Final Project Report: Floating Point Addition and Subtraction

EE 4243

Computer Organization and Architecture

Fall 2019

Names: Fabricio Zuniga, Aaron Cantu, Hugo Menendez

33.33%, 33.33%, 33.33%

TABLE OF CONTENTS

SECTION		PAGE NUMBER
I.	INTRODUCTION	2
II.	PROBLEM STATEMENT	2
III.	PROCEDURE	3
IV.	DESIGN METHEDOLOGIES	5
V.	TESTING METHODS	6
VI.	TROUBLESHOOTING	6
VII.	RESULTS	7
VIII.	CONCLUSION	9
IX.	CODE	10
X.	SOURCES	14

I. INTRODUCTION

Throughout the semester, we individually implemented several modules into our Vivado projects, written in Verilog, to create a basic RISC-V CPU. This simple CPU includes the arithmetic logic unit (ALU), data memory, register file, instruction decoder, program counter, immediate generator, multiplexers, and instruction memory. The control signals are represented by the instruction decoder and the virtual input/output (VIO). The final project for this course utilizes our knowledge of RISC-V architecture and Verilog. We're able to now use our current understanding and code to implement additional functionality into our existing design.

II. PROBLEM STATEMENT

The group was assigned to incorporate floating point addition and subtraction arithmetic onto the existing RISC-V CPU that was created throughout the labs. This would require two source registers to hold the two floating-point numbers and a destination register for the result of the operation. For this project, the group decided to abide by the IEEE-754 single precision floating-point standard.

III. PROCEDURE

- Research/organization
  - IEEE-754
    - RISC-V ISA Manual
    - Course Textbook(Computer Organization and Design The Hardware/Software Interface: RISC-V Edition by Patterson/Hennessy)
    - H-shmidt.net floating point converter
- Implementation
  - Writing 32-bit instruction codes
  - Arithmetic module
    - Procedure of floating point addition and subtraction
      - 1. Identify the exponent, mantissa, and sign bit for the inputs A and B (32 bits)
      - 2. Calculate the difference of exponents (greater - smaller)
      - 3. Shift right the mantissa of the smaller exponent by the difference
      - 4. Calculate the sum or difference of the two mantissas (using 2's complement for negative values)
      - 5. Check if the sign bit of the mantissa is negative (use 2's complement if necessary)
      - 6. The output will be a concatenation of the sign bit, the bigger exponent between A and B, and the sum of mantissa A and mantissa B
    - Case statements to differ FADD.S and FSUB.S (based on funct3[000/111])
    - Modify the instruction decoder for FADD.S and FSUB.S
    - Incorporate the instruction codes into the instruction memory module
  - Use the floating-point arithmetic module inside the ALU

The first step was to research IEEE-754 floating point standard. The group only focused on the single precision standard since the CPU has a 32-bit ISA. For reference, the RISC-V ISA Manual provided by riscv.org was used as well as the textbook for this course: Computer Organization and Design, The Hardware/Software Interface: RISC-V Edition by Patterson/Hennessy.

Since the only operations to be implemented were floating point addition and subtraction, they would be implemented as R-type instructions. Similarly to the normal addition and subtraction instructions, these instructions contain fields for rs1, rs2, rd and the opcode (1010011). The only difference is that for FADD/FSUB instructions, they carry a funct5, fmt, and rm. Funct5 is used to specify the operations for FADD and FSUB, fmt is used to specify single-precision or double-precision and rm is an input register. In this case, since the group chose not to strictly abide by the actual RISC-V floating point instructions, funct5 and fmt are left as '0000000' and rm will be used to differentiate the operation between FADD and FSUB with '000' being associated with FADD and '111' with FSUB. This allowed for a seamless integration of these instructions into the design, without having to modify the instruction decoder.

Having the instruction format ready, instruction codes were written that would be stored within the instruction memory. The instructions were initially written in RISC-V assembly language and then translated to binary and then to hexadecimal. The website rapidtable.com was used to verify the binary code for the instructions was correctly converted to hexadecimal.

After implementing the instruction code, the ALU was modified by instantiating a module called "f_arithmetic" that would compute the two necessary floating-point operations. The output of that module was then added to the case statement for the select line. In order to distinguish between FADD and FSUB within the instantiation, an additional wire called "addSub" was created. This wire goes high when the select line chooses FSUB and low for FADD.

