Floating Point Unit

Noel Esqueda, Winston Do

Abstract

Floating Point Units (FPU) are integral to performing more precise mathematical operations. Our objective was to implement a Floating-Point Unit in Verilog that is capable of performing mathematical operations similar to an ALU but with floating-point values. The FPU is designed to perform addition and multiplication on single precision 32-bit IEEE 754 floating point values. The FPU takes two 32-bit values and depending on the control signal, multiplies or adds them. It was initially planned to have the FPU be capable of addition, multiplication, subtraction and division. However, there was significant difficulty in the implementation of division and subtraction, so the scope of the FPU was reduced. A testbench was written for the FPU to have it add and multiply six floating point values. It was found that the FPU is capable of adding and multiplying two positive floating-point values. As the addition and multiplication process requires shifting bits, some precision from the initial 32-bit floating point values were lost to rounding.

1. Index terms/Keywords

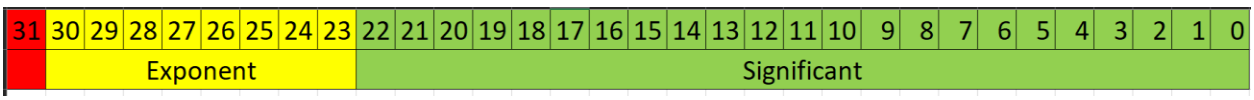
Floating Point Unit, IEEE 754, Floating Point, **Significand, Precision, 32-bit, Exponent, Precision**

2. Introduction

Most computer processors have ALUs to perform mathematical operations on integer values. However, in real world applications, decimal numbers are required to perform precise mathematical calculations. There are various ways to encode decimal numbers in binary. Early computers had their own method of encoding and performing operations with floating point value. These methods could differ from processor to processor. Modern processors, including this FPU, all use the IEEE 754 Floating Point Standard.

32-bit IEEE 754 Floating Point

The IEEE 754 Floating Point Standard uses a 32-bit value to encode a decimal number. There are 64 bit encoding methods, but the 32-bit one is the most common and the onc used in this FPU. The IEEE 754 uses a standard similar to scientific notation and splits a 32-bit value into its component pieces. The most significant bit or first bit is used to encapsulate the sign of the decimal value. The next byte is referred to as the exponent. This encodes the position of the decimal value. The final 23-bits are used to encode the actual value of the decimal number.



As floating point values are encoded in binary in a different manner than normal signed and unsigned integer values, there needs to be dedicated hardware to perform mathematical operations on them. Having the FPU integrated into the CPU allows it to calculate non-integer operations faster therefore freeing up time for other operations the CPU may need to compute. For older CPUs that didn't have FPUs

integrated in them, FPU's could be implemented as a microprogram within the CPU. Our goal when creating the Floating Point Unit was to add as many of the arithmetic operations in our Verilog program as possible and gain a better understanding of their inner workings. In the following section, we will be discussing the specifications of our program such as the inputs, outputs, behavior, and a labeled diagram . The following sections will discuss the procedure and tools used in this project, the results of our program, any possibilities for the future, and lastly our conclusion.

3. Specifications

The FPU takes a total of three inputs, two of them 32-bits long and one input consisting of a single bit width, which serves as a control signal. The FPU outputs a single 32-bit output, which can be referred to as the result of the FPU's mathematical operation. The FPU will add if the control signal is low and will multiply if the control signal is high. Internally the FPU consists of two modules, referred to as the Floating Point Addition Unit (FPAU) and the Floating Point Multiplication Unit (FPMU).

Floating Point Addition Unit (FPAU)

The FPAU is the module responsible for adding two floating point values. It consists of two 32-bit inputs and a single 32-bit output. The FPAU's first task is to determine the resulting sum's sign by performing an OR operation between the first bits of the floating point values. Next the FPU needs to determine which of the 32-bit operands are larger. It then outputs the largest float value on a wire designated as operand A. The smaller of the two values is placed on the operand B wire. So operand A is the larger float value and operand B is the smaller float value. Now each of the components of the operands are split into their constituent parts: the sign bit, the exponent byte and the significant 23-bit field. When adding two floating point values, the decimal position of the significants must match. This process is called normalization. The FPU will modify the significant of the smaller operand so that the decimal positions match. The FPU does this by shifting the operand B by the difference between the exponents. The example below shows the operation in action.

