RISC-V Floating Point Unit

ECE 571 - Final Project Presentation

Chuck Faber, Bhargavi Chunchu, Ramprakash Baskar, Keerthan Nayak

https://github.com/Keerthan1994/RISC-V-Floating-Point-Processor

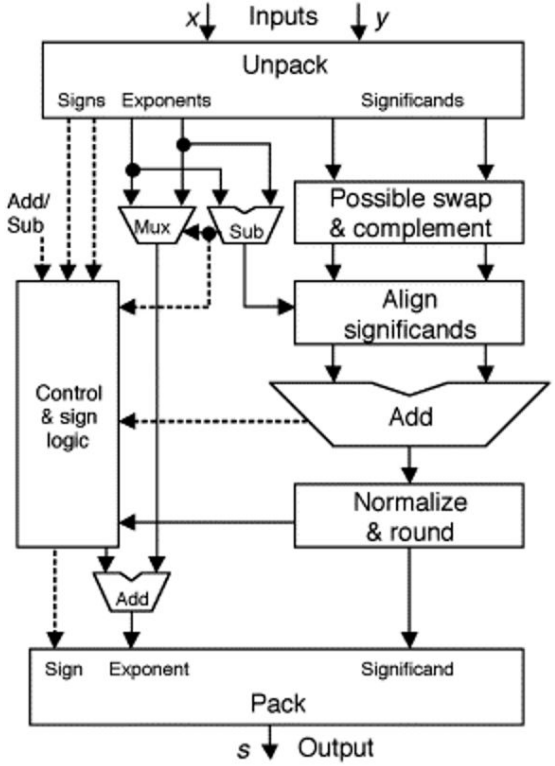
Project Proposal Overview

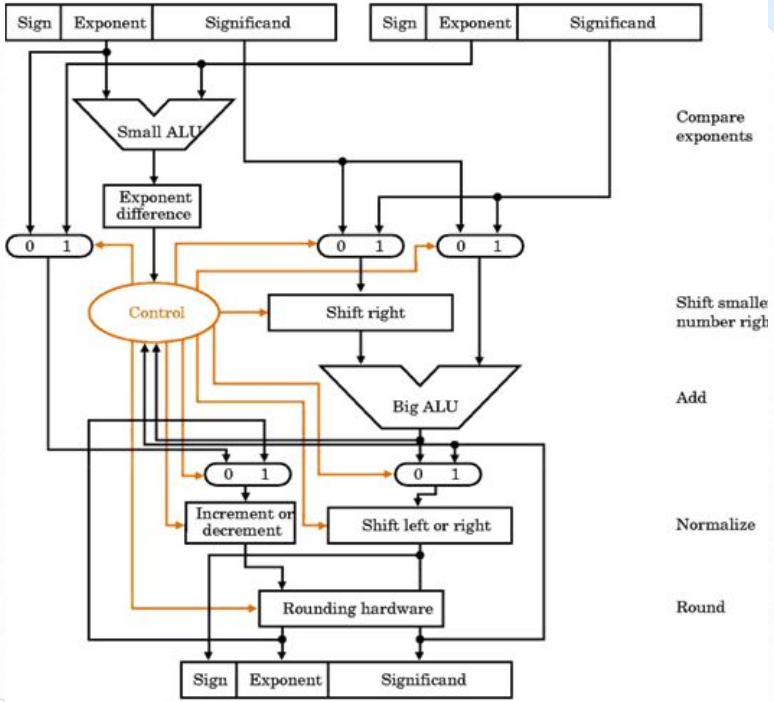
Develop a Floating point unit for a RISC V processor that does

- Addition
- Subtraction
- Multiplication
- Division

Operands are represented in the hardware using IEEE 754 format. Try to implement it into a RISC-V processor.

Our Starting Point





**Full References at End of Presentation*

Who Did What

Add/Sub Modules

Chuck and Bhagi worked on the Add/Subtract modules.

Chuck also wrote the packages and testbench.

Mult/Div Modules

Keerthan worked on the multiply module.

Ram worked on the divide module and RISCV processor search.

What We Accomplished

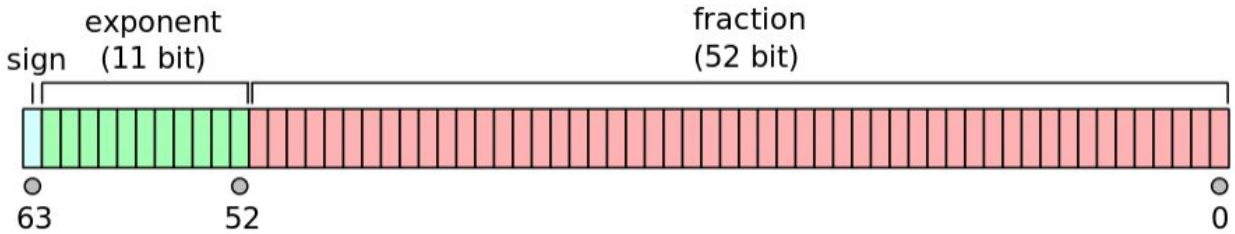
- A Single-Precision Floating Point unit with three operations
    - Addition, Subtraction, Division
- A directed random testbench that tests randomized operands of all different corner cases
- A weight random testbench that generates FP fields based on set weights.
- Standardized error outputs for Overflow/Invalid/Underflow, etc. for some of the operations
- “Round to Even” rounding schema.
- Multiplier is fixed and working as expected.

What We Didn’t Get To...

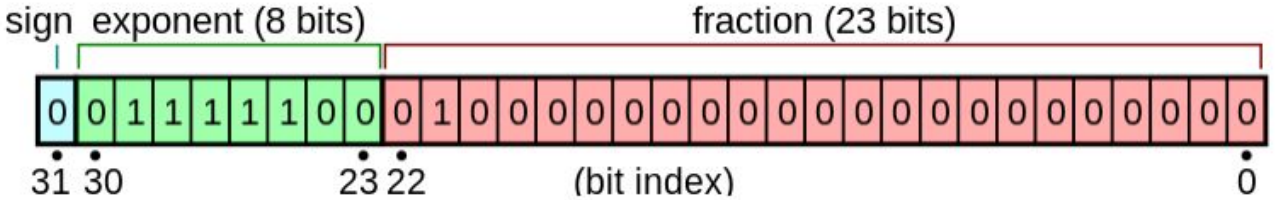
- Because of the time constraints we could not integrate our FPU into a RISC V processor as we originally planned.

