 통해 작성 시 불필요한 연산과 latch, bit등을 줄이는 것에 유념해야 한다.

실제로 우리의 팀에서 설계한 design은 설명하기 직관적이고 작성하기 편하도록 만들어졌다. adder와 subtractor의 stage1을 예시로 들면 sel값과 a와 b의 sign을 통해 연속적인 if문으로 작성되었는데 각각의 bit를 xor을 통해 판별한다면 절반으로 감소하여 작성가능하다. 몇 가지 부분에서 이런 식으로 낭비되는 결과가 발생하게 되고, 전체적으로 보았을 시 축적된 결과로 인해 area가 예상보다 큰 결과로 나타나는 것이다.

또한 방법적으로 두 수의 비교하는 연산을 줄이고 bit를 통한 판단이 area감소에 도움이 된다. 예를 들어 항상 16보다 작은 숫자를 가지는 어떤 수 x가 8보다 크거나 같은지, 작은지에 따라 연산이 결정되어 진행된다면 'if

(x[3])’ 으로 대체하여 bit 하나를 통해 판별 가능하다. 이는 timing적으로도 빠르며 비교 연산을 위한 operator가 필요하지 않게 되므로 area 감소에도 도움이 된다.

이러한 부분에 대해 우리의 design에서 round half up 과정에서 area 낭비가 발생되었다고 추측 가능한데, mantissa를 다시 normalize하기 위해서는 앞선 normalize의 과정과 같이 MSB하나만 바라보면 되지만 rounding stage를 만들지 않기 위해 integer와 대소 비교를 하였다. 즉 비교 연산을 수행하기 위한 logic이 합성되므로 area가 증가하였으리라 추측할 수 있다. 하지만 그 결과 우리는 stage를 줄이고 불필요한 register의 선언을 방지할 수 있었는데, 이로 인한 area 감소를 고려해 볼 수 있다. 어떤 방법이 area 감소에 효과적인지는 해당 코드를 작성 후 synthesize를 통해 확인해봐야할 사항인 것이다.

더욱이 floating point에 대해서 adder나 subtractor의 설계과정은 multiplier나 divider에 비해 복잡하다. 후자의 경우 exponent는 더하거나 빼고, mantissa는 곱하거나 나누어주고 normalize를 진행하면 된다. 하지만 전자의 경우 상위 bit(표현되지 않는 mantissa의 최상위 bit, mantissa[23])가 1이 될 때까지 shift가 필요하고 그에 따른 exponent 계산이 필요하기 때문에 복잡할 수 있다. 우리의 디자인은 mantissa의 절반을 나누어 0인지 판별하고 0이 맞지 않다면 다시 절반을 나누어 판별하는 것을 반복하여 shift 시작 위치를 찾아주었다. 이 과정에서 2-input MUX를 가정하여 if - else로 이어지는 구문이 반복되는데 3-input, 4-input등이 효과적인지, 각각의 소자의 area는 어떠한지, 제공하는 라이브러리에 존재하는지 등을 고려하여 if - else if - … - else문으로 이어지는 설계를 통해 효과적인 소자를 이용한다면 area 측면에서 이득을 얻을 것이다.