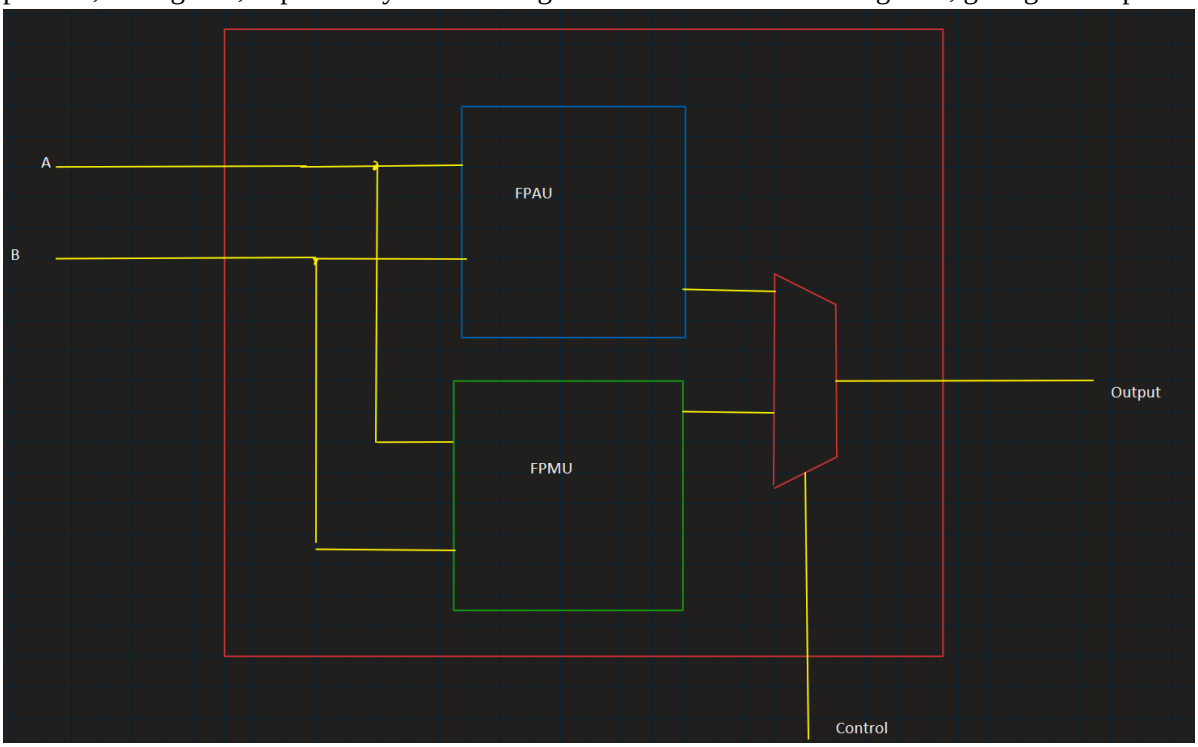
$$x.xxxx \times 2^{215} + y.yyyy \times 2^{209} => x.xxxx \times 2^{215} 1.00000yyyyy \times 2^{209}$$

The example shows that the secondary operand, B, was shifted by 215-209 = 6 decimal places. In addition, the 24-bit significants must be appended by a 1 if the initial 32-bit value is positive or a 0 of the value is negative. Once the significand of the B operand has been normalized, the FPU can now add the two significants. This value is then combined with the exponent byte of operand A and the sign of the sum of the two floating point values.

Floating Point Multiplication Unit (FPMU)

The multiplication module is similar to the addition module in input and output. It consists of two 32-bit inputs and a single 32-bit output. Multiplication of floating point values is much simpler than addition. There is no need for the position of decimal points of the two values to match. Therefore, the exponent bytes of the two operands are added together. The sum of the exponents need to be biased and are added with 127. The sign of the product is determined by an XOR operation on the first bit of the two floating values. The significants of both values are multiplied together. To normalize the significant after multiplication, the first bit is extracted from the product of the two significants. It is then used to

determine whether or not to round the values after multiplication. Finally, all three components of the product, the sign bit, exponent byte and the significant are concatenated together, giving us the product.



4. Procedure and Tools

The first thing we did as we approached this project was to get a better understanding of how floating-point numbers are calculated using addition for single precision. If two numbers have the same exponent then the significands can be added together. In the case where two numbers have different values for their exponents, we must first find the difference between exponents. The number with the smaller exponent is then shifted by the difference in exponents to the right. After these two steps, the exponents are now the same so the numbers can now be added together. This is how it is done mathematically and done similarly in computer programming.