Some IEEE 754 F.P. Review

64-bit Double Precision F.P.

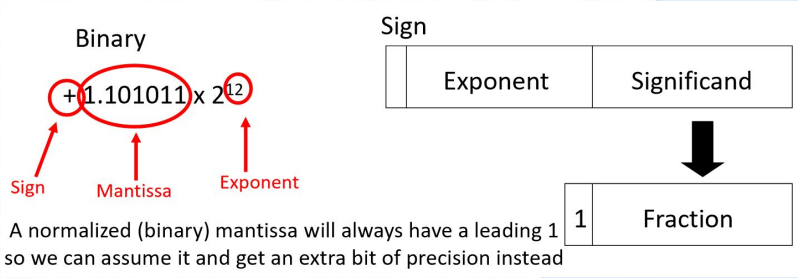


32-bit Single Precision F.P.



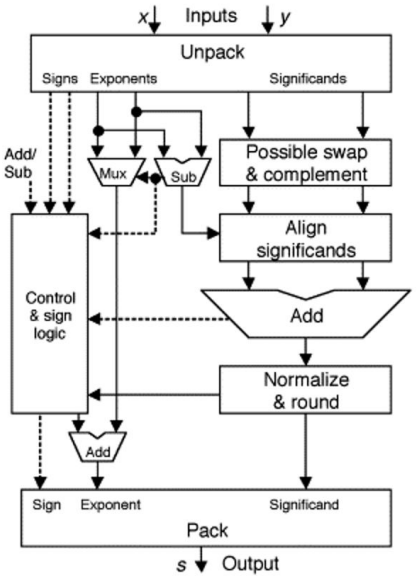
Special Cases:

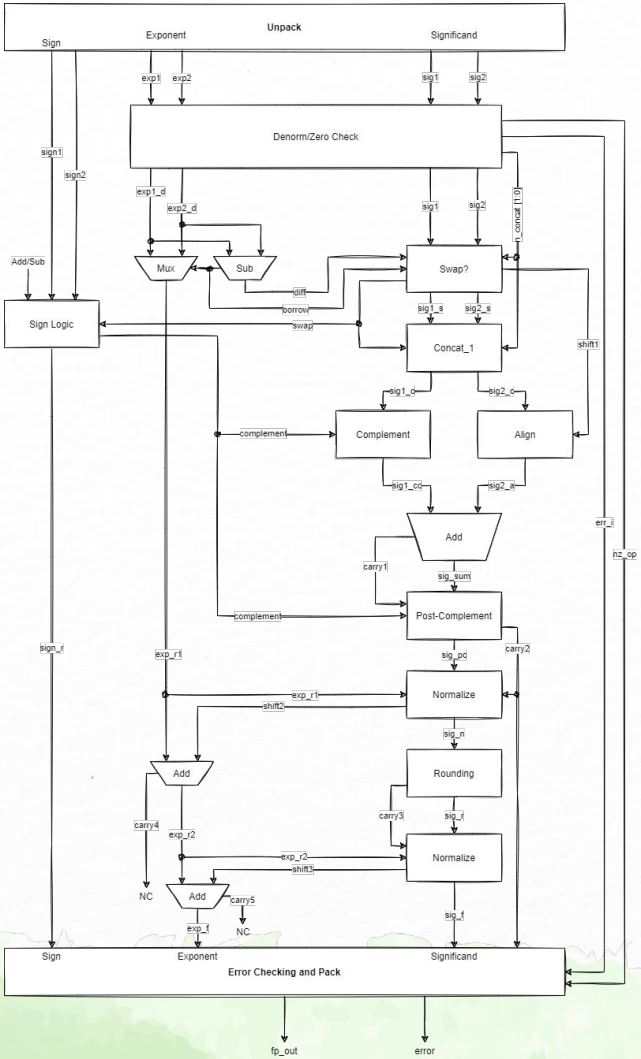
- Denormalized Numbers
- Infinity
- NaN
- Zero



Addition / Subtraction Design Details

F.P. Addition And Subtraction





Add/Sub HW - Sign Logic

Sign and Complement Logic					
Inputs				***Outputs***	
Op2 > Op1	Add/Sub Op	Sign1	Sign2	Sign_Out	Complement?
0	0	0	0	0	0
0	0	0	1	0	1
0	0	1	0	1	1
0	0	1	1	1	0
0	1	0	0	0	1
0	1	0	1	0	0
0	1	1	0	1	0
0	1	1	1	1	1
1	0	0	0	0	0
1	0	0	1	1	1
1	0	1	0	0	1
1	0	1	1	1	0
1	1	0	0	1	1
1	1	0	1	0	0
1	1	1	0	1	0
1	1	1	1	0	1

Sign_Out K-map

{Swap,Opcode}/ {Sign1,Sign2}	00	01	11	10
00	0	0	1	1
01	0	0	1	1
11	1	0	0	1
10	0	1	1	0

```
sign_out = (!swap && sign1) || (swap && opcode && !sign2) || (swap
&& !opcode && sign2)
```

Complement K-Map

{Swap,Opcode}/ {Sign1,Sign2}	00	01	11	10
00	0	1	0	1
01	1	0	1	0
11	1	0	1	0
10	0	1	0	1

```
complement = (opcode && !sign1 && !sign2) || (!opcode && !sign1 &&
sign2) || (opcode && sign1 && sign2) || (!opcode && sign1 && !sign2)
```

Normalization and Rounding

- Once the significands are added/subtracted and complemented as needed, they are normalized
- Normalization is a process performed on a binary number except zero where exponent is adjusted so that radix falls onto the right of the leftmost bit 1
- Floating point computation result is often not equal to the “true” result.
- Rounding is performed to choose the result with least error using the rounding bits.

