디지털시스템설계 Project

FPU Design

7 조 강상원

7 조 신민영

목차

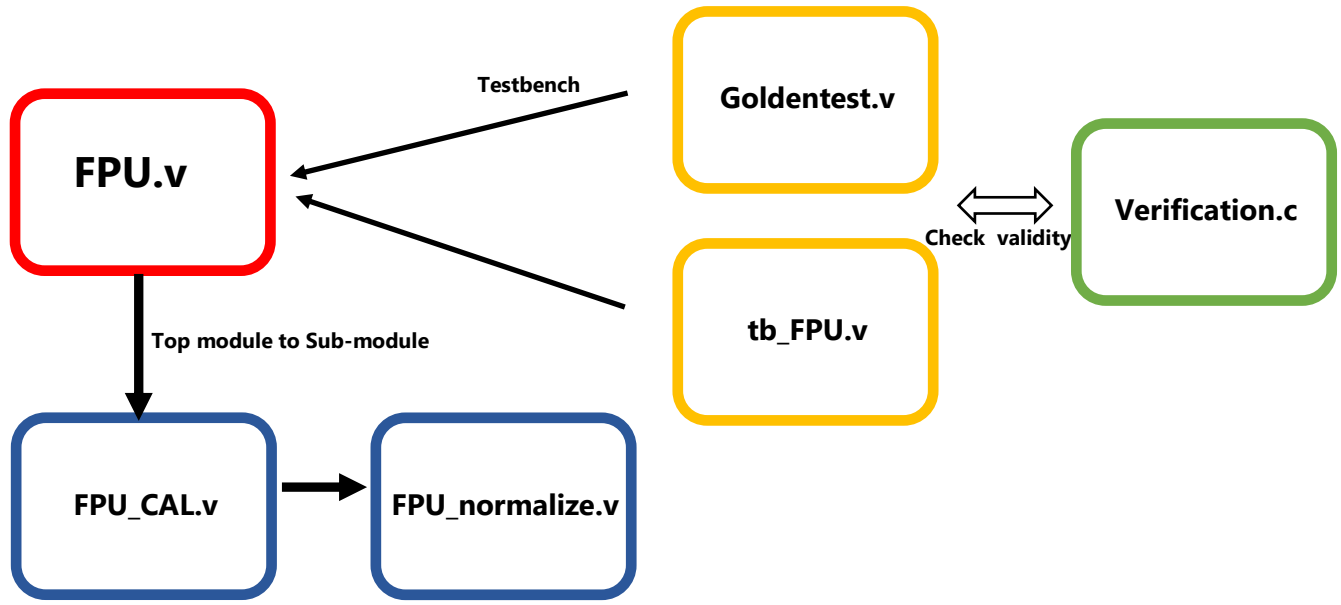
Role Allocation

역할 분배	
강상원	FPU_CAL 의 초안 작성, 사칙연산의 결과가 올바른 값을 내도록 구현, tb_FPU 를 이용하여 verification
신민영	FPU_CAL 의 exception handling, FPU_normalize 를 이용한 clock 분배, golden testbench 를 이용하여 verification

FPU Design and Testbench

1. Code Description for FPU Design

FPU Design and Verification Diagram



FPU.V

```
module FPU(clk, reset, A, B, sel,
round_mode, start, error, over_flow, Y);
input clk, reset;
input [31:0] A, B;
input [1:0] sel;
input round_mode, start;
output reg error, over_flow;
output reg [31:0] Y;

reg [31:0] A_fpu, B_fpu;
reg [1:0] sel_fpu;
reg round_fpu;

wire error_fpu, over_flow_fpu;
wire [31:0] Y_fpu;

FPU_CAL fpu_cal(clk, start, A_fpu, B_fpu,
sel_fpu, round_fpu, error_fpu,
over_flow_fpu, Y_fpu);

always @(posedge clk, negedge reset)
begin
if(~reset) begin
        A_fpu <=0;
        B_fpu <=0;
        sel_fpu <=0;
        round_fpu <=0;
 end
else begin
        if(start) begin
                A_fpu <=A;
                B_fpu <=B;
                sel_fpu <=sel;
                round_fpu <=round_mode;
        end
        else begin
                A_fpu <=A_fpu;
                B_fpu <=B_fpu;
                sel_fpu <=sel_fpu;
                round_fpu <=round_fpu;
                Y <= Y_fpu;
                over_flow <=
over_flow_fpu;
                error <= error_fpu;
        end
end
end
endmodule
```

module FPU

input : A, B, sel, clk, reset. start, round_mode

output : error, over_flow, Y

FPU_CAL module instance

reset -> 0 이면 리셋

start -> 1 일 때 레지스터에 값 저장

저장된 값으로 계산하는 단계

FPU_CAL.V

```
module FPU_CAL (clk, start, A, B, sel,
round_mode, error, over_flow, Y);
input clk, start;
input [31:0] A, B;
input [1:0] sel;
input round_mode;
output reg error,over_flow;
output reg [31:0] Y;

wire S1, S2;
wire [7:0] E1, E2;
wire [22:0] M1, M2;
wire [22:0] result;
wire [4:0] shifted;

reg [24:0] M1_t, M2_t,M_sum;
reg S;
reg [8:0] E;
reg [47:0] M1M2;
reg [47:0] div;
reg [23:0] divider;
reg [24:0] quotient;

FPU_normalize fpu_normalize(M_sum,
result, shifted);

assign S1 = A[31];
assign E1 = A[30:23];
assign M1 = A[22:0];

assign S2 = B[31];
assign E2 = B[30:23];
assign M2 = B[22:0];

parameter
NaN =
32'b0_11111111_10000000000000000000000,
INF =
31'b11111111_00000000000000000000000;

parameter STATE1 = 0,
STATE2 = 1,
STATE3 = 2,
STATE4 = 3;
reg [2:0] next_state;

always @(posedge clk) begin
if(start) begin
error=0;
over_flow=0;
next_state=STATE1;
end
```

module FPU_CAL

-M_sum : 결과 Y의 가수부
-S : 결과 Y의 MSB
-E : 결과 Y의 지수부
-M1M2 : 가수부 곱셈 값
-div : 나눗셈에서 피제수
-divider : 나눗셈에서 제수
-quotient : 나눗셈에서 몫
-module FPU_normalize instanciation

-S1 : A의 MSB
-E1 : A의 지수부
-M1 : A의 가수부
-S2 : B의 MSB
-E2 : B의 지수부
-M2 : B의 가수부

-NaN 값 정의
-INF 값 정의

-clock 분배를 위한 state 정의

-start가 1일 때, 즉 입력 값이 저장될 때 next state = STATE1
-STATE1
next state = STATE2

```
else if(next_state==STATE1) begin
next_state=STATE2;
case (sel)
2'b00, 2'b01 : begin
if ((A[30:0]==INF && B[30:0]==INF) &&
(A[31]^B[31]^sel[0])) begin
Y = NaN;
error = 1;
next_state=STATE4;
end
else begin
M1_t={2'b01,M1};
M2_t={2'b01,M2};
if (E1 > E2) begin
M2_t = M2_t >> E1-E2;
E = {1'b0,E1};
end
else begin
M1_t = M1_t >> E2-E1;
E = {1'b0,E2};
end
if (M1_t>M2_t) begin
S=S1;
if (S1^S2^sel[0])
M_sum = M1_t-M2_t;
else
M_sum = M1_t+M2_t;
end
else begin
if (S1^S2^sel[0]) begin
M_sum = M2_t-M1_t;
S = 1-S1;
end
else begin
M_sum = M2_t+M1_t;
S = S1;
end
end
end
end
2'b10 : begin
if ((A[30:23] == 8'b0 && B[30:0]==INF) ||
(A[30:0] == INF && B[30:23]==8'b0)) begin
Y = NaN;
error = 1;
next_state=STATE4;
end
else if ((A[30:0] == INF) || (B[30:0] ==
INF)) begin
Y={S1^S2,INF};
over_flow=1;
next_state=STATE4;
end
else if ((A[30:23] == 8'b0) || (B[30:23]
== 8'b0)) begin
Y={S1^S2,31'b0};
```

-덧셈, 뺄셈의 case

(INF – INF)이 되는 경우 NaN
next state = STATE4

-NaN이 아닌 경우
가수부에서 생략된 1과 overflow를 막기 위해 2‘b01 추가
-E1>E2인 경우 그 차이만큼 B의 가수부 비트 이동 (지수부를 맞춰주기 위함)
overflow를 막기 위한 1’b0 추가
-E1<E2인 경우 차이만큼 A의 가수부 비트 이동
-A의 가수부가 더 큰 경우 Y의 MSB는 덧셈 뺄셈에 관계없이 S1

-S1^S2^sel[0] 값이 1이면 가수부는 뺄셈
-S1^S2^sel[0] 값이 0이면 가수부는 덧셈

-B의 가수부가 더 큰 경우
-S1^S2^sel[0] 값이 1이면
가수부는 뺄셈
Y의 MSB는 S1의 반대
-S1^S2^sel[0] 값이 0이면
가수부는 덧셈
Y의 MSB는 S1