In programming the most significant bit represents the sign bit, followed by 8-bits representing the exponent, and lastly the remaining 23-bits represent the significand of the number. The significand has a leading one value acting as the 24th bit which acts as the bit in front of the decimal point while the rest of the significand follows after the decimal point. In the case when the exponents are the same, as done mathematically, we just add the significands together. If the leading 1 bit increases to multiple bits after addition, then we must shift the value to the right until we only have 23 bits as the significand. This causes the exponent to increase as well as lose precision due to the fact that the bits that are shifted beyond the 23rd bit are lost since there is no longer any available space to store those bits. In the case when exponents are not the same, we have to normalize the number with the lower exponent to the higher exponent by shifting the significand by the difference in their exponents. When you shift the significand, you fill in the open bits with zeroes. Then once normalization is complete, you may proceed with the addition. For addition of numbers with drastic differences in exponents, the number with the large exponent will stay the same since the other number will have a negligible impact in the operation.

Knowing how floating-point addition works logically, we then set to write our program in Verilog. In this project we used Modelsim PE Student Edition provided to us through this course to implement and

simulate the logic of our Floating-Point Unit. The Modelsim program allows us to prepare a source file, edit and compile it, and simulate the compiled version.

- Editing and Compilation
- Simulation

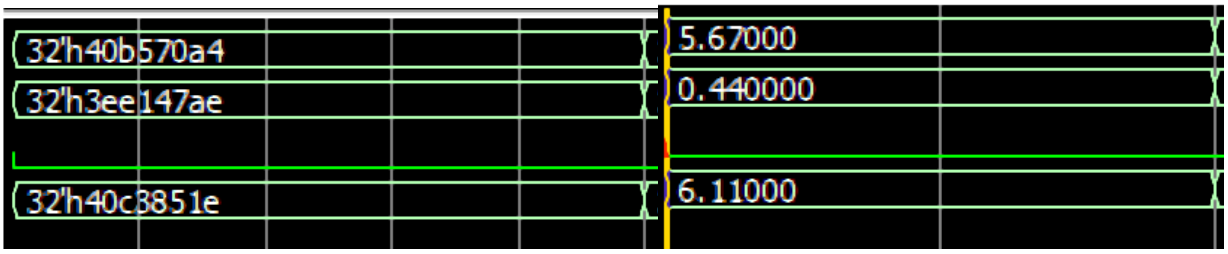
5. Results and Discussion

The FPU was able to perform multiplication and addition successfully. The FPU was limited to positive 32-bit floating point values. A test bench was written for it to have it multiple and add three values.

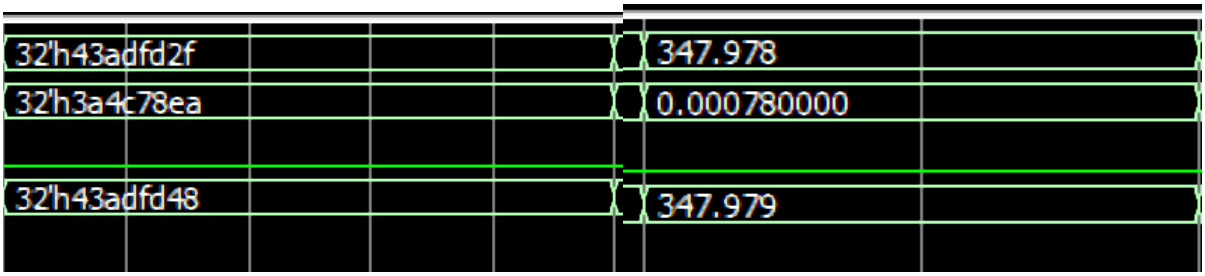
Test Operand A Decimal	Test Operand A Hex	Test Operand B Decimal	Test Operand B Hex
5.67	40B570A4	0.44	40C3851F
347.978	43ADFD2F	0.00078	43ADFD49
12	41400000	0.0487	4140C77A