TABLE A

Sign = 1

Exponent = 1111110

Fraction = 0.00110101010101010011

TABLE B

Sign = 1

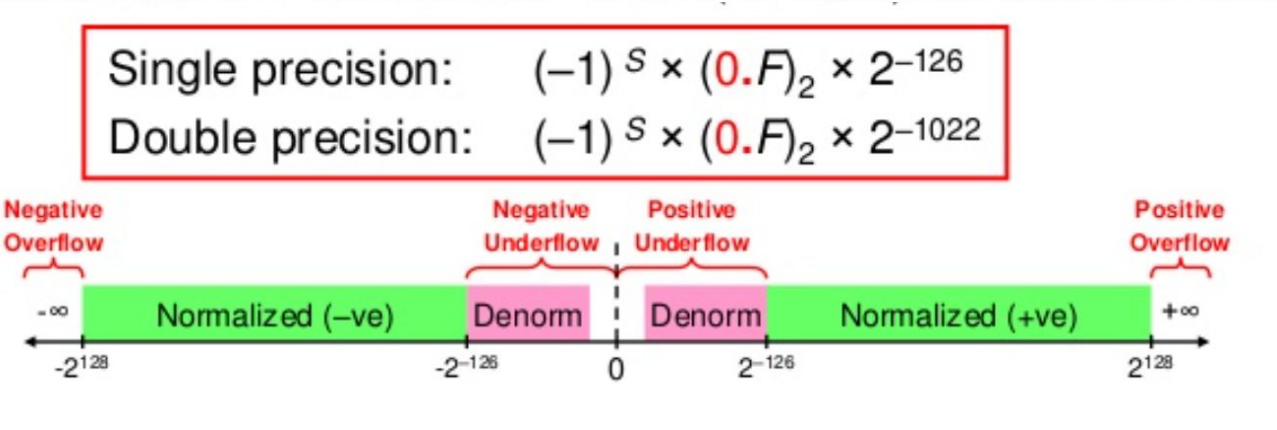
Exponent = 11111010

Fraction = 1.10101010101010110000

G	R	S	Rounding Action
0	0	0	Truncate
0	0	1	Truncate
0	1	0	Truncate
0	1	1	Truncate
1	0	0	Round to Even
1	0	1	Round Up
1	1	0	Round Up
1	1	1	Round Up

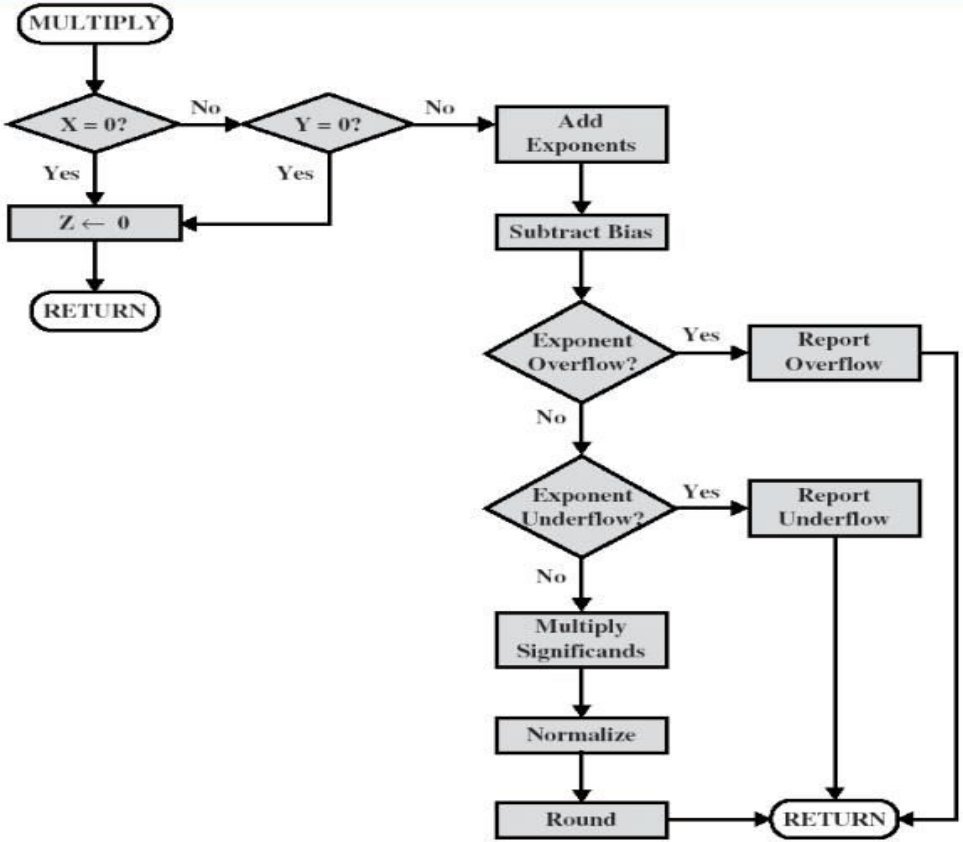
Denormalized Numbers

- IEEE standard uses denormalized numbers to fill the gap between 0 and smallest normalized float and provide gradual underflow to zero
- Exponent field is 0 and fraction != 0
- No implicit leading 1
- We specified that our design should handle denormalized numbers and this complicated our design further.