The most difficult implementation for this design was the f_arithmetic module because it was designed to compute both FADD and FSUB operations. However, it was decided not to include any floating point rounding as there were too many issues incorporating it into this module. In order to compute this addition and subtraction, it was quickly made apparent that many bit-selects and shifting would need to be performed. Thus, regs were made to store the sign bits, exponents, and mantissas of the two inputs A and B respectively. Once done, a combinational always block was made to compute all of the required calculations. First, the exponents of A and B were compared and their difference was calculated. The input with the larger exponent was left alone while the other input's mantissa was right-shifted by the difference. Then, the sign bits of both inputs were checked to see if the numbers are negative. If they are negative, the mantissas would be converted using 2's complement. After this, the addSub value is checked which determines if the mantissas are added or subtracted. The sum of the mantissas is then normalized and/or converted using 2's complement if necessary. The final part of this always block then performs checks for special cases such as zero and NAN. Finally, the module ends with the output F being a concatenation of the new sign bit, exponent, and mantissa sum.

As for the actual floating-point values, these were added to the registers using add instructions and the VIO just as it was done in lab 8. These values were decided upon as integers and then converted to IEEE-754 single precision using the website h-schmidt.net.

IV. DESIGN METHOLDOLOGIES

1. Design Specification

Before any sort of implementation, a design must be proposed that follows the goal or purpose of the project. The design specifications may include the number of Configurable Logic Blocks (CLBs), the size of memory such as block RAM (BRAM), and the number of Lookup Tables (LUTs). This is important to note since knowing the design specifications will allow the planner to decide which Field Programmable Gate Arrays (FPGAs) is best suitable based on those specifications.

2. Hardware Planning

This stage requires research of a variety of FPGAs and considers the specifications to select the most suitable choice based on the project's design specifications. In this case, our FPGA was the Digilent Nexys 4 DDR equipped with an Artix-7 chip. The FPGA uses 36Kbits of BRAM/FIFO with each CLB containing 8 flip flops, 8 flip flop/latches and 8 6-input look up tables. The specifications of this FPGA were suitable for this project.

3. Create Hardware Description Language (HDL) Code/issue constraint file

HDL is created for the FPGA based on what the design entails. This includes using an IDE to interface with the FPGA. In this case, Vivado is used as the IDE of choice and Verilog is the HDL used in this project. The HDL code in this project consists of a combination of behavioral modeling and data flow modeling. There is a top module that ties everything together (the CPU) and various sub-modules that perform specific tasks.

4. Synthesis

The completed HDL code is converted into a gate-level netlist as a circuit that consists of building blocks or macros. With Vivado, a schematic is created representing the pre-optimized design at the Register Transfer Level (RTL), which can help with possible debugging.

5. Implementation

The netlist is converted into a map that consists of fitting the design based on the available resources of the FPGA. The translated file will create the placement of pins and routing onto the FPGA.

6. Generate Bitstream

The Implementation is converted into a file (bitstream) that can be loaded onto the FPGA. This stage will test the functionality of the design and see if it meets project goals.

7. Simulation

Creating a testbench that will simulate the design with desired input values to test the functionality of the design. The extended modules can be tested individually as well as the CPU with the modules implemented. The simulations can be tested using waveforms that are generated, which are great tools for debugging.

V. TESTING METHODS

There were 2 testing methods involved for this project.

Using Virtual Input/Output module to cycle through the pipeline.

A bitstream was generated and the FPGA was programmed. Since the VIO was not modified from lab 8, it was used in the same way. To clarify, AND operations were used to clear the registers to be used, ADD operations and rd1_alt to inject the floating point numbers into the registers, and then FADD and FSUB to verify their functionality. Using this method, it is easy to see if the sums are being calculated and stored correctly.

Test Bench

In order to test the functionality of the ALU module, and therefore simultaneously testing the f_arithmetic module, a test bench was written to create a simulation. Both FADD and FSUB were tested and verified to work correctly. The results of this simulation can be viewed in the **Results** section of the report.

VI. TROUBLESHOOTING

During the testing process, there were some issues with the instruction codes. As the instructions were reviewed, the group realized that the codes were either shorter or longer by 1 bit and miswrote values for the instruction in certain parts. This required rewriting and verifying that they were all correct multiple times.