*Zoom in to the values more clearly, nothing else i can do

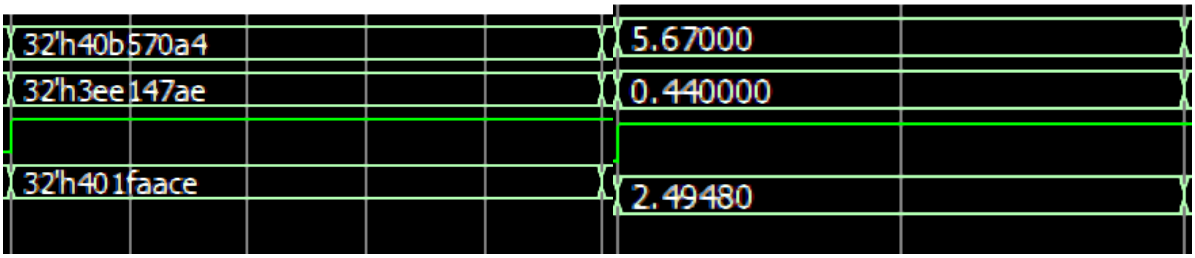
Test Case 1 (Addition):



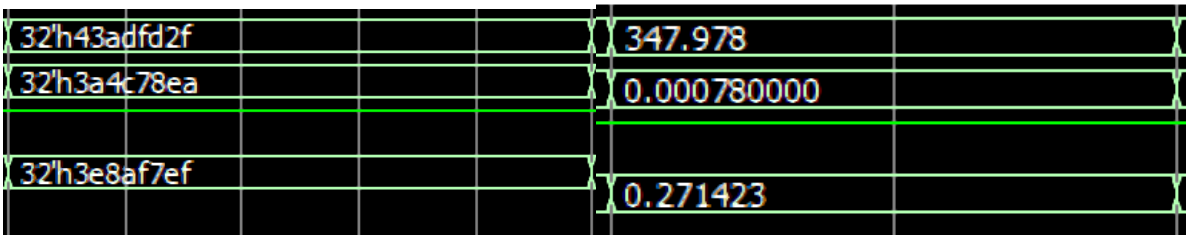
Test Case 2 (Addition)



Test Case 4 (Multiplication)



Test Case 5 (Multiplication)



6. Future Work

There are specific improvements that could be made as well as additional functionality.

The FPU was limited to positive values as the Floating Point Addition Module was not designed or tested with negative values. This can improve the robustness of the FPU by ensuring both negative and positive can be added.

With the Floating Point Multiplication Module, there are no guards to check against overflow. When multiplying two n-bit numbers, the resulting product will be 2n-bits wide. This significantly increases the probability of overflowing. More development time could yield guards against overflow, as well as flag wires to indicate when they have happened. In addition, the division and subtraction operations are what we would implement in the future. These operations would be added to our current program written in Verilog as well as creating a testbench to show correct functionality from each new operation. With these new added operations, our FPU program would be an overall rounded arithmetic calculator capable of doing the most essential arithmetic operations.

7.Conclusion

The designed Floating Point Unit performed as was expected. The FPU was able to add and multiply three IEEE 754 32-bit float values. As with all floating operations, some precision will be lost when

adding or multiplying. This is a limit of the 32-bit single precision value. The computer has a finite amount of bits that can be used to encode both the value and its exponent.

Floating point operations are typically the most costly operation a processor can perform. Many high performance processors will typically be measured on the speed at which they perform floating operations, known as Floating Point Operations per Second or FLOPS.

8. Appendix

Source Code

Addition Unit

```
module FP_Addition_Unit(
        input [31:0] A, B,
        output [31:0] sum
);
//exponenent part of the IEEE 754 is reperesented as a unsigned 8-bit val
wire [31:0] wr_A_operand, wr_B_operand;
wire [23:0] wr_A_significant, wr_B_significant; //24 bits account for leading 1
wire [7:0] wr_A_exp, wr_B_exp, wr_exp_diff;
wire wr_A_sign, wr_B_sign, wr_sum_sign;

wire [23:0] wr_shifted_B_significant; //used for nomalization
wire [7:0] wr_normalized_B_exp;         //normalized exponent of B operand

//sum of significants of the two operators, includes the hidden bit.
wire [23:0] wr_significand_sum;
//A wire is always the largest number, this will assign A and B inputs to their respective wires based on the exponent comparison
assign wr_A_operand = (A[30:23] < B[30:23]) ? B : A;
assign wr_B_operand = (A[30:23] < B[30:23]) ? A : B;

assign wr_A_significant = {!(wr_A_operand[31]), wr_A_operand[22:0]}; //significants are appended with a leading one if the value is positive
assign wr_B_significant = {!(wr_B_operand[31]), wr_B_operand[22:0]};
assign wr_A_exp = wr_A_operand[30:23];
assign wr_B_exp = wr_B_operand[30:23];
```

```
assign wr_sum_sign = wr_A_operand[31] || wr_B_operand[31];

//difference between exponents, used to shift

//should always yield a positive value as from this point on A operan > B operand

assign wr_exp_diff = wr_A_operand[30:23] - wr_B_operand[30:23];

//normalization

assign wr_shifted_B_significant = wr_B_significant >> wr_exp_diff;

assign wr_normalized_B_exp = wr_B_operand[30:23] + wr_exp_diff;

//addition block--------------------------------------------------

//adds the two significants together

assign wr_significand_sum = wr_A_significant + wr_shifted_B_significant;

//output

assign sum = {wr_sum_sign, wr_A_exp, wr_significand_sum[22:0]}; //truncates the leading 1 bit


endmodule
```