7. Timing reports with analysis

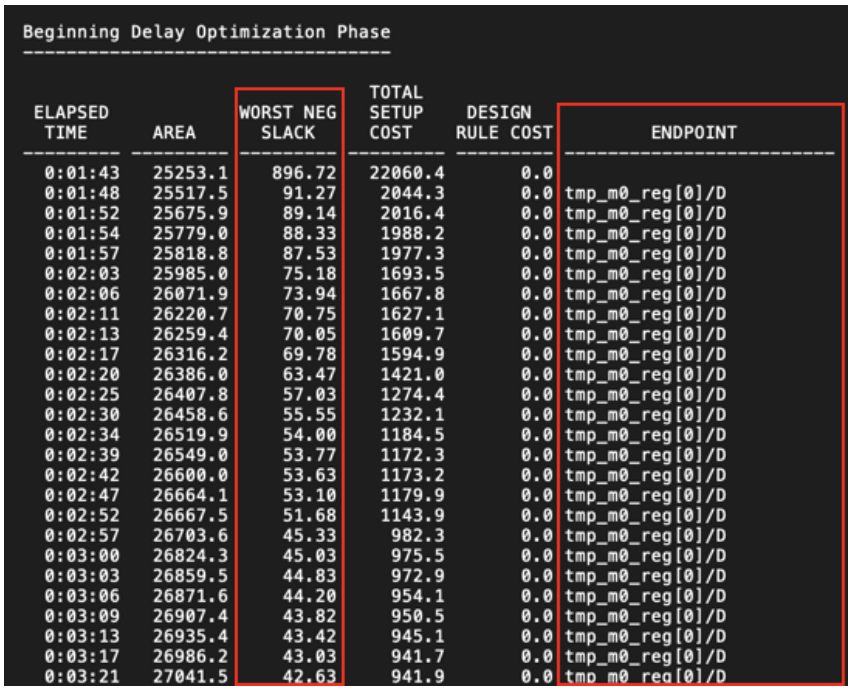
(1) A timing report with 111MHz clock

$ create_clock { "clk" } -name "clk" -period 9 -waveform { 0 4.5 }
$ compile
$ write -format verilog -hier -output ../outputs/fpu_mapped.v

```
****************************************
Report : timing
        -path full
        -delay max
        -max_paths 1
Design : fpu
Version: H-2013.03-SP5-4
Date   : Fri Dec 20 14:16:36 2019
****************************************

Operating Conditions: ff1p16v125c   Library: saed32rvt_ff1p16v125c
Wire Load Model Mode: enclosed

  Startpoint: b_m_reg[1] (rising edge-triggered flip-flop clocked by clk)
  Endpoint: tmp_m0_reg[3]
            (rising edge-triggered flip-flop clocked by clk)
  Path Group: clk
  Path Type: max

  Des/Clust/Port     Wire Load Model       Library
  ------------------------------------------------
  fpu                35000                 saed32rvt_ff1p16v125c
  fpu_DW_div_uns_2   16000                 saed32rvt_ff1p16v125c

  Point                                                   Incr       Path
  --------------------------------------------------------------------------
  clock clk (rise edge)                                   0.00       0.00
  clock network delay (ideal)                             0.00       0.00

                                  ··· 생략 ···


  data arrival time                                                  8.93

  clock clk (rise edge)                                   9.00       9.00
  clock network delay (ideal)                             0.00       9.00
  tmp_m0_reg[3]/CLK (DFFARX1_RVT)                         0.00       9.00 r
  library setup time                                     -0.02       8.98
  data required time                                                 8.98
  --------------------------------------------------------------------------
  data required time                                                 8.98
  data arrival time                                                 -8.93
  --------------------------------------------------------------------------
  slack (MET)                                                        0.05
```

Figure 7.1 Summing of timing report

(2) Analysis of the results



Beginning Delay Optimization Phase

ELAPSED TIME	AREA	WORST NEG SLACK	TOTAL SETUP COST	DESIGN RULE COST	ENDPOINT
0:01:43	25253.1	896.72	22060.4	0.0	
0:01:48	25517.5	91.27	2044.3	0.0	tmp_m0_reg[0]/D
0:01:52	25675.9	89.14	2016.4	0.0	tmp_m0_reg[0]/D
0:01:54	25779.0	88.33	1988.2	0.0	tmp_m0_reg[0]/D
0:01:57	25818.8	87.53	1977.3	0.0	tmp_m0_reg[0]/D
0:02:03	25985.0	75.18	1693.5	0.0	tmp_m0_reg[0]/D
0:02:06	26071.9	73.94	1667.8	0.0	tmp_m0_reg[0]/D
0:02:11	26220.7	70.75	1627.1	0.0	tmp_m0_reg[0]/D
0:02:13	26259.4	70.05	1609.7	0.0	tmp_m0_reg[0]/D
0:02:17	26316.2	69.78	1594.9	0.0	tmp_m0_reg[0]/D
0:02:20	26386.0	63.47	1421.0	0.0	tmp_m0_reg[0]/D
0:02:25	26407.8	57.03	1274.4	0.0	tmp_m0_reg[0]/D
0:02:30	26458.6	55.55	1232.1	0.0	tmp_m0_reg[0]/D
0:02:34	26519.9	54.00	1184.5	0.0	tmp_m0_reg[0]/D
0:02:39	26549.0	53.77	1172.3	0.0	tmp_m0_reg[0]/D
0:02:42	26600.0	53.63	1173.2	0.0	tmp_m0_reg[0]/D
0:02:47	26664.1	53.10	1179.9	0.0	tmp_m0_reg[0]/D
0:02:52	26667.5	51.68	1143.9	0.0	tmp_m0_reg[0]/D
0:02:57	26703.6	45.33	982.3	0.0	tmp_m0_reg[0]/D
0:03:00	26824.3	45.03	975.5	0.0	tmp_m0_reg[0]/D
0:03:03	26859.5	44.83	972.9	0.0	tmp_m0_reg[0]/D
0:03:06	26871.6	44.20	954.1	0.0	tmp_m0_reg[0]/D
0:03:09	26907.4	43.82	950.5	0.0	tmp_m0_reg[0]/D
0:03:13	26935.4	43.42	945.1	0.0	tmp_m0_reg[0]/D
0:03:17	26986.2	43.03	941.7	0.0	tmp_m0_reg[0]/D
0:03:21	27041.5	42.63	941.9	0.0	tmp_m0_reg[0]/D

