IEEE Standard Floating Point Processor on Spartan 3E FPGA

Erich Meissner[1] and Josiah Boyle[2]

Abstract— **Throughout this project, we recognized that floating point operations are difficult to implement on Field Programmable Gate Arrays (FPGA) because of the complexity of algorithms. In our research, we learned that many scientific applications require floating point arithmetic because of the high accuracy required in their calculations. In this project, we explored and FPGA implementation in the Institute of Electrical and Electronics Engineers (IEEE) -754 floating-point numbers standard. Again, floating point arithmetic is significant because many algorithms require them because they support a comprehensive range. In our project and this final report, we describe our efficient implementation of an IEEE 754 single precision floating point processor. We also tested our design in a Xilinx Spartan3E Nexys2 FPGA. We were tasked to implement pipelining into the IDE environment we wrote our floating point arithmetic in. The pipelining provides high performance and is used to execute multiple instructions simultaneously. Our team used top-down design approach for the three arithmetic modules, which are addition, subtraction, and multiplication. Synthesis and simulation results are obtained by using the Xilinx 14.7 ISE platform.**

I. INTRODUCTION

We offer a background on the importance of FPGAs, and we also touch on the qubiquity of the IEEE-754 format. In both industy and research, Field Programmable Gate Arrays (FPGAs) allow the developers to build any logic device both quickly and easily. The programmability and flexibility of FPGAs make them ideal for prototyping, quick time-to-market applications, one-off implementations, and customized hardware. They are especially valuable in applications when a custom circuit is required. An example of this application is in the NYSE trading floor where FPGAs are used to input orders seamlessly. However, we learned that the production volume does not always justify the costs and time of fabricating them. Recent advances in process technology have led to a dramatic increase in FPGA densities and speeds, meaning the ENEE359F course should look into more modern FPGAs. We suggest the UMD ECE department purchase new FPGAs beause they are now becoming more suitable for supporting designs with dense computations and high operating frequencies. Furthermore, FPGAs are becoming more suitable for supporting high speed floating point processors, meaning future ENEE359F courses could implement more advanced arithmetic.

We find it important to also note that floating point units are widely used in digital applications such as digital signal processing, digital image processing, and multimedia. We discovered that in conventional floating point units, the most frequently used floating point operations are multiplication and addition/subtraction counting for more than 94 percent of all floating point instructions. Hence, our ENEE359F final project was especially applicative to real-world use. We noticed in our project that floating-point addition is aroguably the most complex operation in a floating-point arithmetic. The addition consists of many variable latency and area dependent sub-operations. Our team learned that in floating-point addition implementations, latency is the primary performance bottleneck. In our struggles building a floating-point adder, we researched several papers that are dedicated to improving the overall latency of floating-point adders. We discovered that various algorithms and design approaches have been developed. Overall, the floating-point unit is one of the most important custom applications needed in most hardware designs, as it adds accuracy, robustness to quantization errors, and ease of use. There are many commercial products for floating-point addition that can be used in custom designs in FPGAs but cannot be modified for specific design qualities like throughput, latency, and area. We also learned of research that has also been done to design custom floating-point adders in FPGAs. Hence, we were tasked in the project to attempt to increase the throughput by pipelining.

II. FLOATING POINT ADDER AND SUBTRACTOR DESIGN

Our algorithms for addition and subtraction required more complex operations due to the need for operator alignment. We studied three different floating point add and subtract algorithms. The three include: standard, leading-one predictor (LOP), and 2-path. It is important to note that the implementation of these steps defines floating point arithmetic unit latency and area. Our team stuck with the standard method of floating point addition and subtraction. We defined standard floating point addition to require these five steps:

1. Exponent difference
2. Pre-shift for mantissa alignment
3. Mantissa addition/subtraction
4. Post-shift for result normalization
5. Rounding (our team was tasked to round to infinity)

The standard floating point adder we implemented is first figure in the appendix below. The exponents of the two input operands, ExponentA and ExponentB are fed to an exponent comparison program. Then, in our pre-shifter, a new mantissa is created by right shifting the mantissa corresponding to the smaller exponent by the difference of the exponents so that

*This project is part of the University of Maryland ECE curriculum in the ENEE359F course.

[1] me@erichmeissner.com
[2] josiahboy96@gmail.com

the resulting two mantissas are aligned and can be added correctly. Our right shifting is simply dividing by a power of 2. Then, if the mantissa adder generates a carry output, then the resulting mantissa is shifted one bit to the right and the exponent is increased by one. We should not that much of our project was implemented using "if-thrn" logic. Next, the normalizer transforms the mantissa and exponent into a normalized format, which our team defined for ourselves as it was not explicitly stated what "normalizing" required.

