1 Introduction

1.1 Purpose of the Project

The purpose of this project is to explore how floating point numbers are operated on in a CPU, as well as how different parts of the CPU signal each other to pass information and commands between different parts. Most FPGAs have their FPU blocks optimized where the code is no longer readable, and understanding how they are implemented at a fundamental level in verilog will also allow the group to write and conceptualize better verilog. With this project, the implementation will also require me to gain practical experience with either System Verilog or Verilog, allowing me to use those in other projects in the future.

1.2 Features of the Project

The two features of the current version of the project are:

- All RISCV-I ALU operations
- FPU Addition
- FPU Subtraction (Incomplete)

2 Design Overview

The main design of the FPU is a simple ALU which (currently) supports two operations of the floating point variety, i.e. adding and subtraction. The pipeline of the testing to building of the FPU is first it is written in Verilog, test benches are executed to verify its functionality, with a special focus on edge cases. All the verilog is then checked to make sure that it is synthesizeable, before it is then built onto the FPGA using Yosys and Nextpnr.

3 Implementation Details

3.1 Floating point IEEE-754 Representation

Floating point notation in the IEEE standard is taking advantage of scientific notation to represent numbers both extremely large and small using the same number of bits in the same configuration. This is done by using the following representation:

- 1 bit at the beginning to represent sign
- 8 bits after to represent the exponent, which is the 8 bit value minus 127
- 23 bit mantissa at the end, representing the actual value of the number.

An example number would be as follows: 0_01111111_11000000000000000000000 This is the number 1.75, since we have 0 bit 1, i.e. positive, 127 for the exponent, meaning $2^{127-127} = 2^0$, and a mantissa of 110000... when combined with a hidden bit, yields $1.11 * 2^0$ or 1.75

3.2 Algorithm for addition and subtraction

Simply put, the algorithim is as follows:

- Pull both numbers' hidden bits in
- Denormalize number with smaller exponent, i.e. make the two numbers have the same exponent
- Negate numbers which are negative
- Add or Subtract (Add two's complement) the numbers
- Renormalize i.e. shift if there is a carry and add to the exp
- Put the sign, new exponent and mantissa into the output

3.3 Implementation in Verilog

The implementation of the RISCV-I ALU was trivial, and only used the unoptimized verilog arithmetic elements, and so will not be included in this report. (If curious it will be on git.joshuayun.com along with the FPU). The FPU was implemented using only assign statements, as it is a combinational logic only circuit in most cases, however this does introduce much more complexity into the design, as only ternary operators can be used in assign statements which need choice logic.

3.4 Testbench

The testbench was quite naively implemented, with the group simply trying multiple test cases sequentially without a loop due to the fact that the bench was used mostly as a way to debug the output, rather than a true test of functionality: the code can be found in the appendix or on my git repository (soon @ git.joshuayun.com)

3.5 FPGA

The verilog was all synthesized and put onto the Orangecrab using the yosys nextpnr toolchain, however there was not sufficient time to test the true functionality of the FPU on an actual FPGA due to an inability for data to be taken out or display from the FPGA, nor could data be put into the FPGA.

4 Results

The verilog on the FPU in the icarus verilog simulator is passing all of the given tests, however it does not fully have NaN, infinity, nor zero functionality. However, more tests on niche cases, i.e. larger + smaller numbers and certain signed operations may be dysfunctional at the moment.

5 Problems and Challenges

5.1 Design Problems

The only major issue with designing and planing out the mechanics of the FPU was the issue of not having the proper knowledge on how exactly to best implement these algorithms, especially when it came to try and deal with adding negative numbers or subtracting positive ones, as it was often not clear how to properly represent negative numbers internally in the FPU, as well as when then carry is used for determining sign or carrying the exponent higher.

5.2 Debugging

The major problem encountered in this project mostly dealt with understanding the actual algorithms at the lower levels that the FPUs use, as well as how they would be synthesized in verilog. Having a small amount of verilog experience meant that the second issue was mitigated, but still was a problem in that there were often many smaller mistakes that were difficult to find which significantly changed the output of the FPU. It was also difficult having consistent debugging information in the FPU at all times, mostly due to the fact that since the FPUs is a series of combinational logic with assign statements, internal displays could not be used on the ternary operators to determine whether the branching condition was correct or not.