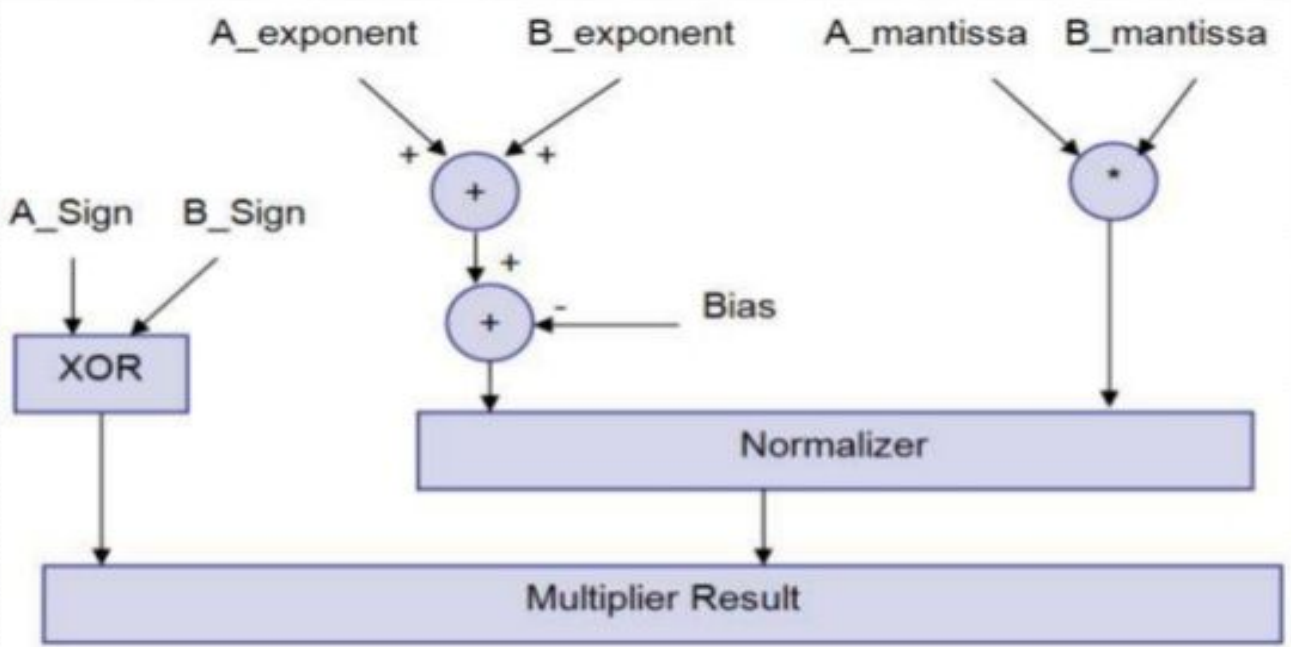


Multiply / Divide Design Details

Floating point Multiplication



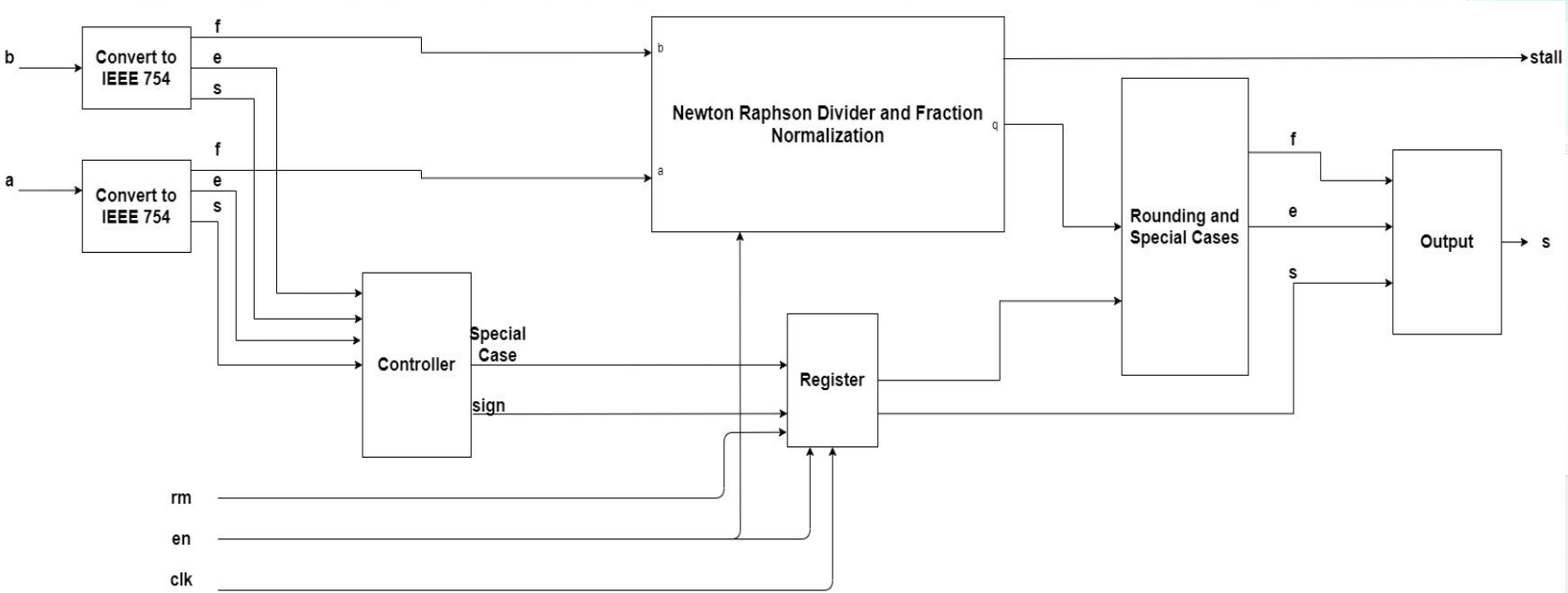
Block Diagram of Floating Point Multiplier



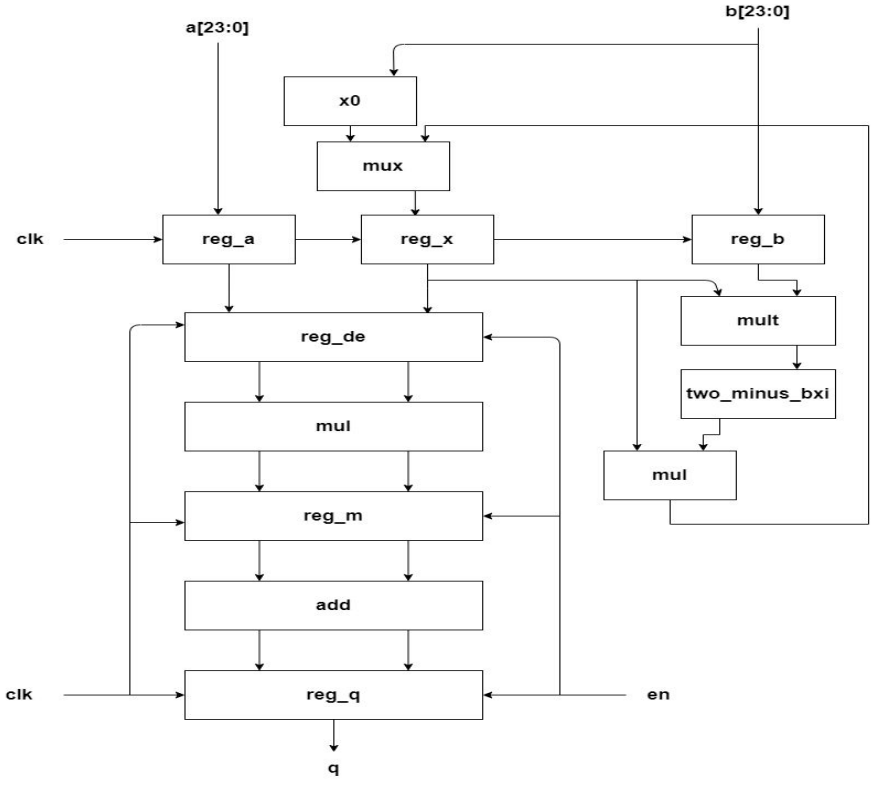
Divider

- Floating point division: $a/b = (a_s/b_s)*2^{(ae - be)}$
- Significand division: (a_s/b_s) - use Newton Raphson Method
- $(a/b) \sim (a*(1/b))$
- $f(x)$, find x such that $f(x) \sim 0$
- Start with x_0, then use tangential equation $y-f(x_0)=0$
- The equation is continually iterated: $x_{i+1} = x_i(2-x_ib)$
- $q=a*x_n$