Our program made decisions based on the position of the most significant one in the mantissa; therefore, the resulting mantissa is left-shifted by an amount subsequently deducted from the exponent. In our normalization process, if the adder result is too large, then it shifts to right (which is simply dividing by 2). Furthermore, if the adder result is too small, then it shifts to right (which is simply multiplying by 2). We noticed in our processor that precision is lost. This loss is due to the event when some bits are shifted to right of the right most bit or are thrown out completely. We attempted to obtain better accuracy by shifting out bits.

A. Methodology of Standard Floating Point Add/Subtract Algorithm

For the our adder and subtractor, we approched it with a standard algorithm as opposed to the other methods we researched and mentioned above. We have our exponent comparison program that is implemented with a subtractor and a multiplexer. The comparison program requirements are based on the exponent bit-width. Therefore, the size of the pre-shifter is also based on the bit-width of the matissa. The size of our mantissa adder depends on the adder architecture and sign mode. It is also important for the requirements of the ripple-carry adder. The normalizer is about the same size as the mantissa adder, and the shifter is equal in size to the pre-shifter. The subtractor is about the same size as the exponent comparison program. Finally, the size of the normalizer is about the sum of the sizes of the other three components. This section is illustrated in the first figure of the appendix.

III. ALGORITHM FOR FLOATING POINT MULTIPLICATION

We defined floating point numbers to have the following form:

Z= (-1S) * 2 (E - *Bias*) * (1.M)

As we did in our finaly presentation, we defined the following steps to multiply two floating point numbers:

1. Multiplying the significand; for example (1.M1*1.M2)
2. Placing the decimal point in the result
3. Adding the exponents; for example (E1 + E2 *Bias*)
4. Obtaining the sign; for example (s1 xor s2)
5. Normalizing the result; for example (obtaining 1 at the most significant bit of the results significand)
6. Rounding the result to fit in the available bits; our assigned rounding scheme was to round to infinity
7. Checking for underflow or overflow occurrence

The second figure shows our data path for floating-point multiplication, and we simplified our illustration to show only the main parts of the data path. Our prealignment and normalization stages require large shifters. Our prealignment step requires a right shifter that is twice the number of mantissa bits, for example 48 bits is required for this single-precision floating point processor. This requirement is because the bits shifted out have to be maintained to generate the needed bits for rounding. Furthermore, our shifter only needs to shift right by up to 24 places for this single-precision processor.

Next, the normalization stage requires a left shifter equal to the number of mantissa bits plus 1, for example there will be 25 bits for our single-precision implementation. Also, if the rounding of the mantissa results in an overflow, then our mantissa will shift right by one and the exponent is then incremented. This increment will be 24 by 24 bit for this single-precision processor.

In our 359F project, we were taksed to employ a Radix-4 modified booth encoded Wallace multiplier. We illustrated our multiplier in the third figure in the appendix. We based our illustration on the designs we looked over in class and lab. Our radix-4 recoding halves the number of partial products. This halving reduces the number of levels required in the Wallace tree and slightly improved our performance by reducing area requirements.

IV. SIMULATION RESULTS

A. Adder and Subtractor

The simulations results for our adder and subtractor can be viewed in the fourth figure of the appendix. The op-a and op-b are the two inputs, which are both 32 bits as required in our project, of our floating point adder and subtractor. The "add" variable, which is 32 bits as well, is the output of floating point adder. In our program, overflow bit will be high if the range exceeds the maximum value. Our program handles the case when the range is smaller than the minimum value by making the underflow high.

B. Multiplier

We defined fp-a and fp-b to be the the two inputs, which are both 32 bits, of floating point multiplier. Our fp-z, which is also 32 bits, is the output of floating point multiplier. The results can be seen in the fifth figure of the appendix.

V. CONCLUSIONS

Our ENEE359F project is an investigation into the IEEE Standard Floating Point Processor. We learned of the importance of this format and the intricacies of implementing the arithmetic. We have presented a single-precision floating point processor with three operations implemented and simulated. These three operations are addition, subtraction, and addition. The floating point addition was the most interesting to build as there were several possible methods of implementation. We believe The future scopes of this project are to build the ENEE359F IEEE Standard Floating Point Processor using more advanced Field-Programmable Gate Arrays (FPGAs).

ADDENDA

The multiplier and FPGA communications were developed by Erich Meissner. The adder/subtractor and optimization was performed by Josiah Boyle. Again, we believe The future scopes of this project are to build the ENEE359F IEEE Standard Floating Point Processor using more advanced Field-Programmable Gate Arrays (FPGAs).

The report was written and edited entirely by Erich Meissner

APPENDIX