In the Verilog code, there was debugging involved that was a result from synthesis and implementation errors that prevented the project from operating. This involved looking for any syntax errors. The code contained data types that were used incorrectly such as in always blocks where the wire type was used instead of a reg type on the left hand side of a procedural statement. Inside the always blocks, the blocking and non-blocking assignments were used incorrectly, that resulted into issues with a combinatorial loop that prevented implementation. Finally, there was also the process of comprehending the IEEE-754 floating point standard for single precision. As part of the research process, it involved having to make sure the binary was written correctly based on the format that was used.

VII. RESULTS

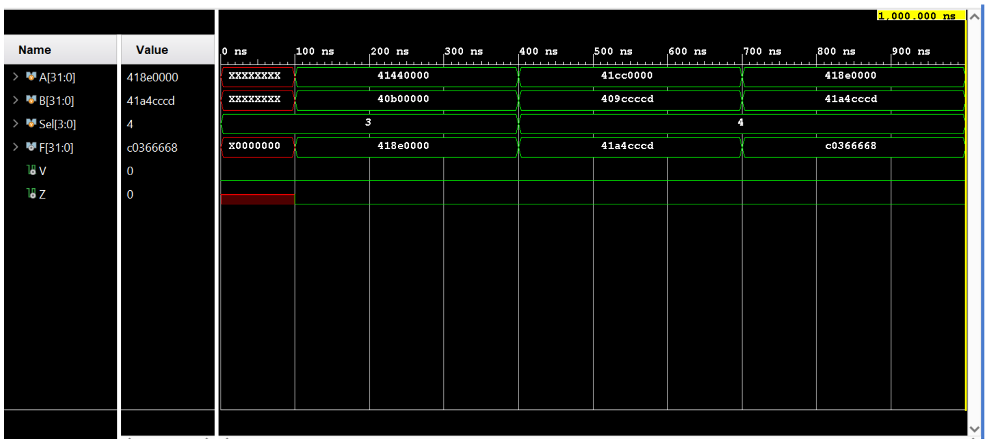


Figure (a).

Simulation waveform testing ALU for FADD and FSUB

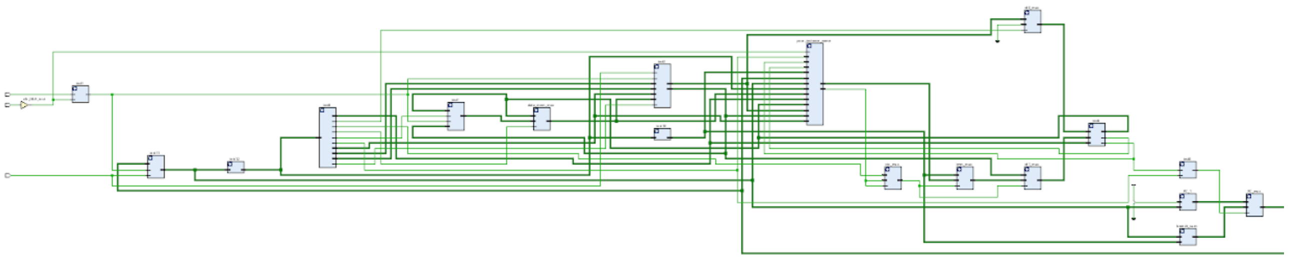


Figure (b).