-곱셈의 case
(0 x INF)의 경우 NaN

next state -> STATE4

-NaN이 아니면서 하나의 값이 INF인 경우
Y = INF

next state -> STATE4

-NaN이 아니면서 하나의 값이 0인 경우
Y=0

```
next_state=STATE4;
end
else if (E1+E2<9'b0_0111_1111) begin
Y={S1^S2,31'b0};
next_state=STATE4;
end
else begin
E=E1+E2-8'b0111_1111;
M1M2={1'b1,M1}*{1'b1,M2};
end
end
2'b11 : begin
if ((A[30:0]==INF) && (B[30:0]==INF))
begin
error = 1;
Y=NaN;
next_state=STATE4;
end
else if ((A[30:23]==8'b0) &&
(B[30:23]==8'b0)) begin
error = 1;
Y=NaN;
next_state=STATE4;
end
else if (B[30:23]==8'b0) begin
Y={S1^S2,INF};
over_flow=1;
next_state=STATE4;
end
else if (E1<=E2-8'b0111_1111) begin
Y={S1^S2,31'b0};
next_state=STATE4;
end
else begin
div={1'b1,M1,24'b0};
divider = {1'b1,M2};
E=E1-E2+8'b0111_1111;
end
end
endcase
end
else if(next_state==STATE2) begin
next_state=STATE3;
case (sel)
if (M_sum == 25'b0) begin
Y=0;
next_state = STATE4;
end
else if (E < shifted) begin
Y=0;
next_state = STATE4;
end
else begin
E = E - shifted + 1;
end
2'b10 : begin
```

next state -> STATE4

-E1+E2 값이 127이 두 번 더해지므로 한번 빼주어야 하는데 이때 0 이하가 되는 경우 Y=0,
next state -> STATE4
-그 외의 경우 127을 한번 빼줌
M1M2에 가수부 곱 저장

-나눗셈의 case
-INF / INF 면 Y=NaN

next state -> STATE4

-0 / 0 이면 Y=NaN

next state -> STATE4

-이 외의 경우에 제수가 0이면 Y=INF

next state -> STATE4

-지수부의 뺄셈 후 127 한 번 더한 값이 0보다 작은 경우 Y = 0
next state -> STATE4
-그 외의 경우
소수아래 23bit도 몫으로 만들기 위해 피제수에 24'b0 추가, 생략된 1 추가
지수부의 뺄셈 후 127 한번 더함

-STATE2
next state -> STATE3
-덧셈, 뺄셈의 case
-가수부의 결과가 0인 경우
Y=0;
next state -> STATE4
-빼야하는 shifted가 E보다 큰 경우
Y=0;
next state -> STATE4

-가수부의 결과가 0이 아닌 경우
FPU_normalized의 shifted 값만큼 뺌
(가수부가 1.xxx가 되도록 하는 과정)

```
if (M1M2[47]==1) begin
E=E+1'b1;
end
else begin
M1M2=M1M2 << 1;
end
end
2'b11 : begin
if (M2>M1)begin
E=E-1'b1;
quotient={div/divider,1'b0};
end
else begin
E=E;
quotient=div/divider;
end
end
endcase
end
else if(next_state==STATE3) begin
case (sel)
2'b00, 2'b01 : begin
if (E >= 9'b0_1111_1111 ) begin
over_flow=1;
Y = (S==0) ? {1'b0,INF} : {1'b1,INF};
end
else if
({M_sum[15:14],round_mode}==3'b110)
Y={S,E[7:0],result[22:15]+1'b1,15'b0};
else
Y={S,E[7:0],result[22:15],15'b0};
end
2'b10 : begin
if (E >= 9'b0_1111_1111) begin
Y={S1^S2,8'b1111_1111,23'b0};
over_flow = 1;
end
else if (E == 9'b0) begin
Y={S1^S2,31'b0};
end
else if
({M1M2[39:38],round_mode}==3'b110) begin
Y={S1^S2,E[7:0],M1M2[46:39]+1'b1,15'b0};
end
else
Y={S1^S2,E[7:0],M1M2[46:39],15'b0};
end
2'b11 : begin
if (E >= 9'b0_1111_1111) begin
Y={S1^S2,8'b1111_1111,23'b0};
over_flow = 1;
end
else if (E == 9'b0) begin
Y={S1^S2,31'b0};
end
```

-곱셈의 case
A,B의 곱셈의 결과 가수부가 1x.xxxx가 된 경우 E에 +1 하여 1.xxxx로 맞춰줌
-곱셈 결과 01.xxx가 된 경우 1.xxxx로 맞춰주기 위해 M1M2 한 칸 쉬프트
-나눗셈의 case
-M2>M1이면 몫이 0.1xxx이기 때문에 연산 후에 1.xxx로 바꿔줌

-M1>=M2의 경우 단순히 나눗셈 연산
몫은 quotient에 저장

-STATE3

-덧셈, 뺄셈의 case
-E가 8'b1111_1111이상인 경우 overflow
MSB값에 따라 +-INF

-가수부의 9번째 bit에서 라운딩
-rounding ties to even이면서 8,9번째 bit가 11이면 +1
-rounding down이면 9번째 이하 비트 버림
곱셈의 case
-E가 8'b1111_1111 이상이면 overflow

-계산 결과 denormalized 수 일 경우
Y=0

-가수부의 9번째 bit에서 라운딩

-버림 방식으로 라운딩

-나눗셈의 case
E가 8'b1111_1111 이상이면 overflow

-계산 결과 denormalized 수 일 경우
Y=0

```
else if
({quotient[16:15],round_mode}==3'b110)
Y={S1^S2,E[7:0],quotient[23:16]+1'b1,15'b
0};
else
Y={S1^S2,E[7:0],quotient[23:16],15'b0};
end
endcase
end
else begin
Y = Y;
end
end
endmodule
```

-가수부의 9번째 비트에서 라운딩

- 버림 방식으로 라운딩

-STATE4
그대로 출력값 유지

FPU_normalize.V

```
module FPU_normalize (
    input [24:0] src,
    output [22:0] result,
    output [4:0] shifted
);

wire [24:0] layer_0 = src;
wire [24:0] layer_1 = layer_0[24:9]  ==
0 ? { layer_0[8:0], 16'd0 } : layer_0;
wire [24:0] layer_2 = layer_1[24:17] ==
0 ? { layer_1[16:0], 8'd0 } : layer_1;
wire [24:0] layer_3 = layer_2[24:21] ==
0 ? { layer_2[20:0], 4'd0 } : layer_2;
wire [24:0] layer_4 = layer_3[24:23] ==
0 ? { layer_3[22:0], 2'd0 } : layer_3;
wire [22:0] layer_5 = layer_4[24]    ==
0 ? layer_4[22:0] : layer_4[23:1];

assign result = layer_5;

assign shifted = { layer_0[24:9] == 0,
layer_1[24:17] == 0, layer_2[24:21] == 0,
layer_3[24:23] == 0, layer_4[24] == 0 };

endmodule
```

module FPU_normalize
덧셈 뺄셈 경우에 가수부가 1.xxx가 되도록 shift를 하기 위함

src에는 FPU_CAL의 M_sum이 들어감
상위 16개의 bit가 0이면 16bit 쉬프트
-> 상위 16bit 안에 1 존재
상위 8개의 bit가 0이면 8bit 쉬프트
-> 상위 8bit 안에 1 존재
상위 4개의 bit가 0이면 4bit 쉬프트
-> 상위 4bit 안에 1 존재
상위 2개의 bit가 0이면 2bit 쉬프트
-> 상위 2bit 안에 1 존재
최상위 bit가 0이면 1bit 쉬프트
-> 최상위 bit가 1

result = 최상위 bit가 1인 23bit 수

위에서 각 layer 별로 쉬프트 되는 수의 합

2. Code Description for Testbench

tb_FPU.V

```
`timescale 1ns/1ns
module tb_FPU;
reg Clock, Reset;
reg [31:0] A, B;
reg [1:0] Sel;
reg round;
reg start;
wire [31:0] Y;
wire Overflow, Error;

FPU FPU(Clock,Reset,A,B,Sel,round, start,
Error,Overflow,Y);

initial
        $monitor("A = %b, B = %b, Sel
= %b, round = %b, Y = %b, Overflow = %b,
Error = %b", A, B, Sel, round, Y,
Overflow, Error);

initial
begin
        Clock = 1'b0;
        forever #2 Clock = ~Clock;
end

initial begin
Reset=1;
#3
Reset=0;
#2
Reset=1;
        A=32'b0_00010010_0110101010010100
0000000;
B=32'b0_00001110_0111000100100100000000;
Sel = 2'b00;
start =1;
round =0;
#2
start=0;
#30
        A=32'b0_00010010_0110101010010100
0000000;
B=32'b0_00001110_0111000100100100000000;
Sel = 2'b00;
start =1;
round =1;
#2
start=0;
#30
```

시간단위 1ns, 해상도 1ns
module tb_FPU

입출력 값이 바뀔 때마다 monitor

Clock 주기 4ns

Reset 초기화

일반적인 덧셈 확인
Y = 0_00010010_10000001101001100100000
에서 round=0 으로 9번째 비트에서 반올림 되어서
Y = 0_00010010_10000010000000000000000
이 되어야함

위와 같은 값에서 round = 1로 반올림 없이 버려지는 경우
Y = 0_00010010_10000001000000000000000
이 되어야함

A=32'b0_00010010_0110101010010
1000000000;
B=32'b1_00001110_0111000100100100000
00;
Sel = 2'b00;
start =1;
round =0;
#2
start=0;
#30
	B의 sign이 음수라 결과적으로 뺄셈이 되는 경우
	Y = 0_00010010_01010100000000000000000 이 되어야함