6 Future Plans

The main future plan for this project is the first add most of the NaN and zero functionality to this FPU, as it is lacking even the most basic of operation support other than two non-zero integers. The next obvious step would then be to implement division and multiplication using the FPU, and this would most likely along with significant amounts of debugging be the completion of the FPU. Future plans will also include incoroporating this into a RISCV-I processor that I plan to design and synthesize onto an FPGA, along with enough modifications to the control logic and FPU to make it RISCV-IF compliant as well.

7 References

Martin, R., 2021. Computer Organization and Design Unit 7: Floating Point. [online] Cis.upenn.edu. [Accessed 12 December 2021].

8 Appendix

8.1 Verilog FPU

```
`include "exp_calc.v"
module fpu_2(
input wire add_not,
input wire[31:0] a_in, b_in,
output wire[31:0] out
);

wire[23:0] a_sig, b_sig, b_shft_sig, a_shft_sig, a_sign_sig, b_sign_sig;
wire[24:0] sig_sum, sig_diff, sig_op, sig_final;
wire[7:0] exp;
wire[7:0] diff, neg_diff;
wire same_sign;

assign diff = a_in[30:23] - b_in[30:23];
assign neg_diff = b_in[30:23] - a_in[30:23];
assign exp = diff[7] ? b_in[30:23] : a_in[30:23];

assign same_sign = ~(a_in[31] ^ b_in[31]);

// Pull hidden bit into sig, if exp 0, no hidden bit
assign a_sig = (|a_in[30:23] ? {1'b1, a_in[22:0]} : {1'b0, a_in[22:0]});
assign b_sig = (|b_in[30:23] ? {1'b1, b_in[22:0]} : {1'b0, b_in[22:0]});

assign a_shft_sig = (diff[7] ? a_sig >> neg_diff : a_sig);
assign b_shft_sig = (diff[7] ? b_sig : b_sig >> diff);

//2C Invert if Negative and not same signs
assign a_sign_sig = same_sign ? a_shft_sig :
(a_in[31] ? ~(a_shft_sig) + 24'b1 : a_shft_sig);
assign b_sign_sig = same_sign ? b_shft_sig :
(b_in[31] ? ~(b_shft_sig) + 24'b1 : b_shft_sig);

//Adding
assign sig_sum = a_sign_sig + b_sign_sig;
/* assign sig_sum = a_shft_sig + b_shft_sig; */
//Subtraction
assign sig_diff = a_shft_sig + ~(b_shft_sig) + 25'b1;

assign sig_op = add_not ? sig_diff : sig_sum;
assign sig_final = sig_op[24] | same_sign ? sig_op : ~(sig_op) + 24'b1;
```

```
// Assign exp and mantissa
assign out[31] = a_shft_sig > b_shft_sig ? a_in[31] : b_in[31];
assign out[30:23] = sig_sum[24] & same_sign ? exp + 8'b1 : (same_sign ? exp : exp);
assign out[22:0] = sig_final[24] & same_sign ? sig_final[23:1] : sig_final[22:0];

//assign out = {sig_sum, 7'b0};
// assign out = {diff, neg_diff, 16'b0};
/* assign out = {sig_op, 7'b0}; */

endmodule
```

8.2 Verilog Testbench

```
`timescale 1us/1ns

`include "fpu_2.v"

module fpu_bench;

reg[31:0] input1, input2;
reg add = 1'b1;
wire[31:0] fpu_output;

fpu_2 fpu0 (add,input1, input2, fpu_output);

initial begin
input1=32'b00111111010000000000000000000000; // -.75
input2=32'b10111111110000000000000000000000; // -1.5
#5;
$display("\nSum: %16b + %16b\n  =  %16b",input1,input2,fpu_output);

input1=32'b00111110100000000000000000000000; //.25
input2=32'b00111111010000000000000000000000; // .75
#5;
$display("\nSum: %16b + %16b = %16b",input1,input2,fpu_output);

input1=32'b01000000000111001100110011001101; // 2.45
input2=32'b00111111001001100110011001100110; //.65
#5;
$display("\nSum: %16b + %16b\n  =  %16b",input1,input2,fpu_output);

input1=32'b01000000000111001100110011001101; // 2.45
input2=32'b10111111001001100110011001100110; //.65
#5;
```

```
$display("\nSum: %16b + %16b\n  =  %16b",input1[22:0],input2[22:0],fpu_output[31:8]);


$finish;
end
endmodule
```