RTL schematic of CPU

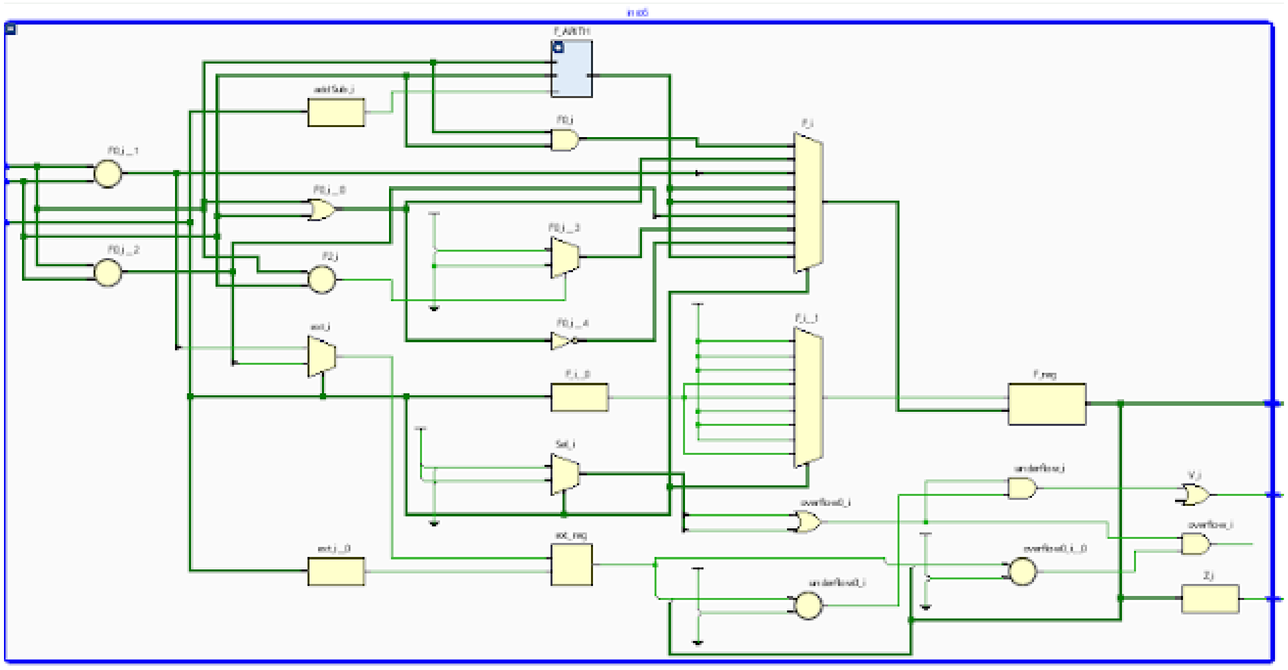
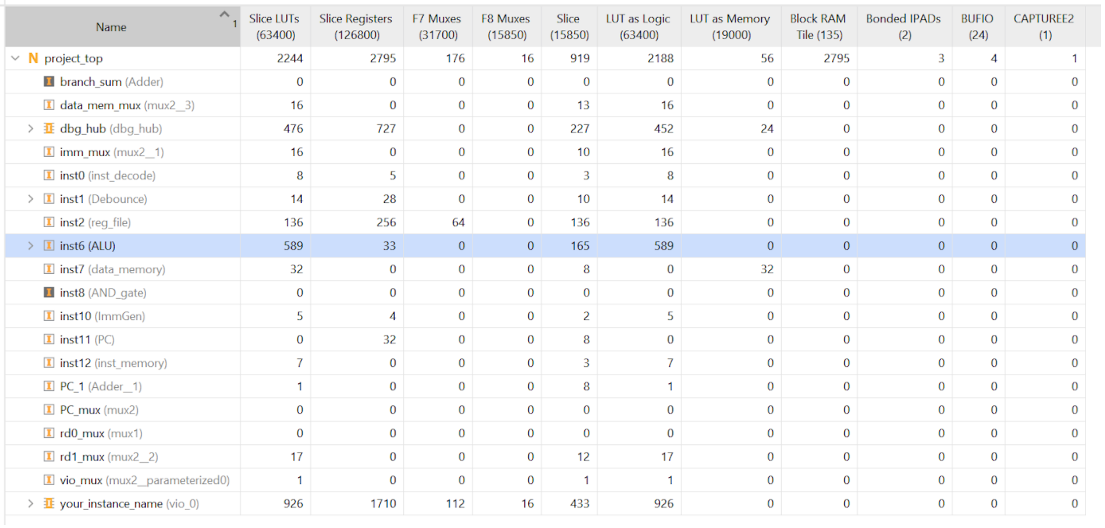


Figure (c).