Divider Design



Newton Raphson Method



Exception and Error Handling

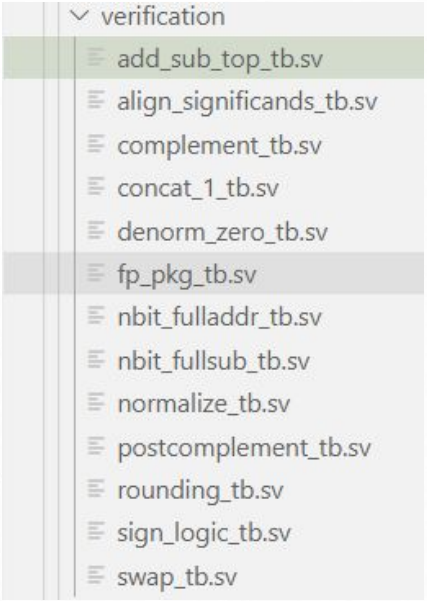
- As a group we wanted to implement the following 5 error outputs:
  - INVALID
  - DIVBYZERO
  - OVERFLOW
  - UNDERFLOW
  - NONE
- The error signal is determined based on the output exponent and significand bits.
  - INVALID - NaN
  - OVERFLOW - Infinity or Overflow
  - UNDERFLOW - Denormalized Number

Design Verification

Independent Testing of Add/Sub Modules

```
module top();

logic [26:0] sig_n, sig_r;
logic co;

rounding r0 (.*);

initial begin
    for (int i = 0; i < 10; i++) begin
        sig_n = $urandom();
        for(int j = 0; j < 8; j++) begin
            sig_n[2:0] = j;
            #10;
            $display("IN: sig_n: %24b %3b. co: %1b.
        end
    end
end

endmodule
```

Testing Rounding Module with Simple TB




```
\nresult: %1b %8b %027b\n", ast0.swap, ast0.complement, ast0.exp_r1, ast0.shift1, ast0.shift2, ast0.exp_r2,
```

Hierarchical Referencing of Signals was very useful

Multiply/Divide Testing Process

```
class transaction;
  rand bit[22:0] mantissa_A;
  rand bit[7:0] exponent_A;
  rand bit sign_A;

  rand bit[22:0] mantissa_B;
  rand bit[7:0] exponent_B;
  rand bit sign_B;

  constraint mant{
  mantissa_A != 0;
  mantissa_B != 0;
  }

  constraint expo{
  exponent_A != 0;
  exponent_B != 0;
  }

  constraint mantis{
  mantissa_A != 7'b1;
  mantissa_B != 7'b1;
  }

endclass
```

```
initial begin
  transaction tx;
  tx = new();

  repeat(100) begin
    assert(tx.randomize());
    repeat(1) begin
     // Add driver here
      strb_A = 1'b1;
      strb_B = 1'b1;
      out_prod_ack = 1'b1;
      in_A= {tx.sign_A,tx.mantissa_A,tx.exponent_A};
      in_B= {tx.sign_B,tx.mantissa_B,tx.exponent_B};
      @(posedge clk);

      strb_A = 1'b0;
      strb_B = 1'b0;
      out_prod_ack = 1'b0;
      @(posedge clk);


    end
  end
  #100 $finish;
end


endmodule
```

Directed Randomization Approach

Add/Sub Testing Plan

	Op Combination Table	
	Operand Combinations	
TC	Op1	Op2
0	Reg	Reg
1	Reg	NaN
2	Reg	Inf
3	Reg	Zero
4	Reg	Denorm
5	Reg	Max
6	Reg	NormMin
7	Reg	DenormMin
8	NaN	Reg
9	NaN	NaN
10	NaN	Inf
11	NaN	Zero
12	NaN	Denorm
13	NaN	Max
14	NaN	NormMin
15	NaN	DenormMin
16	Inf	Reg
17	Inf	NaN
18	Inf	Inf
19	Inf	Zero
20	Inf	Denorm
21	Inf	Max
22	Inf	NormMin

Special Cases	exponent	significand
MAX	11111110	11111111111111111111111
NORM_MIN	00000001	00000000000000000000000
DENORM_MIN	00000000	00000000000000000000001
INF	11111111	00000000000000000000000
NaN	11111111	non-zero
DENORM_MIN	00000000	non-zero
ZERO	00000000	all zeros

8 Special Cases
64 Possible Combinations
Randomized Operands where Possible
Test all sign combinations

FloatingPoint Object and Constrained Random

```
class FloatingPoint;

    local fp_t fp;
    local rand bit sign;
    local rand bit [EXP_BITS-1:0] exponent;
    local rand bit [SIG_BITS-1:0] significand;
    fp_case_t op_case;

    //Randomizaton Weight for non-edge-case testing
    int exp_zero_w = 1, exp_reg_w = 98, exp_max_w = 1;
    int sig_zero_w = 1, sig_reg_w = 98, sig_max_w = 1;

    //Randomization constraints
    constraint edge_case_c {
        if (op_case == NAN) {
            exponent == {EXP_BITS{1'b1}};
            significand > 1;
        } else if (op_case == INF) {
            exponent == {EXP_BITS{1'b1}};
            significand == 'b0;
        } else if (op_case == ZERO) {
            exponent == 'b0;
            significand == 'b0;
```

```
    constraint rand_exp_c {
        exponent dist {8'b0000_0000 :/ w_exp_zero,
        [8'b0000_0001:8'b1111_1110] :/ w_exp_reg,
        8'b1111_1111 :/ w_exp_max};
    }

    constraint rand_sig_c {
        significand dist {23'h000000 :/ w_sig_zero,
        [23'h000001:23'h7FFFFE] :/ w_sig_reg,
        23'h7FFFFF :/ w_sig_max};
    }
```