A=32'b0_11111111_0000000000000
0000000000;
B=32'b0_00001110_0111000100100100000
00;
Sel = 2'b00;
start =1;
round =0;
#2
start=0;
#30
	A 값이 INF인 경우 Y = INF가 되어야함

A=32'b0_11111111_0000000000000
0000000000;
B=32'b1_11111111_0000000000000000000
00;
Sel = 2'b00;
start =1;
round =0;
#2
start=0;
#30
	(+INF)+(-INF) 구조로 Y = NaN이 되어야함

A=32'b0_11111110_1110000000000
0000000000;
B=32'b0_11111110_1110000000000000000
00;
Sel = 2'b00;
start =1;
round =0;
#2
start=0;
#30
	A, B의 합이 overflow가 나는 경우로 Y = INF

A=32'b0_00000001_0000111000000
0000000000;
B=32'b1_00000001_0000010110000000000
00;
Sel = 2'b00;
start =1;
round =0;
#2
start=0;
#30
	A+B의 결과가 denormalized number인 경우 Y=0 이 나와야함

A=32'b0_00010010_0110101010010 1000000000; B=32'b1_00001110_0111000100100100000 00; Sel = 2'b01;	일반적인 뺄셈 확인 Y=0_00010010_10000010000000000000000 이 되어야함

```
start =1;
round =0;
#2
start=0;
#30
        A=32'b0_01010010_01100000000000000000000;
B=32'b0_01001110_01110000000000000000000;
Sel = 2'b10;
start =1;
round =0;
#2
start=0;
#30

A=32'b0_00010010_01100000000000000000000;
B=32'b0_00001110_01110000000000000000000;
Sel = 2'b10;
start =1;
round =0;
#2
start=0;
#30
        A=32'b0_00010010_01100000000000000000000;
B=32'b0_00000000_00000000000000000000000;
Sel = 2'b10;
start =1;
round =0;
#2
start=0;
#30
        A=32'b0_11111111_00000000000000000000000;
B=32'b0_00001110_01110000000000000000000;
Sel = 2'b10;
start =1;
round =0;
#2
start=0;
#30
        A=32'b0_11111111_00000000000000000000000;
B=32'b0_00000000_00000000000000000000000;
Sel = 2'b10;
start =1;
round =0;
#2
start=0;
```

일반적인 곱셈 확인
Y=0_00100001_11111010000000000000000
이 되어야함

곱셈 결과 denormalized 수가 되는 경우
Y=0 이 되어야함

B가 0인 경우
Y=0 이 되어야함

A가 INF이고, B != 0인 경우
Y= INF 가 되어야함

INF x 0 의 경우
Y = NaN 이 되어야함

```
#30
        A=32'b0_11111110_00000000000000000000000;
B=32'b0_10000000_00000000000000000000000;
Sel = 2'b10;
start =1;
round =0;
#2
start=0;
#30
        A=32'b0_00001100_01000000000000000000000;
B=32'b0_10000110_01000000000000000000000;
Sel = 2'b11;
start =1;
round =0;
#2
start=0;
#30
        A=32'b0_00001100_01000000000000000000000;
B=32'b0_00000000_00000000000000000000000;
Sel = 2'b11;
start =1;
round =0;
#2
start=0;
#30
        A=32'b0_11111111_00000000000000000000000;
B=32'b0_00000000_00000000000000000000000;
Sel = 2'b11;
start =1;
round =0;
#2
start=0;
#30

A=32'b0_11111111_00000000000000000000000;
B=32'b0_11111111_00000000000000000000000;
Sel = 2'b11;
start =1;
round =0;
#2
start=0;
#30

A=32'b0_00000000_00000000000000000000000;
```

곱셈 결과로 지수부가 11111111이 되는 경우
Y = INF 가 되어야함

일반적인 나눗셈
Y = 0_00000101_00000000000000000000000
이 되어야함

제수가 0인 경우
Y = INF 가 되어야함

피제수가 INF, 제수가 0인 경우
Y = INF 가 되어야함

INF/INF 의 경우
Y = NaN 이 되어야함

0/0의 경우
Y = NaN 이 되어야함

```
B=32'b0_00000000_00000000000000000000
00;
Sel = 2'b11;
start =1;
round =0;
#2
start=0;
#30

A=32'b0_00001100_01000000000000000000
00;
B=32'b0_11111111_00000000000000000000
00;
Sel = 2'b11;
start =1;
round =0;
#2
start=0;
#30
        A=32'b0_00000001_0100000000000
0000000000;
B=32'b0_10000000_00000000000000000000
00;
Sel = 2'b11;
start =1;
round =0;
#2
start=0;
#30
$stop;
end

endmodule
```

제수가 INF 인 경우
Y = 0 이 되어야함

나눗셈 결과 denormalized 수가 되는 경우
Y = 0 이 되어야함

3. Code Description for Verification.c

Verification.c

```c
#include <stdio.h>

float bintofloat(unsigned int x) {
        union {
                unsigned int  x;
                float  f;
        } temp;
        temp.x = x;
        return temp.f;
}

unsigned int floattobin(float f) {
        union {
                unsigned int  x;
                float  f;
        } temp;
        temp.f = f;
        return temp.x;
}
int main() {
        unsigned int A[] ={};
        unsigned int B[] ={};
        unsigned int Sel[] = {};
        unsigned int round[62] = {};

for (int i = 0; i < 62; i++) {
     float result;
     printf("%d 번째 연산: %f ", i,
bintofloat(A[i]));
      switch (Sel[i]) {
                case 0:
                        result =
bintofloat(A[i]) + bintofloat(B[i]);
                        printf("+ ");
                        break;
                case 1:
                        result =
bintofloat(A[i]) - bintofloat(B[i]);
                        printf("- ");
                        break;
                case 2:
                        result =
bintofloat(A[i]) * bintofloat(B[i]);
                        printf("x ");
                        break;
                case 3:
                        result =
bintofloat(A[i]) / bintofloat(B[i]);
                        printf("/ ");
```

tb_FPU 혹은 Goldentest 등 testbench 결과를 검증하기 위한 C code file

-bintofloat

32bit 의 값을 float 타입으로 반환

-floattobin

float 타입을 32bit 바이너리로 반환

-Input A, B, Sel, round

testbench 에서 입력한 순서대로 배열에 값을 넣어주어야 동작함.

-For loop

입력값을 모두 연산하는 Loop. 62 번의 iteration 을 하는 이유는 Goldentest 의 입력이 62 개이기 때문이다. testbench 에 따라 변경 가능.

-ADD

-SUB

-MUL

-DIV

```
                        break;
                default:
                        printf("something
wrong\n");
                }
                unsigned int temp =
floattobin(result);
                if ((((temp >> 14) & 3)
== 3) && (round[i] == 0))
                        temp = (temp + (1
<< 14));

                temp = temp & 0xffff8000;
                result =
bintofloat(temp);
                printf("%f = %f --
>16 진수로는 0x%x\n", bintofloat(B[i]),
result, floattobin(result));
        }
        return 0;
}
```

-Rounding ties to even 일 경우

2^14 의 값을 더해줌.

-소수점 아래 8 자리까지만 Masking on

-binary result 를 float type result 로 변환 후 출력

Testbench results

1. Expected results for the testbench

Testbench 를 실행하기 앞서서, 예상되는 결과를 Verification.c 를 통해서 출력해보는 과정이 필요하다. Testbench 를 통해서 waveform 을 확인할 수 있지만, 곱셈과 나눗셈 등의 연산을 한 뒤 Rounding Rule 을 적용하는 FPU 의 경우 waveform 만을 보고 값이 제대로 나왔는지 한 눈에 확인하는 것이 쉽지 않기 때문이다. Verification.c 의 Code description 은 위의 FPU Design and Verification 섹션에서 설명하고 있다. 우리가 작성한 Testbench 인 tb_FPU.v 은 21 개의 서로 다른 입력 값에 대하여 검증하고 있다. 21 개의 입력 A, B 와 round, sel 을 Verification.c 에 입력하고 실행하면 결과를 확인할 수 있다. 이 때, 주의할 것이 두 가지 있다. 첫 번째, C 언어는 floating point 의 값이 NaN 일 때 이를 항상 -NaN(0xffc00000)로 표현한다. exponent bit 과 sign bit 을 모두 1 로 표시하기 때문이다. 하지만 실제로 -NaN 일 필요일 없는 것에 대해서도 마이너스 부호를 붙이기 때문에 Verification.c 를 통해서 나오는 -NaN(0xffc00000)과 NaN(0x7fc00000)은 모두 NaN(0x7fc00000) 로 바꾸어서 표에 정리하였다.

두 번째, Modelsim 은 Radix 가 float 32 일 때 decimal 기준의 소수점 아래 5 자리까지만 출력을 한다. C 언어는 float 32 는 소수점 아래 6 자리까지만 디폴트로 출력을 하고 있다. 따라서 우리는 이러한 프로그램 자체 Rounding Policy 에 의한 혼란을 막고자 Hex 값을 같이 표기한다. 그 결과를 Table 1 에 정리하였다. 이것을 실제 Testbench 를 통해서 나온 결과인 Table 2 와 비교하여, 우리의 FPU design 을 검증하는 과정을 거쳤다.