Figure 7.2 Details of timing report

컴파일 시 로그(compile_log.txt 첨부 파일)를 살펴보면 tmp_m0 레지스터에서 WNS(Worst Negative Slack)이 크게 발생하고 다른 연산에서는 큰 문제가 없었다. tmp_m0는 stage 1에서 각 연산별 mantissa를 계산하고 대입해줄 때 사용하는 레지스터이다.

tmp_m0 레지스터에서 WNS가 크게 나타나는 까닭은 failing path로 인한 문제로 짐작된다. tmp_m0 레지스터는 초기 설정에서 multiple 연산을 위하여 48bit로 설정하였다. 그 결과, 다른 연산에서는 48 bit 전체를 사용하는 것이 아니라 최대 25bit 까지만 사용하기때문에 사용하지 않은 23bit의 false path로 인해 failing path로 인식되어 WNS가 증가한다. 따라서, tmp_m0의 WNS를 감소시켜 구동 clock을 증가시키는 방법은 두 가지로 나누어진다.

첫째, multiple 연산을 위한 기존 48bit의 tmp_m0 레지스터와 별개로 다른 연산을 위한 25bit의 register를 추가한다. 하지만, 이 register을 추가할 경우 WNS의 감소는 기대할 수 있지만, area의 증가는 피할 수 없다.

둘째, tmp_m0 레지스터를 multiple이 아닌 다른 연산에서 사용 될 때는 [47:25] bit는 false path로 constraint를 지정하여 WNS를 감소시킬 수 있다.(‘set_false_path’ command 이용)

추가적으로 area와 timing의 관계는 trade-off이기에 앞서 area report analysis에서 설명한 부분을 토대로 설계하는 것을 통해 보다 빠른 clock에서 동작하는 unit을 설계할 수 있을 것이다.

8. Appendix. 제출 폴더 구성

|_ dsd_final.pdf

|_ code

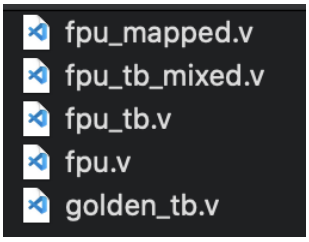
|_ fpu.v

|_ fpu_tb.v - general cases and exception testbench

|_ fpu_tb_mixed.v - mixed general cases testbench

|_ golden_tb.v

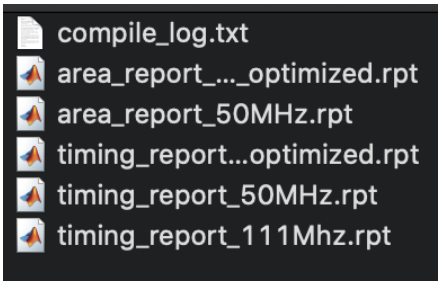
|_ fpu_mapped.v - for the timing report(111Mhz)



|_data

|_ (뽑은 결과들)

|_ compile_log.txt



9. Reference

- Floating point: https://ko.wikibooks.org/wiki/C_%ED%94%84%EB%A1%9C%EA%B7%B8%EB%9E%98%EB%B0%8D_%EC%9E%85%EB%AC%B8/%EB%B6%80%EB%8F%99%EC%86%8C%EC%88%98%ED%98%95_%EB%8D%B0%EC%9D%B4%ED%84%B0

- Single precision & Double precision: https://whatisthenext.tistory.com/146