Testing Method with FloatingPoint Class

```
//·Using·this·class:
//·1.·Create·FloatingPoint·Objects·for·OP1,·OP2,·and·OUT·and·EXP.
//·2.·Use·generateCase(fp_case_t)·to·generate·a·new·randomized·value·for·OP1,·and·OP2·of·specified·type.
//·2a.·Use·OP1.setSign(sign)·and·OP2.setSign(sign)·to·set·the·appropriate·signs
//·3.·Use·EXP.setSR(OP1.getSR·+·OP2.getSR)·and·feed·it·the·shortreal·result·from·SV.
//·4.·Feed·the·machine·OP1.sign,·OP1.exponent,·OP1.significand,·etc.·Also·for·OP2.
//·5.·Use·OUT.setBits(machine_output)·and·feed·it·the·machine·output·to·set·the·OUT·value.
//·6.·Use·OUT.equals(EXP)·to·see·if·they·are·the·same!
```

```
····//·Sets·op_case·and·randomizes·accordingly
····function·void·generateCase(fp_case_t·op_case);
····|···this.op_case·=·op_case;
····|···this.constraint_mode(0);
····|···this.edge_case_c.constraint_mode(1);
····|···assert(this.randomize())
····|···else·$fatal(0,·"FloatingPoint::generateCase·-·randomize·failed");
····|···if·(op_case·!==·this.op_case)·$error("FloatingPoint::generateCase··
····endfunction

····//·Generates·a·random·operand·based·on·the·distribution·weights
····function·void·generateRandom();
····|···this.constraint_mode(0);
····|···this.rand_exp_c.constraint_mode(1);
····|···this.rand_sig_c.constraint_mode(1);
····|···assert(this.randomize())·else·$fatal(0,·"FloatingPoint::generateRar
····endfunction
```

```
····//·Checks·if·one·FloatingPoint·Object·is·the·Same·as·this·one.
····function·bit·equals(FloatingPoint·obj);
····|···if·(obj.op_case·!=·this.op_case)·return·0;
····|···if·(this.op_case·==·NAN)·return·1;
····|···else·if·(obj.sign·==·this.sign·&&·obj.exponent·==·this.exponent·
····|···&&·obj.significand·==·this.significand)·return·1;
····|···else·return·0;
····endfunction
```

Debugging Process

```
# //
# Loading sv_std.std
# Loading work.addpkg(fast)
# Loading work.align_significands_sv_unit(fast)
# Loading work.add_sub_top_sv_unit(fast)
# Loading work.add_sub_top_tb_sv_unit(fast)
# Loading work.top(fast)
# 14500::VALUE/TYPE MISMATCH: OP1=1.172168e-39 DENORM. OP2=-1.893709e-39 DENORM. OP_CODE=0. EXP=-7.215412e-40. RES=-8.354876e+37. EXP_TYPE=DENORM. RES_TYPE=REG.
# Internal Error: NO_ERR. Final Carry: 0
# EXP_BITS=1 00000000 00001111101101101011101.
# RES_BITS=1 11111100 11110110110101110100000.
# 14550::VALUE/TYPE MISMATCH: OP1=-6.875390e-39 DENORM. OP2=7.666105e-39 DENORM. OP_CODE=0. EXP=7.907149e-40. RES=9.155853e+37. EXP_TYPE=DENORM. RES_TYPE=REG.
# Internal Error: NO_ERR. Final Carry: 0
# EXP_BITS=0 00000000 00010001001110000110001.
# RES_BITS=0 11111101 00010011100001100010000.
# 14600::VALUE/TYPE MISMATCH: OP1=-4.698692e-39 DENORM. OP2=-4.670253e-39 DENORM. OP_CODE=0. EXP=-9.368946e-39. RES=-6.982948e-39. EXP_TYPE=DENORM. RES_TYPE=DENORM.
# Internal Error: NO_ERR. Final Carry: 1
# EXP_BITS=1 00000000 11001100000010011001111.
# RES_BITS=1 00000000 10011000000100110011110.
```

Normalization Logic needed

~~Logic in Swap needed~~ fixing

<< 5 exp

Denorm - Denorm

<< 4 exp

- Denorm + Denorm

<< 1

- Denorm - Denorm

```
# Loading sv_std.std
# Loading work.addpkg(fast)
# Loading work.align_significands_sv_unit(fast)
# Loading work.add_sub_top_sv_unit(fast)
# Loading work.add_sub_top_tb_sv_unit(fast)
# Loading work.top(fast)
# 248450::VALUE/TYPE MISMATCH: OP1=3.422043e-38 REG. OP2=-1.175494e-38 NORMMIN. OP_CODE=0. EXP=2.246548e-38. RES=4.493096e-38. EXP_TYPE=REG. RES_TYPE=REG.
# Swap: 0. Complement: 1. Borrow: 0. Diff: 1
# OP1_BITS=0 00000010 01110100101000001010000.
# OP2_BITS=1 00000001 00000000000000000000000.
# EXP_BITS=0 00000001 11101001010100001010000.
# RES_BITS=0 00000010 11101010101000001010000.
# 0 00000010 11110100101000001010000000 Carry: 0
# 248550::VALUE/TYPE MISMATCH: OP1=7.629149e-37 REG. OP2=-1.175494e-38 NORMMIN. OP_CODE=0. EXP=7.511599e-37. RES=1.502320e-36. EXP_TYPE=REG. RES_TYPE=REG.
# Swap: 0. Complement: 1. Borrow: 0. Diff: 6
# OP1_BITS=0 00000111 00000011100110101000001.
# OP2_BITS=1 00000001 00000000000000000000000.
# EXP_BITS=0 00000110 11111111001101010000010.
# RES_BITS=0 00000111 11111111001101010000010.
# 0 00000111 11111111100110101010000010000 Carry: 0
# 250150::VALUE/TYPE MISMATCH: OP1=-2.819400e-38 REG. OP2=1.175494e-38 NORMMIN. OP_CODE=0. EXP=-1.643906e-38. RES=-3.287812e-38. EXP_TYPE=REG. RES_TYPE=REG.
# Swap: 0. Complement: 1. Borrow: 0. Diff: 1
# OP1_BITS=1 00000010 00110011000000101101010.
# OP2_BITS=0 00000001 00000000000000000000000.
# EXP_BITS=1 00000001 01100110000000101101000.
# RES_BITS=1 00000010 01100110000000101101000.
# 1 00000010 10110011000000010110100000 Carry: 0
```