Table 1. float32 and Hex values for the expected tb_FPU.v generated by Verification.c

		Iteration																				
		1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21
Radix	float 32	0	0	0	inf	NaN	inf	0	0	0	0	0	inf	NaN	inf	0	inf	inf	NaN	NaN	0	0
	HEX	**41000000**	**40800000**	**2a000000**	**7f800000**	**7fc00000**	**7f800000**	**58000000**	**41000000**	**10fd0000**	**0**	**0**	**7f800000**	**7fc00000**	**7f800000**	**28000000**	**7f800000**	**7f800000**	**7fc00000**	**7fc00000**	**0**	**50000000**

2. Waveform Simulation Results

figure 1 부터 *figure 4* 까지 총 5+ 5 + 5 + 6 회의 iteration 을 진행하였다. figure 의 /tb_FPU/FPU/Y 행을 보면 출력된 FPU 의 Y 값을 확인할 수 있고, Table 2 에는 이 값이 Table 1 과 일치하는 지에 대한 여부가 표시되어 있다. 결론적으로 말하자면, Table 1 과 Table 2 의 Y 값은 모두 일치한다. 이것은 Hex 행을 비교해보면 빠르게 알 수 있다. 따라서 Table 2 의 Verifiaction 행은 모두 'O'로 표시되었다.

각각의 Iteration 에서 Verify 하려했던 했던 목적은 Table 2 를 통해 알 수 있다. Iteration 1 은 일반 덧셈을 검증하고 있다. Iteration 2 는 Round down 을 검증하고 나머지의 경우는 모두 Rounding ties to even 에 대하여 검증하고 있다. Iteration 3, 8 은 일반 뺄셈을 검증한다. Iteration 9, 15 는 각각 곱셈과 나눗셈에 대하여 검증하고 있다. Iteration 4 와 6, 12, 14, 16, 17 은 모두 무한 발생 조건을 검증하는데 각기 다른 무한 발생 조건에 대해 고려하고 있다. Iteration 5, 13, 18, 19 는 NaN 발생 조건을 검증하고 있다. Iteration 7, 10, 11, 20, 21 은 Zero 발생 조건을 검증하고 있다. Zero 발생 조건에 denormalized number 가 포함되어 있다. 사칙 연산을 통해서 출력되는 값의 지수부가 0 이이고, fraction bit 이 0 이 아닐 때, 이 숫자는 denormalized number 이다. 이 값을 모두 0 으로 처리하라는 design policy 에 의해서 우리의 FPU 는 denormalized number 는 모두 0 으로 처리하고 있고, 그에 대한 검증을 iteration 7, 10, 21 에서 진행하고 있다.

Table 2. test purpose for the designed FPU

Verification Purpose		Iteration #
ADD		1
SUB		3, 8
MUL		9
DIV		15
Rounding Rules	**Rounding down**	2
	Rounding ties to even	All except iteration #2
NaN 발생 조건	**∞ + (-∞)**	5
	0 x ∞	13
	0 ÷ 0	19
	∞ ÷ ∞	18
INF 발생 조건	**INF + 상수**	4
	INF x 상수	12
	상수 + 상수(overflow)	6

	상수 x 상수(overflow)	14
	상수 ÷ 0	16
	INF ÷ 0	17
Zero 발생 조건	**상수 x 0**	11
	상수 / INF	20
	상수 + 상수 = denormalized	7
	상수 x 상수 = denormalized	10
	상수÷상수 = denormalized	21

Y 값은 Verification.c 와 testbench 의 Y 출력 값을 비교해봄으로써 검증하였다. Verification.c 에서는 error 및 overflow 에 대한 예상 출력 값을 나타내고 있지 않기 때문에, error 와 overflow 에 대해서는 waveform 을 통해서 직접 확인해 보아야 한다.

먼저, error 는 NaN 발생 조건과 동일하다. 출력값으로 NaN 이 나온다면, error 가 1 이 되어야 하고, NaN 이 나오지 않는다면, error 는 0 으로 표시되어야 한다. 따라서 우리의 Iteration 중 NaN 의 발생을 목적으로 하였던 Iteration 5, 13, 18, 19 에서 error 가 1 이 되고 나머지는 0 이 될 것으로 기대한다. *figure 1* 부터 *figure 4* 를 보면, NaN 의 발생과 함께 error 가 발생하고, 그 외에는 모두 0 으로 표시되는 것을 확인할 수 있다.

overflow 는 waveform 에서 error 행 밑에서 볼 수 있다. overflow 는 0 으로 값이 나누어지거나, 덧셈과 곱셈 과정에 의해서 INF 가 출력될 경우 1 로 나타난다면 정상적으로 동작하는 것이다. 우리의 testbench 에서는 위에 해당하는 경우는 모두 INF 가 출력되는 상황이므로, Y 출력값이 INF 일 때, overflow 가 나오는 지를 확인하면 된다. iteration 4, 12, 6, 14, 16, 17 에서 우리는 overflow 가 나올 것으로 기대할 수 있고, figure 을 통해서 overflow 가 모두 올바르게 출력되고 있는 것을 확인하였다.

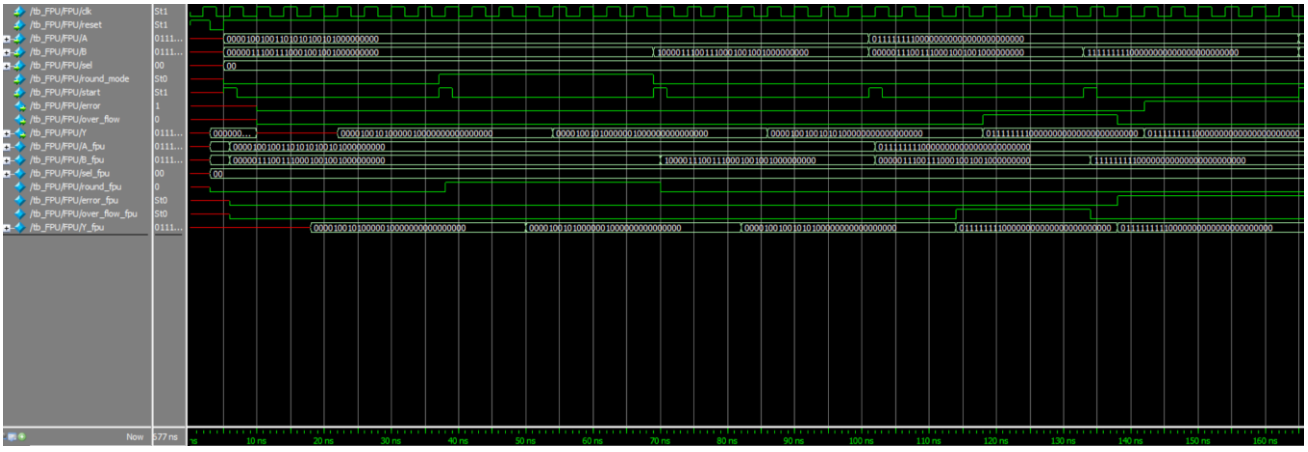


figure 1. Waveform results from 0ns-165ns(5 iteration)

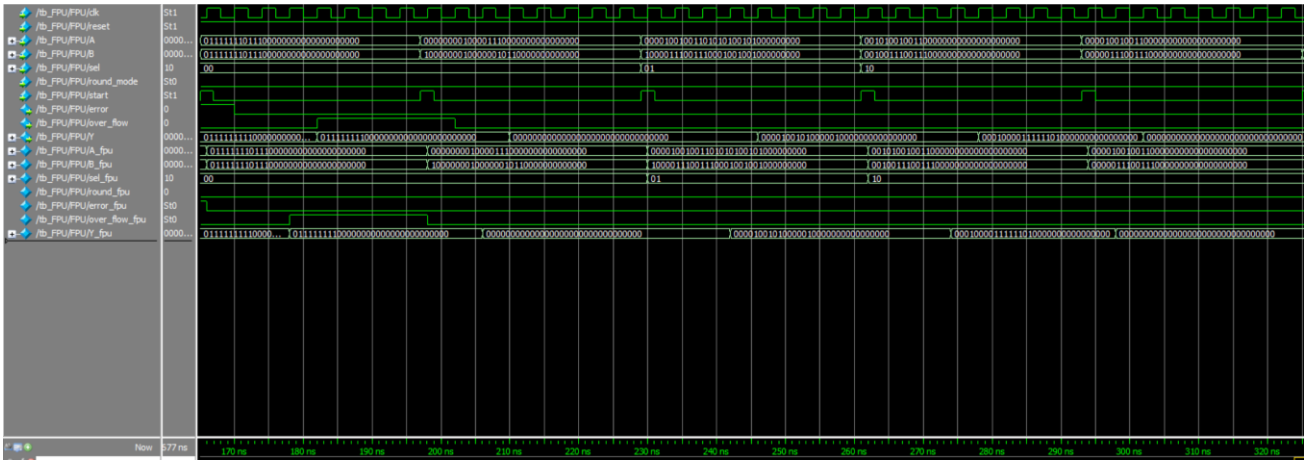


figure 2. . Waveform results from 165ns-325ns(5 iteration)