RTL schematic of the arithmetic logic unit with the 'f_arithmetic' module inside



Name	Slice LUTs (63400)	Slice Registers (126800)	F7 Muxes (31700)	F8 Muxes (15850)	Slice (15850)	LUT as Logic (63400)	LUT as Memory (19000)	Block RAM Tile (135)	Bonded IPADs (2)	BUFIO (24)	CAPTUREE2 (1)
project_top	2244	2795	176	16	919	2188	56	2795	3	4	1
branch_sum (Adder)	0	0	0	0	0	0	0	0	0	0	0
data_mem_mux (mux2__3)	16	0	0	0	13	16	0	0	0	0	0
dbg_hub (dbg_hub)	476	727	0	0	227	452	24	0	0	0	0
imm_mux (mux2__1)	16	0	0	0	10	16	0	0	0	0	0
inst0 (inst_decode)	8	5	0	0	3	8	0	0	0	0	0
inst1 (Debounce)	14	28	0	0	10	14	0	0	0	0	0
inst2 (reg_file)	136	256	64	0	136	136	0	0	0	0	0
inst6 (ALU)	589	33	0	0	165	589	0	0	0	0	0
inst7 (data_memory)	32	0	0	0	8	0	32	0	0	0	0
inst8 (AND_gate)	0	0	0	0	0	0	0	0	0	0	0
inst10 (ImmGen)	5	4	0	0	2	5	0	0	0	0	0
inst11 (PC)	0	32	0	0	8	0	0	0	0	0	0
inst12 (inst_memory)	7	0	0	0	3	7	0	0	0	0	0
PC_1 (Adder__1)	1	0	0	0	8	1	0	0	0	0	0
PC_mux (mux2)	0	0	0	0	0	0	0	0	0	0	0
rd0_mux (mux1)	0	0	0	0	0	0	0	0	0	0	0
rd1_mux (mux2__2)	17	0	0	0	12	17	0	0	0	0	0
vio_mux (mux2__parameterized0)	1	0	0	0	1	1	0	0	0	0	0
your_instance_name (vio_0)	926	1710	112	16	433	926	0	0	0	0	0

Figure (d).

Report Utilization of RISC-V CPU with 'f_arithmetic' module

The simulation waveform provided in the figure (a) shows the functionality of the f_arithimetic module through the ALU module. In the testbench code, the ALU is instantiated since the f_arithmetic is located within the ALU. The results of the waveform show that fadd was able to add the floating-point numbers 12.25 and 5.5 to output 17.75. fadd can subtract 4.9 from 25.5 to output 20.6 and subtract 20.6 from 17.75 to output -2.85. The same values were tested in the VIO and the same results were gathered.

VIII. CONCLUSION

Based on the results, we were able to implement a floating-point arithmetic module within the ALU that can perform floating-point addition and subtraction using the IEEE-754 single precision standard. From the teachings of this course, successful implementation of an extension onto the RISC-V CPU was accomplished. As part of this experience, we can preserve this project and implement any RISC-V extensions for future plans and continue to develop knowledge in FPGA design which includes writing in Verilog and FPGA functionality. Lastly, this project will help us greatly in standing out in both job applications and internships, which is highly beneficial to our future careers.