Reg - Normmin

exp
res = exp + 1

sig ok

Reg - Normin

exp
res = exp + 1

-Reg + Normin

exp = exp + 1

```
RES=nan. EXP_TYPE=INF. RES_TYPE=NAN.
W. RES_ERR=INVALID.
.. EXP=inf. RES=nan. EXP_TYPE=INF. RES_TYPE=NAN.
!R=OVERFLOW. RES_ERR=INVALID.
:. RES=nan. EXP_TYPE=INF. RES_TYPE=NAN.
W. RES_ERR=INVALID.
if. RES=-nan. EXP_TYPE=INF. RES_TYPE=NAN.
W. RES_ERR=INVALID.
E=0. EXP=0.000000e+00. RES=-0.000000e-00. EXP_TYPE=ZERO. RES_TYPE=ZERO.
DE=1. EXP=0.000000e+00. RES=-0.000000e+00. EXP_TYPE=ZERO. RES_TYPE=ZERO.
DE=0. EXP=1.168239e-38. RES=inf. EXP_TYPE=DENORM. RES_TYPE=INF.
!R=UNDERFLOW. RES_ERR=OVERFLOW.
COD=0. EXP=-1.092033e-38. RES=-inf. EXP_TYPE=DENORM. RES_TYPE=INF.
RR=UNDERFLOW. RES_ERR=OVERFLOW.
CODE=0. EXP=5.849025e-39. RES=nan. EXP_TYPE=DENORM. RES_TYPE=NAN.
RR=UNDERFLOW. RES_ERR=INVALID.
_CODE=0. EXP=-1.205667e-39. RES=-1.396067e+38. EXP_TYPE=DENORM. RES_TYPE=REG.
_ERR=UNDERFLOW. RES_ERR=NONE.
DE=1. EXP=-3.400325e-39. RES=-nan. EXP_TYPE=DENORM. RES_TYPE=NAN.
```

inf check has to be before nan

(1) exact logic code

Reg - - inf = Reg + inf

(2) Reg - - Max → inf (incorrect logic?)

inf - - Reg = inf + Reg (see (1))

- inf - Reg (see (1))

(3) Adjust equal method

(4) something wrong in module

Company 0 and -0

zero + denorm

carry? error topic

SystemVerilog Constructs

Shared Package

Enumerated Types

```
//////////////////////
// Enumerated Types //
//////////////////////

// Internal Error Codes and Flags
typedef enum logic [2:0] {
    NO_ERR, ZERO_ERR, ZERO_OP_ERR, NAN_ERR, INF_ERR
} i_err_t;

// Output Error Codes
typedef enum logic [2:0] {
    NONE, INVALID, DIVBYZERO, OVERFLOW, UNDERFLOW, INEXACT
} o_err_t;

// Special Test Cases
typedef enum logic [2:0] {
    REG, NAN, INF, ZERO, DENORM, MAX, NORMMIN, DENORMMIN
} fp_case_t;

// Operation Codes
typedef enum logic[1:0] {
    ADD, SUB, MUL, DIV
} opcode_t;
```

Structs and Unions

```
////////////////////////
// Structs and Unions //
////////////////////////

// IEEE 754 Single Precision Floating Point Format
typedef struct packed {
    logic sign;
    logic [EXP_BITS-1:0] exponent;
    logic [SIG_BITS-1:0] significand;
} ieee754_sp_t; //Breaks the 32-bit short real into bits

// Generic floating point type to use for unpacking
typedef union {
    logic [EXP_BITS+SIG_BITS:0] bits; //Shortreal (32 bit) float value
    ieee754_sp_t unpkg; //Single Precision Floating Point Unpacked
} fp_t;
```

Classes and Methods

```
////////////////////////
// Classes and Objects //
////////////////////////

class FloatingPoint;

    local fp_t fp;
    local rand bit sign;
    local rand bit [EXP_BITS-1:0] exponent;
    local rand bit [SIG_BITS-1:0] significand;
    fp_case_t op_case;
```

Constraints

```
    constraint rand_exp_c {
        exponent dist {8'b0000_0000 :/ exp_zero_w,
                       [8'b0000_0001:8'b1111_1110] :/ exp_reg_w,
                       8'b1111_1111 :/ exp_max_w};
    }

    constraint rand_sig_c {
        significand dist {23'h000000 :/ sig_zero_w,
                          [23'h000001:23'h7FFFFE] :/ sig_reg_w,
                          23'h7FFFFF :/ sig_max_w};
    }
```

Functions and Tasks

```
////////////////////////////////////
// External Functions and Tasks //
////////////////////////////////////

// This task will generate two operands of the type op_case_t, and calculate the
// expected value given the opcode and the signs of each operand.
task automatic generateTestCase(
    input FloatingPoint op1, op2, exp,
    input fp_case_t op1_case, op2_case, bit op1_sign, op2_sign, opcode_t opcode
    );
```

Parameterization

Results

Our best results of our testing of each unit and the full unit are follows:

ADD/SUB with Randomized Operands:	58 errors / 512K tests (0.01% fr)
ADD/SUB with Directed Randomized Corner Cases:	02 errors / 512K tests (0.0004% fr)
MULT with Randomized Operands:	Unavailable
MULT with Directed Randomized Corner Cases:	Unavailable
DIV with Randomized Operands:	61k errors / 512k tests (8-10%)
DIV with Directed Randomized Corner Cases:	23k errors / 256k tests (8%-10%)

Remaining Error Characterization

ADD/SUB - Type 1: Off by one LSB (Rounding?). Type 2: Underflow Shifted Left 1

```
# 2972400::VALUE/TYPE MISMATCH: OP1=1.575072e+31 REG. OP2=-4.253529e+37 REG. OP_CODE=SUB. EXP=4.253531e+37. RES=4.253531e+37. EXP_TYPE=REG. RES_TYPE=REG.
# EXP: 0 11111100 00000000000000000000011
# OUT: 0 11111100 00000000000000000000010
```