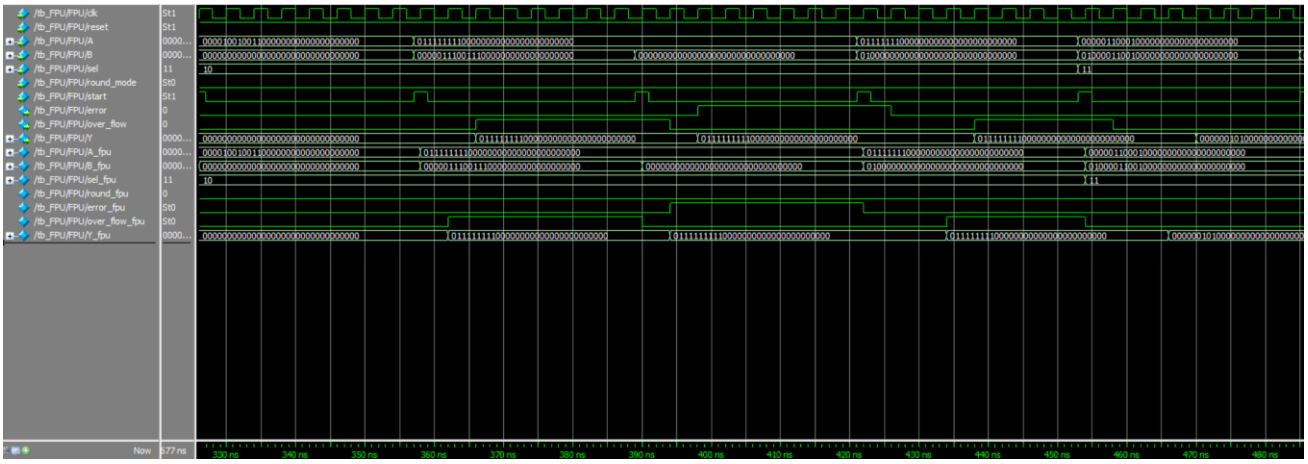


figure 3. Waveform results from 325ns-485ns(5 iteration)

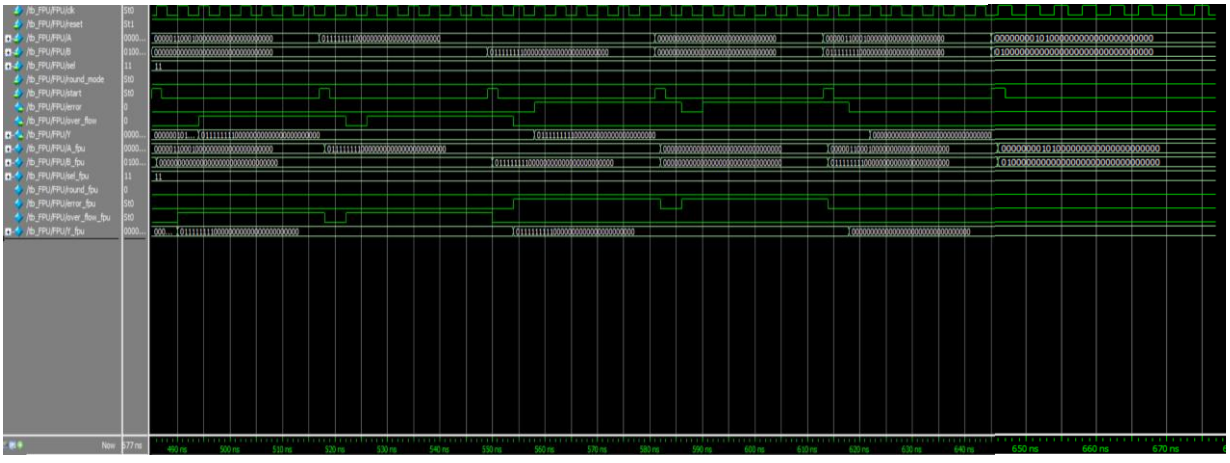


figure 4. Waveform results from 485ns-675ns(6 iteration)

Table 3. Results table from the waveform. Purpose flag 'A'= Add operate. 'S' = Subtract operate. 'M' =Multiply operate. 'D' = Divide operate. 'R' = Rounding policy. 'I' = Infinite condition. 'N' =NaN condition. 'Z' = Zero condition

		Iteration																				
		1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21
Radix	float32	2.32315e-33	2.31713e-33	2.04630e-33	inf	NaN	inf	0	2.32315e-33	9.97909e-29	0	0	inf	NaN	inf	1.88079e-37	inf	inf	NaN	NaN	0	0
	HEX	**941000 00**	**940800 00**	**92a000 00**	**7f8000 00**	**7fc000 00**	**7f8000 00**	**58000**	**941000 00**	**10fd000**	**0**	**0**	**7f8000 00**	**7fc000 00**	**7f8000 00**	**280000 0**	**7f8000 00**	**7f8000 00**	**7fc000 00**	**7fc000 00**	**0**	**500000 0**
Purpose		A	R	S	I	N	I	Z	S	M	Z	Z	I	N	I	D	I	I	N	N	Z	Z
Verification		O	O	O	O	O	O	O	O	O	O	O	O	O	O	O	O	O	O	O	O	O

Table 3 은 Y 출력값에 대하여 Functional Correctness 를 정리하였다. Verification.c 를 통해서 예상했던 Y 값과 일치할 경우 Verification 행에 'O'가 표시된다. 결과적으로 모두 Verified 되었고 앞선 overflow 와 error 도 figure 의 waveform 을 통해서 확인할 수 있었다.

3. Expected results for the Golden testbench

주어진 골든 테스트 벤치에 대해서도 동일하게 Verification.c 를 통해서 값을 예상할 수 있다. Verification.c 를 통해 값을 얻어야만 Golden testbench 를 실행하였을 때 얻는 waveform 의 성과를 평가할 수 있다. 각각의 프로그램 상의 Rounding Policy 에 따라서 보이는 값이 달리 보일 수 있으므로, Hex 값을 같이 표기한다.

Table 4. Expected results according to the iteration 1-21

		Iteration																				
		1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21
Radix	float32	360	120	28800	-2	6.03125	6.015625	inf	-inf	0	-0	NaN	NaN	NaN	249	124	63	95	-97	-11	-249	-273
	HEX	**43b400002**	**42f00000**	**46e10000**	**c0000000**	**40c10000**	**40c08000**	**7f800000**	**ff800000**	**0**	**80000000**	**7fc00000**	**7fc00000**	**7fc00000**	**43790000**	**42f80000**	**427c0000**	**42be0000**	**c2c20000**	**c1300000**	**c3790000**	**c3888000**

Table 5. Expected results according to the iteration 22-42

		Iteration																				
		22	23	24	25	26	27	28	29	30	31	32	33	34	35	36	37	38	39	40	41	42
Radix	float32	-141	-33	9	-15	-113	29	130	-191	159	-157	43	-9	15	113	-29	1376	360	22	-1376	-360	-22
	HEX	**c30d0000**	**c2040000**	**41100000**	**c1700000**	**c2e20000**	**41e80000**	**43020000**	**c33f0000**	**431f0000**	**c31d0000**	**422c0000**	**c1100000**	**41700000**	**42e20000**	**c1e80000**	**44ac0000**	**43b40000**	**41b00000**	**c4ac0000**	**c3b40000**	**c1b00000**

Table 6. Expected results according to the iteration 43-61

		Iteration																		
		43	44	45	46	47	48	49	50	51	52	53	54	55	56	57	58	59	60	61
Radix	float32	1376	360	22	5.625625	4.75	0.383789	0.012024	-5.65625	-4.75	-0.383789	-0.012024	-5.65625	-4.7	-0.383789	-0.012024	5.65625	4.75	0.383789	0.012024
	HEX	**44ac0000**	**43b40000**	**41b00000**	**40b50000**	**40980000**	**3ec48000**	**3c450000**	**c0b50000**	**c0980000**	**bec48000**	**bc450000**	**c0b50000**	**c0980000**	**bec48000**	**bc450000**	**40b50000**	**40980000**	**3ec48000**	**3c450000**

4. Waveform Simulation Results for Golden testbench

이번 프로젝트에서 요구하는 Golden testbench 의 구간은 0ns-600ns 이다. 따라서 아래 figure 중에서 *figure 5, figure 6* 가 해당하는 구간이다. 나머지 구간은 우리의 Verification 을 더욱 확실히 하고자 첨부하였다. Waveform 을 분석해보면, Reset 신호가 Negative edge 로 활성화된 이후부터 계산이 시작된다. Reset 신호가 들어오기 전에 A 와 B 는 0 으로 초기화되어 있는데, 이 때 Y 값이 출력되고 있다가 Reset 신호가 들어오고 나서 X 로 표시되는 것이 관찰된다. 하지만 이것은 Reset 신호가 들어오기 전의 일이기 때문에 쓰레기 값이다. 따라서 Y=360 이 되는 지점부터 우리는 고려해주면 된다.

우리의 FPU 모델은 positive edge triggered Unit 이며 입력 신호 이후로 두 사이클 후에 출력이 되는 디자인이다. 하지만 Golden testbench 의 Start 신호 및 Input signal 은 negative edge 에서 들어오고 한 사이클 동안 유지된다. 따라서 우리의 FPU 아웃풋은 Start 신호로부터 두 사이클 반 이후에 나오는 것을 확인할 수 있다. Golden testbench 에서 눈 여겨 볼 것은 Y 값 뿐만 아니라 Overflow 및 Error 가 있다. Y 값의 Verification 은 Table 7, Table 8, Table 9 에서 확인할 수 있으니, figure 을 통해서는 Overflow 및 Error 를 확인해보자.