IX. CODE

Source code for ALU module with floating arithmetic module

```
module ALU

#(parameter w = 8)

(A, B, Sel, F, Z, V);



input [w-1:0] A, B;

input [3:0] Sel;

output [w-1:0] F;

output V, Z;



wire overflow;

wire underflow;

reg ext;

reg [w-1:0] F;

wire addSub;



initial

   begin

      ext = 0;

   end



assign overflow = (((Sel == 4'd2) || (Sel == 4'd6)) &&
{ext, F[w-1]} == 2'b01) ? 1:0;

assign underflow = (((Sel == 4'd6) || (Sel == 4'd2)) &&
{ext, F[w-1]} == 2'b10) ? 1:0;

//Underflow change

//assign V = ((overflow) || (underflow)) ? 1:0;
```

```
begin

   if(A[30:23] < B[30:23])

   begin

      c = B;

      d = A;

   end

   else

   begin



       c = A;

       d = B;

   end



   sign_bit_A = c[31];

   sign_bit_B = d[31];

   exponent_A =c[30:23];

   exponent_B =d[30:23];

   mantissa_A = {2'b00, (exponent_A) ? 1'b1 : 1'b0,
c[22:0]};

   mantissa_B = {2'b00, (exponent_B) ? 1'b1 : 1'b0,
d[22:0]};

   difference = exponent_A - exponent_B;

   mantissa_B = mantissa_B >> difference;



   //Compare the sign bits of both

   if(sign_bit_A)
```

```
wire [w - 1 : 0] f_out;

assign addSub = (Sel == 4'h4) ? 1 : 0;

f_arithmetic F_ARITH(A, B, addSub, f_out);


always @ (*)

  begin

    case (Sel)

      4'h0:

        F <= A & B;

      4'h1:

        F <= A | B;

      4'h2:

        //F = A + B;

        {ext, F} <= {A[w-1], A} + {B[w-1], B};

      4'h3://fadd

         F = f_out;

      4'h4://fsub

         F = f_out;

      4'h6:

        //F = A - B;

        {ext, F} <= {A[w-1], A} - {B[w-1], B};

      4'h7:

        F <= (A < B) ? 1:0;

      4'hC:

        F <= ~(A | B);

    endcase
```

```
  mantissa_A = -mantissa_A;

if(sign_bit_B)

  mantissa_B = -mantissa_B;


//Compute the sum

if(addSub)

  mantissa_sum = mantissa_A - mantissa_B;

else

  mantissa_sum = mantissa_A + mantissa_B;


//Check sign of the sum

sign_bit_out = mantissa_sum[25];


if(sign_bit_out)

  mantissa_sum = -mantissa_sum;


//Normalize the sum (special cases)

 if(mantissa_sum[24])

 begin

  exponent_out = exponent_A + 1;

  mantissa_sum = mantissa_sum >> 1;

 end

 else if(mantissa_sum)

 begin

  position = 0;

  for(i = 23; i >= 0; i = i - 1)

    if(!position && mantissa_sum[i])
    end

assign Z = (F == 0) ? 1:0;

assign V = ((overflow) || (underflow)) ? 1:0;


endmodule


module f_arithmetic (A,B,addSub,F);

input [31:0] A, B;

input addSub;

output [31:0] F;


reg [7:0] exponent_A, exponent_B;

reg [7:0] difference;

reg [7:0] exponent_out;


reg sign_bit_A, sign_bit_B;

reg sign_bit_out;


reg [25:0] mantissa_A, mantissa_B;

reg [25:0] mantissa_sum;


integer position, adjust, i;


reg [31:0] c, d;


always @ (*)
```

```
                position = i;

            adjust = 5'd23 - position;

            if(exponent_A < adjust)

            begin

                exponent_out = 0;

                mantissa_sum = 0;

                sign_bit_out = 0;

            end

            else

            begin

                exponent_out = exponent_A - adjust;

                mantissa_sum = mantissa_sum << adjust;

            end

        end

        else

        begin

            exponent_out = 0;

            mantissa_sum = 0;

        end


end


assign F = {sign_bit_out, exponent_out,
mantissa_sum[22:0]};


endmodule
```

Testbench code for ALU

```
module ALU_test_tb();
reg [31:0] A, B;
reg [3:0] Sel;
wire[31:0] F;
wire V, Z;

ALU_test #(32) test(A, B, Sel, F, Z, V);

initial
begin
   Sel = 4'h3;
   #100;
   A = 32'h41440000;
   B = 32'h40b00000;
   #300;

   Sel = 4'h4;
   A = 32'h41CC0000;
   B = 32'h409CCCCD;

   #300;
   Sel = 4'h4;
   A = 32'h418E0000;
   B = 32'h41A4CCCD;
end

endmodule
```

X. SOURCES

"Binary to Hex Converter." *RapidTables* , www.rapidtables.com/convert/number/binary-to-hex.html.

"IEEE-754 Floating Point Converter." *Tools & Thoughts*, www.h-schmidt.net/FloatConverter/IEEE754.html.

Louisiana State University Electrical and Computer Engineering Department. "LSU EE 3755 -- Fall 2012 -- Computer Organization." Verilog Notes 8 -- Floating Point. 1 Dec. 2019, www.ece.lsu.edu/ee3755/2012f/l08.v.html.

Lundgren, David. "Double Precision Floating Point Core Verilog." *Opencores.org*, opencores.org/websvn/filedetails?repname=double_fpu&path=%2Fdouble_fpu%2Ftrunk%2FDouble_FPU.PDF.

Patterson, David A., and John L. Hennessy. *Computer Organization and Design: the Hardware/Software Interface*. Morgan Kaufmann Publishers, an Imprint of Elsevier., 2018.

Waterman , Andrew, et al., editors. *The RISC-V Instruction Set Manual*. Vol. 1, CS Division, EECS Department, University of California, Berkeley, 2017.