Multiplication Unit

```
module FP_Multiplication_Unit(
        input [31:0] A, B,
        output [31:0] product
        //, output overflow
);
wire [23:0] wr_A_significant, wr_B_significant; //24 bits account for leading 1
wire [7:0] wr_A_exp, wr_B_exp, wr_product_exp, wr_exp_sum;
wire [47:0] wr_product, wr_product_normalized; //48 Bits
wire wr_product_sign, wr_product_round, wr_normalized ;
wire [22:0] wr_product_significant;
assign wr_product_sign = A[31] ^ B[31]; //determines the sign of the products
assign wr_A_significant = {!(A[31]), A[22:0]}; //significants are appended with a leading one if the value is positive
assign wr_B_significant = {!(B[31]), B[22:0]};
assign wr_A_exp = A[30:23];
assign wr_B_exp = B[30:23];
//multiplication block
assign wr_exp_sum = A[30:23] + B[30:23]; //sum of the exponents
assign wr_product = wr_A_significant * wr_B_significant; //product of significants
assign wr_product_round = |wr_product_normalized[22:0];//rounding operation
assign wr_normalized = wr_product[47]; //extract first bit of product for normalization
assign wr_product_normalized = wr_normalized ? wr_product : wr_product <<1; //shifts for normilization based on product's MSB
assign wr_product_exp = wr_exp_sum - 8'd127 + wr_normalized;
assign wr_product_significant = wr_product_normalized[46:24] + {21'b0,(wr_product_normalized[23] & wr_product_round)};
assign product = {wr_product_sign, wr_product_exp, wr_product_significant};
endmodule
```

FPU

```
module FPU(
        input [31:0] A, B,
        input control,
        output [31:0] result
);

wire [31:0] product, sum;

FP_Multiplication_Unit FPMU(
        .A(A),
        .B(B),
        .product(product)
);

FP_Addition_Unit FPAU(
        .A(A),
        .B(B),
        .sum(sum)
);

assign result = control ? product : sum;

endmodule
```

FPU TestBench Code

```
module FPU_tb_add;

reg [31:0] A, B;

reg ctrl;

wire [31:0] out;

parameter hold_interval = 10;

FPU DUT(

        .A(A),

        .B(B),

        .control(ctrl),

        .result(out)

);

initial

        begin

                //test adding

                //5.67 + 0.44

                ctrl = 0; A = 32'h40B570A4; B = 32'h3EE147AE; #hold_interval;

                //347.978 + 0.00078

                ctrl = 0;A = 32'h43ADFD2F; B = 32'h3A4C78EA; #hold_interval;

                //12 + 0.0487

                ctrl = 0; A = 32'h41400000; B = 32'h3D4779A7; #hold_interval;

                //test multiplication

                //5.67 * 0.44

                ctrl = 1; A = 32'h40B570A4; B = 32'h3EE147AE; #hold_interval;

                //347.978 * 0.00078

                ctrl = 1;A = 32'h43ADFD2F; B = 32'h3A4C78EA; #hold_interval;

                //12 * 0.0487

                ctrl = 1; A = 32'h41400000; B = 32'h3D4779A7; #hold_interval;

        end

endmodule
```

References

K. Daly, "Addition of single precision floating point numbers," *YouTube*, 07-Mar-2013. [Online]. Available: https://www.youtube.com/watch?v=IiHK18n0pm4. [Accessed: 16-Dec-2020].

J. Schrum, "Floating Point Arithmetic 2: Multiplication," *YouTube*, 03-Jul-2015. [Online]. Available: https://www.youtube.com/watch?v=27JjUa-eu_E. [Accessed: 18-Dec-2020].