먼저 Overflow 는 *figure 5, figure 6* 에서 확인할 수 있다. *figure 5* 의 Overflow 는 큰 두 수를 더하면서 INF 출력이 날 때 발생하는 것을 확인할 수 있다. *figure 6* 의 Overflow 는 큰 절대값을 가지는 양수와 음수의 곱셈에 의해 INF 가 발생할 때 출력이 된다.

Error 는 *figure 6* 에서 확인할 수 있다. (+INF-INF), (0 x INF), (0 / 0)을 하는 경우에 발생하고 있는 것을 볼 수 있다. Error 와 Overflow 의 출력은 모두 Y 의 값의 출력과 동시에 나타난다. 입력이 들어온 이후, 그리고 출력이 나오기 이전에 할당된 0 의 값은 계산이 완료되기 전에 나오는 임시의 값이다.

figure 5 와 *figure 6* 를 통해서 Error 와 Overflow 에 대한 Verification 을 완료하였다. 그 다음은 Y 값에 대한 Verification 을 진행하여 보자.

Table 7, Table 8, Table 9 은 *figure 5, figure 6, figure 7, figure 8, figure 9, figure 10, figure 11, figure 12* 의 Y 값을 표로 정리한 것이다. Table 4, Table 5, Table 6 에서 예측하였던 Expected results tables 와 비교하여 Verification 을 진행한다. Expected results 는 C 언어를 통해서 구한 값이니 신뢰할 수 있다. 이 값과 디자인의 출력 값을 비교함으로써 Y 값을 검증할 수 있다. Radix FP32 로 표현된 값은 *figure* 들에 그대로 적혀 있는 값들이다. 하지만 이것은 소수점 5 자리 아래에서 잘려서 출력이 되기 때문에 더욱 정확한 비교를 위해서 Hex 값을 이용하였다. 그 결과 61 개의 결과값이 모두 일치하는 것을 확인하였다.

결론적으로, 우리는 Verification.c 를 통해서 Goldentest 및 직접 만든 testbench(tb_FPU)의 Y 출력값을 예측하였고, 이것은 waveform 을 통해서 일치하는 결과를 얻었다. 또한 error 와 overflow 는 waveform 상에서 출력되는 결과가 우리의 예상과 일치하는 결과를 보였다. 따라서, 우리는 두 개의 testbench 를 통해서 설계한 FPU 의 Functional Correctness 를 검증하였다.

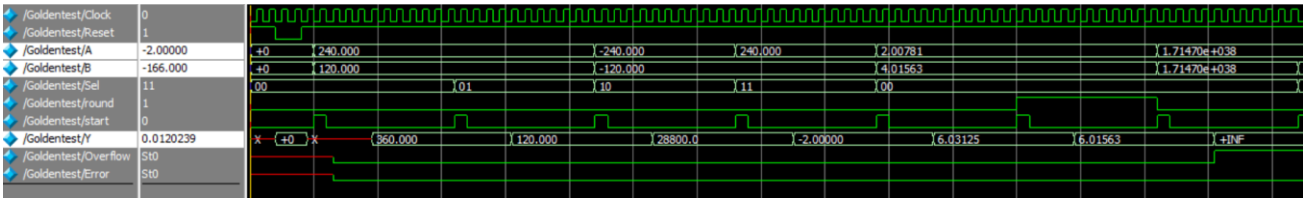


figure 5. Waveform results from 0ns-328ns(7 iteration)

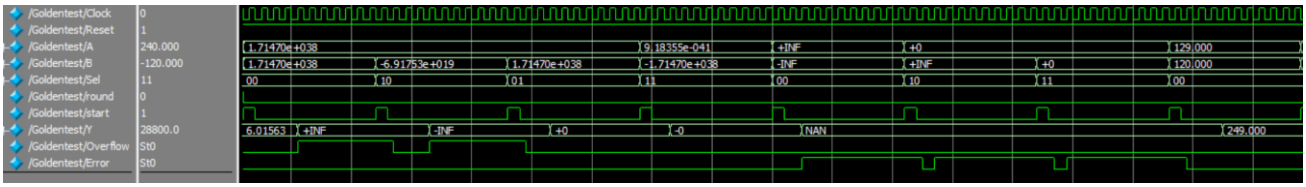


figure 6. Waveform results from 328ns-680ns(7 iteration)

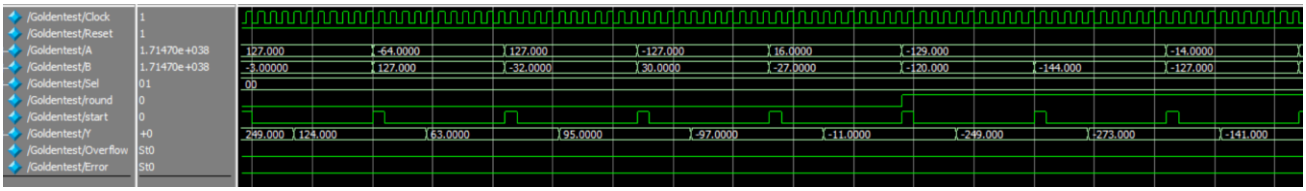


figure 7. Waveform results from 680ns-1032ns(8 iteration)

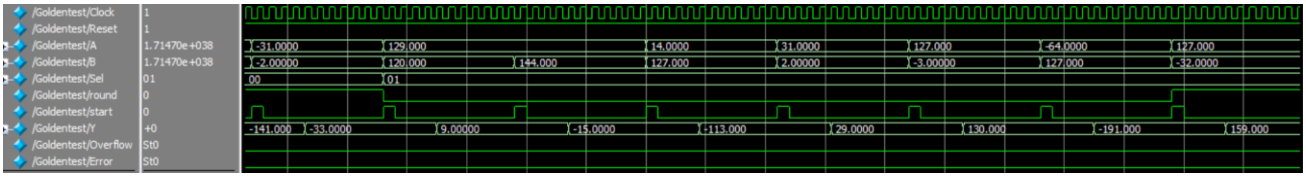


figure 8. Waveform results from 1032ns-1384ns(8 iteration)

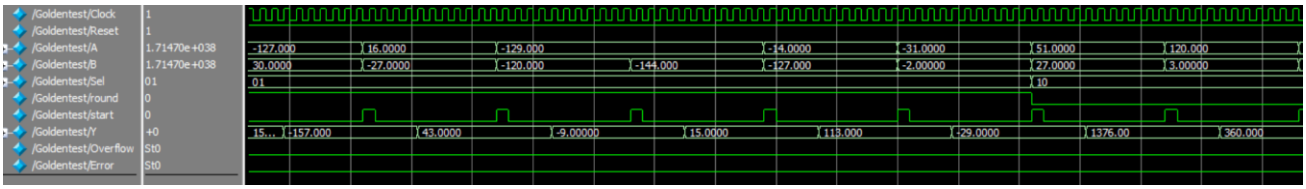


figure 9. Waveform results from 1384ns-1736ns(8 iteration)

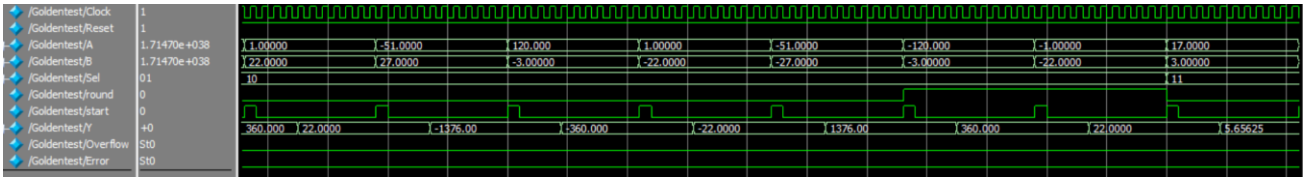


figure10. Waveform results from 1736ns-2088ns(8 iteration)

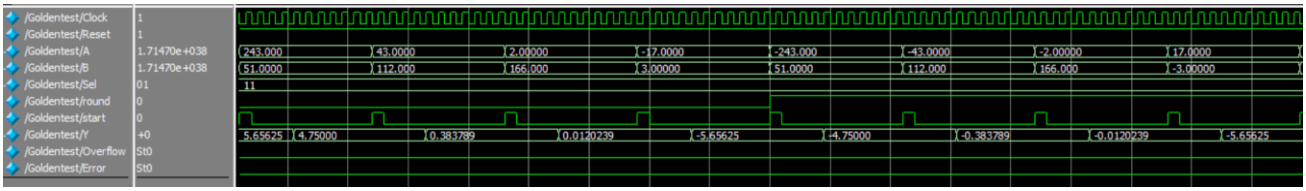


figure 11. Waveform results from 2088ns-2439ns(8 iteration)

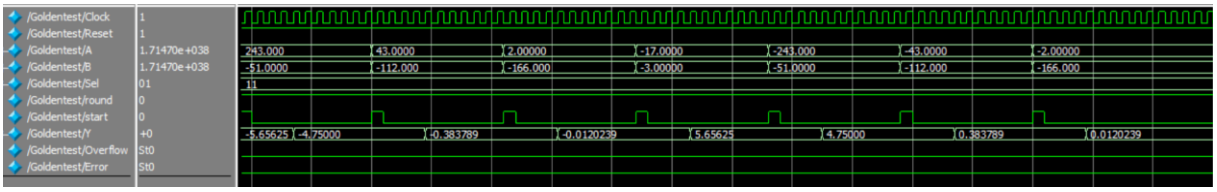


figure 12. Waveform results from 2439ns-2718ns(7 iteration)