54 of the errors out of 512K of this type

```
# 74531100::VALUE/TYPE MISMATCH: OP1=-1.633504e-37 REG. OP2=-1.555824e-37 REG. OP_CODE=SUB. EXP=-7.767958e-39. RES=-3.780972e-39. EXP_TYPE=DENORM. RES_TYPE=DENORM.
# EXP: 1 00000000 10101001001010111101000
# OUT: 1 00000000 01010010010101111010000
```

4 of the errors out of 512K of this type

DIV - Exhibits rounding error

```
# 829596640::VALUE/TYPE MISMATCH: OP1=-1.401298e-45 DENORMMIN. OP2=-9.649204e-39 DENORM. OP_CODE=DIV. EXP=1.452243e-07. RES=1.452242e-07. EXP_TYPE=REG. RES_TYPE=REG.
# EXP: 0 01101000 00110111110111011110001
# OUT: 0 01101000 00110111110111011110000

1218517170::ERROR CODE MISMATCH: EXP_ERR=OVERFLOW. RES_ERR=NONE.
1218518530::ERROR CODE MISMATCH: EXP_ERR=OVERFLOW. RES_ERR=NONE.
```

Next Steps

- **Add/Sub** - Track down the off by 1 error. Track down the rare denorm error.
- **Mult** - The multiplier code which was not working at the time of presentation is now fixed. Slides are added below regarding the fix. The test bench doesn't exhaustively test the design and can be further improved.
- **Div** - The divider currently exhibits the rounding problem so the output displayed is always 1 bit lesser than the expected value. Pipeline the design so it facilitates easy integration with RISCV processor.
- **Full FPU** - Fully implement design with multiplication and more tests. Maybe in the future implement FPU into RISC-V Processor.

RISC-V Processor Implementation

- Parallel exploration were conducted on finding on the appropriate RISC-V core to embed our FP processor.
- Few of the cores which were considered are:
    - Steel Core (link): Steel is a RISC-V processor core that implements the RV32I and Zicsr instruction sets of the RISC-V specifications with 3 stage RISC pipeline.
    - Kronos RISC-V (link): It is again a 3 stage pipelined processor with integration an demo steps available.
    - Rocket Core (link): It is a 5 stage pipelined processor with a integer unit and option FPU. Planned to replace the existing one with our core.

References

Dawson, Bruce. “Exceptional Floating Point”. Apr 2012. Random Ascii Tech Blog. https://randomascii.wordpress.com/2012/04/21/exceptional-floating-point/

Fog, Agner. Floating point exception tracking and NAN propagation. Technical University of Denmark. Apr 2020. https://www.agner.org/optimize/nan_propagation.pdf

“Introduction to Floating Point Arithmetic | Rounding using Guard, Round and Sticky bits [English]”. https://www.youtube.com/watch?v=4VU7ctvMRfs

Miller, Karen. “Floating Point Representation”. http://pages.cs.wisc.edu/~markhill/cs354/Fall2008/notes/flpt.apprec.html.

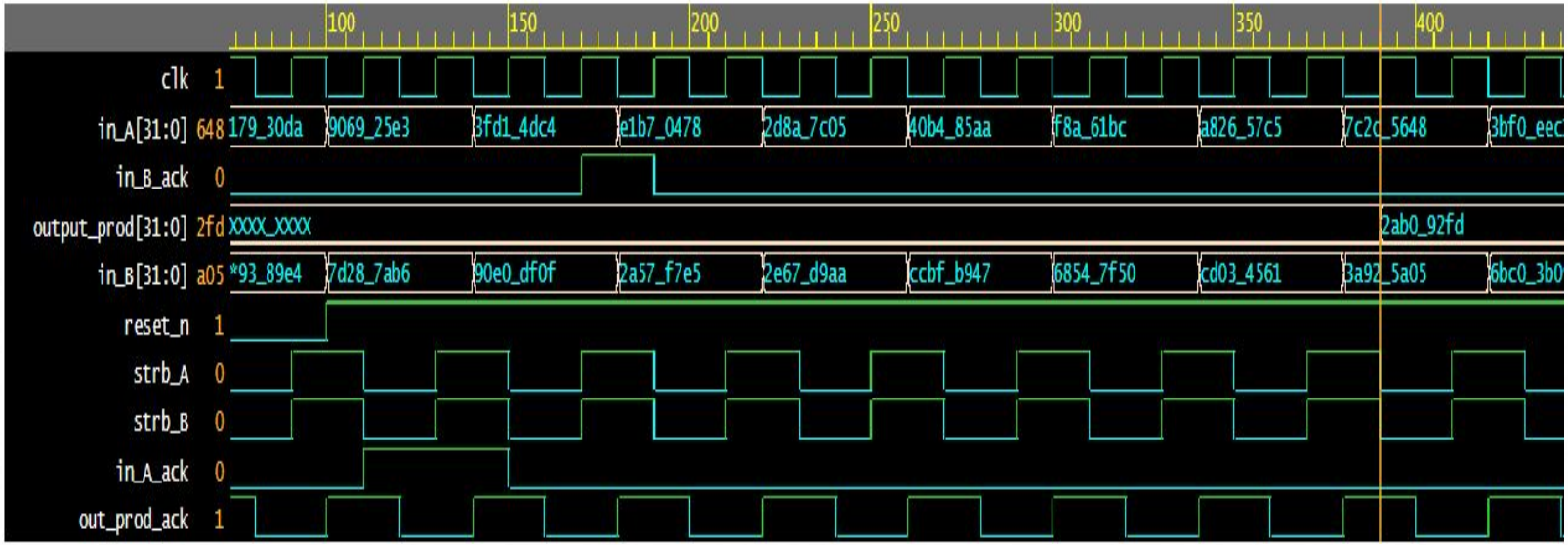
Parhami, Behrooz. “Number Representation and Computer Arithmetic”. Encyclopedia of Information Systems. 2003.

Shanthi, A.P. “Floating Point Arithmetic Unit”. Computer Architecture. https://www.cs.umd.edu/~meesh/411/CA-online/chapter/floating-point-arithmetic-unit/index.html

Fixed the multiplier

The problem was in the testbench and not the design. I had accidentally added an always in front of the reset which means the reset was toggling along with the clock. I missed observing this initially as I didn't pull the reset signal . Hence the multiplier was reset after every cycle and showed an **X** at the output. Also , I had asserted the output acknowledge without any delay(same time as the input). I have fixed the testbench and the design is working as expected now.

Post presentation Fix



The product in binary is 0___01010101____01100001001001011111101