Table 7. results according to the iteration 1-21

		Iteration																				
		1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21
Radix	float32	360	120	28800	-2	6.03125	6.01563	inf	-inf	0	-0	NaN	NaN	NaN	249	124	63	95	-97	-11	-249	-273
	HEX	**43b40002**	**42f00000**	**46e10000**	**c0000000**	**40c10000**	**40c08000**	**7f800000**	**ff800000**	**0**	**80000000**	**7fc00000**	**7fc00000**	**7fc00000**	**43790000**	**42f80000**	**427c0000**	**42be0000**	**c2c20000**	**c1300000**	**c3790000**	**c3888000**
Verification		O	O	O	O	O	O	O	O	O	O	O	O	O	O	O	O	O	O	O	O	O

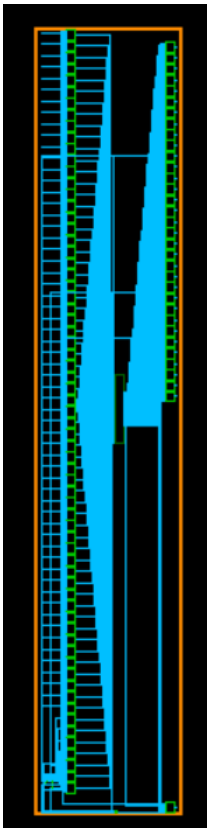
Table 8. results according to the iteration 22-42

		Iteration																				
		22	23	24	25	26	27	28	29	30	31	32	33	34	35	36	37	38	39	40	41	42
Radix	float32	-141	-33	9	-15	-113	29	130	-191	159	-157	43	-9	15	113	-29	1376	360	22	-1376	-360	-22
	HEX	**c30d0000**	**c2040000**	**41100000**	**c1700000**	**c2e20000**	**41e80000**	**43020000**	**c33f0000**	**431f0000**	**c31d0000**	**422c0000**	**c1100000**	**41700000**	**42e20000**	**c1e80000**	**44ac0000**	**43b40000**	**41b00000**	**c4ac0000**	**c3b40000**	**c1b00000**
Verification		O	O	O	O	O	O	O	O	O	O	O	O	O	O	O	O	O	O	O	O	O

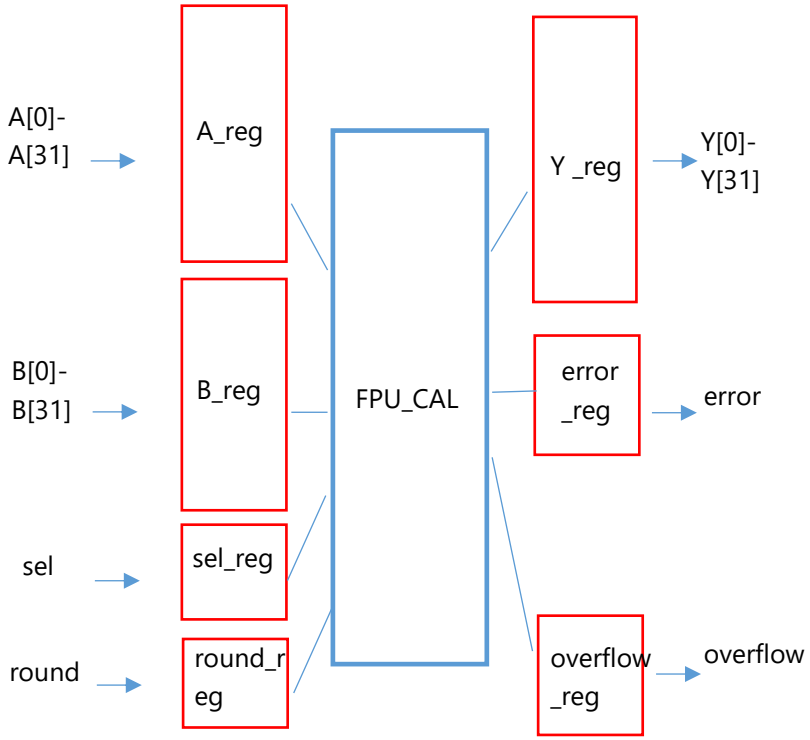
Table 9. results according to the iteration 43-61

		Iteration																			
		43	44	45	46	47	48	49	50	51	52	53	54	55	56	57	58	59	60	61
Radix	float32	1376	360	22	5.625625	4.75	0.383789	0.0120239	-5.65625	-4.75	-0.383789	-0.012024	-5.65625	-4.7	-0.383789	-0.012024	5.65625	4.75	0.383789	0.0120239
	HEX	**44ac0000**	**43b40000**	**41b00000**	**40b50000**	**40980000**	**3ec48000**	**3c450000**	**c0b50000**	**c0980000**	**bec48000**	**bc450000**	**c0b50000**	**c0980000**	**bec48000**	**bc450000**	**40b50000**	**40980000**	**3ec48000**	**3c450000**
verification		O	O	O	O	O	O	O	O	O	O	O	O	O	O	O	O	O	O	O

1. Schematic view of the FPU design



(a) Schematic view of the designed FPU



(b) brief schematic view reproduced from (a)

figure 13. FPU top module schematic view

Design Vision 툴을 이용하여 Synthesis 를 진행하였다. 삼성 32nm 라이브러리를 사용하였고, 합성된 디자인은 *figure 13* 와 같다. (a)는 합성을 통해 출력된 디자인이고, (b)는 이것을 도식화한 것이다. top module 인 FPU 에 대하여 보여주는 Schematic view 로써 우리가 input 으로 지정한 변수인 A 와 B, sel, round 를 위한 인풋 포트가 존재한다. 그리고 정의한 Sub-module 인 FPU_CAL 에 이것을 입력으로 넣어주기 위하여 레지스터를 선언하였기 때문에 그에 따른 레지스터 모듈들이 존재한다. FPU_CAL 모듈에 입력으로 레지스터 값들이 들어간 뒤, 이것의 출력값은 Y_reg, error_reg, overflow_reg 에 저장이 된다. 그리고 아웃풋 포트와 레지스터가 연결 되어있는 것을 확인하였다. 실제로 우리가 선언한 변수이름대로, 인풋, 아웃풋 포트와 레지스터 모듈이 synthesis 되어 있는 것을 확인하였다. 또한 FPU_CAL 모듈이 대부분의 연산을 담당하는데, top module 의 view 에서는 정확히 확인할 수 없었기 때문에 따로 확대하여 분석해보았다.

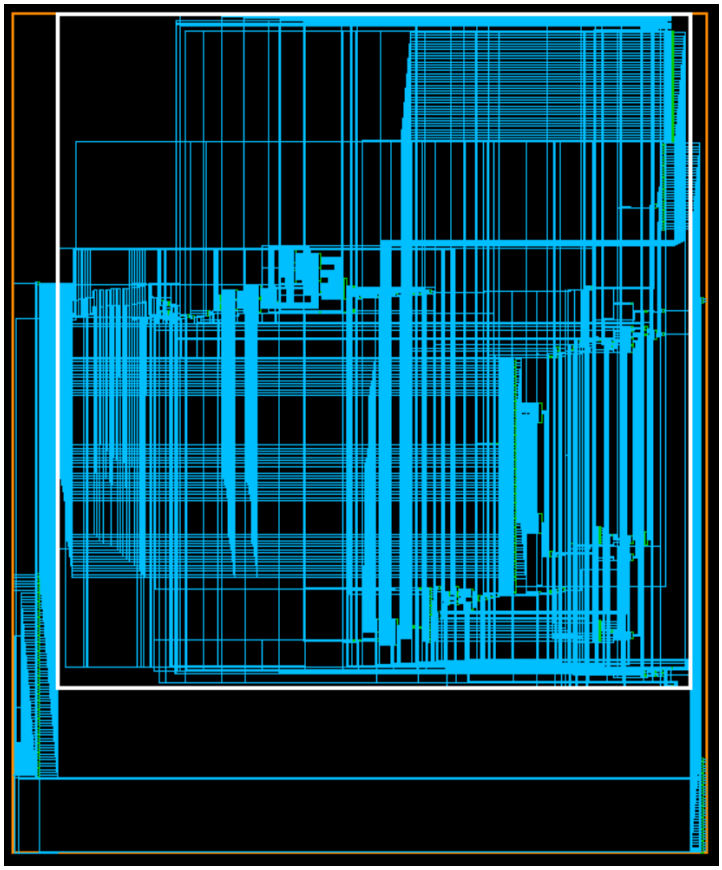


figure 14. inside FPU_CAL module

따로 확대해서 본 FPU_CALU module 의 내부 모습이다. 가장 왼쪽 아래에는 레지스터로부터 들어온 인풋을 위한 포트가 존재한다. 그리고 위쪽에는 STATE machine 을 구현하기 위해 STATE 값을 저장하기 위한 레지스터와 각종 ADD, SUB 모듈과 라이브러리에서 지원한 각종 모듈들이 존재한다. 또 하나의 Sub-module 인 FPU_normalize 는 아래쪽에 작은 크기로 구현이 되었는 것을 확인하였다.

FPU 연산을 위한 사용해 덧셈, 뺄셈, 곱셈, 나눗셈을 RTL 코드에 작성하였고, 그에 따라서 add, sub, mul, div 모듈이 합성된 FPU_CAL 모듈에 위치한 것을 확인할 수 있었다. 특히, 나눗셈은 synthesis 가 되지 않는 것으로 배웠으나, 실제로는 div 모듈을 통해서 '/' operator 가 지원됨을 확인하였다. 가운데에 꽤 큰 크기로 div 모듈이 존재하고 있다.

우리가 명시적으로 작성한 덧셈, 뺄셈, 곱셈, 나눗셈 연산자에 따른 add, sub, mul, div 모듈 외에도, 기대하지 않았던 수많은 덧셈, 뺄셈, gt, c~ 모듈들이 위치한 것을 확인했다. 이것들은 synthesis 툴에서 특정 패턴에 따라서 모듈을 설정하여 합성을 진행하고 있다는 뜻을 해석할 수 있었다. 모듈들 뿐만 아니라 AND, OR, INVERTER, XOR, NOT 등의 primitives 들도 활용하여 Synthesis 를 진행한 것을 확인할 수 있었다.

2. Area report with analysis

```
****************************************
Report : area
Design : FPU
Version: H-2013.03-SP5-4
Date   : Wed Dec 16 15:59:26 2020
****************************************

Library(s) Used:

    saed32rvt_ff1p16v125c (File: /data3/Class/DSD/2020_2/2020_2_dsd16/lib/SAED32_EDK/saed32rvt_ff1p16v125c.db)

Number of ports:                          104
Number of nets:                           300
Number of cells:                          197
Number of combinational cells:            110
Number of sequential cells:                86
Number of macros/black boxes:               0
Number of buf/inv:                         23
Number of references:                       6

Combinational area:              11154.380143
Buf/Inv area:                     1637.195662
Noncombinational area:            1665.659821
Macro/Black Box area:                0.000000
Net Interconnect area:            2637.539586

Total cell area:                 12820.039963
Total area:                      15457.579549
1
```

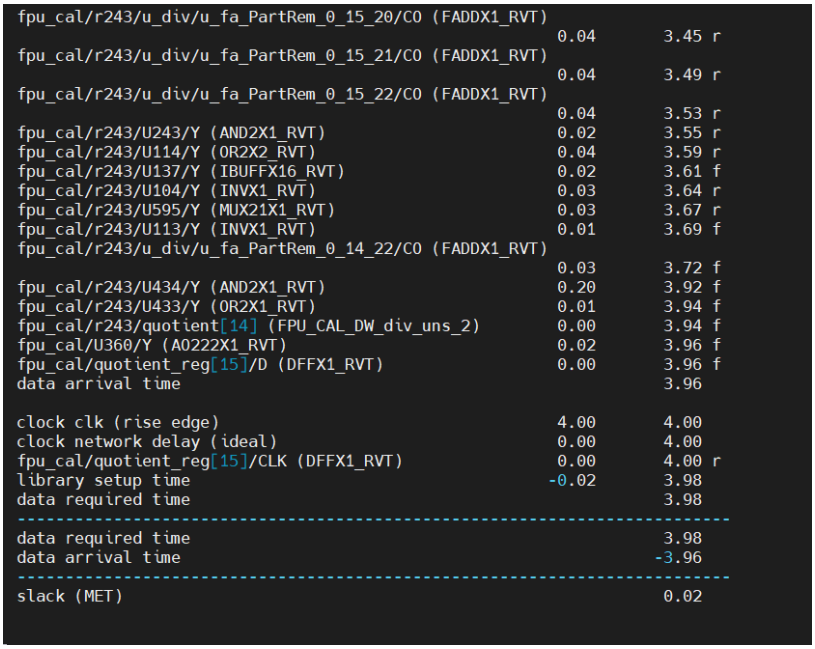
total cell area = 12820

Area 는 총 12820 으로 설계하였다. A 기준 spec 인 35000 보다 절반 넘게 절감한 수치이다. Spec 보다 작은 Area 에 도달할 수 있었던 이유는 모든 분기문마다 default statement 를 삽입하여 불필요한 latch 의 발생을 없앴고, *figure 14* 에서 볼 수 있듯이 회로들이 series 하게 배열되어 있기 때문에, parallel 한 설계보다 Area 가 적게 발생하였다. 다만 이 경우 parallel 설계보다 timing 이 느린 단점이 있다.

우리의 FPU 는 두 번의 클럭 사이클 동안 계산을 진행하도록 설계가 되었다. 총 4 개의 STATE 가 있지만, 실질적으로 모든 상황에서 2 번의 STATE 이동을 통해 출력 STATE 로 이동할 수 있기 때문에 두 사이클 안에 출력이 가능하다. 8 사이클 안에 출력이 가능하면 된다는 Spec 을 모두 사용하지 않고, 두 사이클만을 사용함으로써 회로를 크게 만들지 않을 수 있었다. 다만 이것 역시 timing 과는 trade-off 관계가 있다.

Area 를 줄일 수 있었던 또 하나의 가장 중요한 이유는 Adder, Subtractor, Multiplier, Divider 를 따로 sub-module 로 만들지 않았던 것이다. FPU_CAL 모듈에 모든 연산 기능을 탑재하였고, Adder 와 Subtractor 는 하나의 로직 안에 포함시켜서 둘을 따로 구현하는데 발생하는 Cost 를 최소화할 수 있었다.

3. Timing report with analysis



```
fpu_cal/r243/u_div/u_fa_PartRem_0_15_20/CO (FADDX1_RVT)
                                                 0.04       3.45 r
fpu_cal/r243/u_div/u_fa_PartRem_0_15_21/CO (FADDX1_RVT)
                                                 0.04       3.49 r
fpu_cal/r243/u_div/u_fa_PartRem_0_15_22/CO (FADDX1_RVT)
                                                 0.04       3.53 r
fpu_cal/r243/U243/Y (AND2X1_RVT)                 0.02       3.55 r
fpu_cal/r243/U114/Y (OR2X2_RVT)                  0.04       3.59 r
fpu_cal/r243/U137/Y (IBUFFX16_RVT)               0.02       3.61 f
fpu_cal/r243/U104/Y (INVX1_RVT)                  0.03       3.64 r
fpu_cal/r243/U595/Y (MUX21X1_RVT)                0.03       3.67 r
fpu_cal/r243/U113/Y (INVX1_RVT)                  0.01       3.69 f
fpu_cal/r243/u_div/u_fa_PartRem_0_14_22/CO (FADDX1_RVT)
                                                 0.03       3.72 f
fpu_cal/r243/U434/Y (AND2X1_RVT)                 0.20       3.92 f
fpu_cal/r243/U433/Y (OR2X1_RVT)                  0.01       3.94 f
fpu_cal/r243/quotient[14] (FPU_CAL_DW_div_uns_2) 0.00       3.94 f
fpu_cal/U360/Y (AO222X1_RVT)                     0.02       3.96 f
fpu_cal/quotient_reg[15]/D (DFFX1_RVT)           0.00       3.96 f
data arrival time                                           3.96

clock clk (rise edge)                            4.00       4.00
clock network delay (ideal)                      0.00       4.00
fpu_cal/quotient_reg[15]/CLK (DFFX1_RVT)         0.00       4.00 r
library setup time                              -0.02       3.98
data required time                                          3.98
--------------------------------------------------------------------------
data required time                                          3.98
data arrival time                                          -3.96
--------------------------------------------------------------------------
slack (MET)                                                 0.02
```

Slack = 0.02 @ Clock frequency = 250MHz, Clock period = 4ns

Slack 이란 주어진 Clock period 안에서 얼마만큼 여유가 있는 지를 나타내는 지표이다. 우리의 Design 은 Clock frequency 250MHz(Clock period 4ns)일 때, 0.02 의 값이 나왔으니 이는 4ns 의 Clock period 중 3.98 ns 만을 사용하고 나머지 0.02 동안은 여유롭게 대기할 수 있다는 것이다. 따라서 우리의 디자인은 A 기준 Spec 인 250MHz 를 만족하였다.

Area 부분에서 기술했던 것처럼, Area 를 위해서 timing 을 희생한 부분들이 있었다. 그럼에도 불구하고 timing spec 을 만족하였다.

그 이유 중 하나는 STATE 이다. 가장 초기의 설계에서는 FPU_CAL 모듈에 입력이 들어오자 마자 출력 값을 모두 계산해내는 Combinational Logic 을 설계하였다. FPU_CAL 모듈은 top module 과 clock 신호를 주고받지 않기 때문에 입력과 동시에 계산이 완료될 수 있었다. 하지만 이 경우 timing spec 을 맞출 수 없었다. 4ns 동안 FPU_CAL 이 주어진 연산을 모두 완료할 수 없었기 때문이다. 비록 testbench 시뮬레이션 상에서는 올바른 waveform 이 나오지만 이것은 timing 을 고려하지 않은 채 functional correctness 만 판단하기 때문이다.

따라서 우리는 STATE 에 따라서 계산 과정이 나뉘어지는 Sequential Logic 으로 FPU_CAL 모듈을 설계하였고, Slack 상으로 0.31 정도의 이득을 얻을 수 있었다. Area 의 손해를 줄이기 위해서 최소한의 STATE 만을 사용하였고 그 결과 timing 과 Area Spec 을 모두 만족하였다.