High Speed Computer Arithmetic ECEN 4233 Final Project (v1.0.3) Spring 2024

James E. Stine
Electrical and Computer Engineering Department
Oklahoma State University
Stillwater, OK 74078, USA

The project [1] in this course is designed to test your knowledge of the topics discussed in lecture and apply them to a real-world problem. The project will consist of a task and you should work individually and submit one set of files. Under no circumstances will copying be allowed on this project and any piece of software, hardware, or any other item that is copied may result in a failing grade for this class. However, I do encourage that discussion should be exhibited between all individuals to understand the problem. If you need any clarification on what constitute an illegal action, please feel free to contact me.

In this project, you will be implementing an efficient method for approximating division using Goldschmidt's algorithm or sometimes called division by convergence. For graduate students, this project will involve modifying your implementation, so that it can also produce the square root of your input operand. The Goldschmidt iteration is an attractive method because it converges quadratically. This means that one iteration will double the number of accurate bits in the approximation. For example, if I have an approximation that is accurate up to 2 bits, after the next iteration, it will be accurate up to 4 bits. The Goldschmidt algorithm can be applied to a variety of different systems and is also attractive for square-root thus making it popular for modern high-performance processors.

You will also implement this structure for IEEE 754 floating-point and support both round-to-nearest even and round-to-zero rounding modes [1]. This will involve computing the remainder as well as ancillary logic to handle the exponent.

I. Basic Baseline Project

This project contains a significant amount of work which will require substantial design effort. Therefore, it is probably a good idea to start early. For this project, you will be following the details listed below:

- A 32-bit IEEE 754 Divide (and possibly Square-Root) Unit that uses Goldschmidt's iteration to compute its operation. The unit is designed for use in a single-precision floating-point unit, so it must be accurate to 23 fractional bits. That is, one (1) integer and twenty-three (23) fractional-bits.
- You will need to add the remainder to properly handle the rounding mode.
- This is an individualized project but you are welcome to work together but you should each submit your final HDL.
- In order to verify your results, you must use Java or C (or some other high-level language) to verify your design constraints. Sample Java and C code is provided to help you start this process, as explained in class.
- The HDL is due **April 26, 2024** at midnight with no exceptions except approved medical and legitimate excuses.
- The design should be implemented and tested with SystemVerilog. Several design vectors should be tested to make sure it works appropriately.
- Table I lists some *suggested* input/output signals. Please feel free to add more or delete some, if you wish. Sizes and names are only suggested and may be modified accordingly.
- You should only employ one (1) multiplier in your design, therefore, your final result may take several cycles. Your design can have any number of adders, counters, or compressors, but you may only use **one** multiplier in this project.

II. Overall Goal

This project does require a logic simulation using ModelSim/QuestaSim, as described in our examples with class and our repository. For this project, you are expected to demonstrate your understanding of the procedures needed to compute the final result, an adequate error analysis, and some simulation to show that your procedure would work. One method to complete an acceptable simulation is to write a Java or C program to determine the necessary bit sizes to meet a given accuracy (in our case 23 bits). Another method might use a programming environments like MATLAB or Maple. The best one is to use one that you feel comfortable with. However, based on previous semesters where students did not

[1] Note that this document is updated periodically to provide you with the most reliable information to complete the project. Although previous versions of this document are basically correct, frequent revisions are made in order to help you save time and **not** to hinder your performance. Therefore, make sure you check back periodically for updates. I will add revision numbers to help you identify which version of this document you have.

Signal	Input/Output	Size (bits)	Description
N	Input	32	Input operand (dividend)
D	Input	32	Input operand (divisor)
Operation	Input	variable	Operation selector
rm	Input	2	Rounding mode
clk	Input	1	Clock
Load	Input	1	Load registers
Q	Output	32	Output operand

TABLE I. SUGGESTED PINS AND OPERAND SIZES

understand the importance of this process and fudged Excel, you must use a program to verify the accuracy of your final computation. Therefore, you can use the provided Java or C programs to verify the sizes from each unit. Although you are welcome to use another method, such as from MATLAB, it must be something equivalent to the provided C or Java or code in its operation.

For any algorithm and especially for algorithms that may be employed to save or protect people's lives, it is imperative that an analysis of the error involved in the computation be completed. That is, a worse-case scenario should be examined so that the precision required is satisfactory met. In other words, that we can compute an operation between two 24-bit operands and the answer should be accurate to less than a unit in the last position (ulp) of the precision required (i.e. 23 fractional bits).

A. Required Elements

The following are required elements to the project:

- Name your top-level design `fpdiv` or `fpdivsqrt`.
- A program or mathematical analysis showing your algorithm will work. Usually, designers utilize a program to verify your operation, as well as your testbench, that you feel comfortable with. You **must** use the C or Java program (or equivalent) to test your design. It is also advisable to test several hundred vectors to see if it meets your requirements. This should be relatively easy within the program. Ask me, if you need help with this.
- A block diagram of the hardware necessary to implement your algorithm. It is easiest to take a hierarchical approach to documenting your block diagram. That is, display the top-level block diagram on one page, and then on each subsequent page show what each lower-level block diagram is composed of. Diagrams usually take a ridiculous amount of time to create and an early implementation will help you complete this project effectively. Although I have done many figures over my lifetime, I am still looking for the best program to create figures. I would highly recommend a program like Microsoft Visio, draw.io, or xfig. Please try to simplify the diagrams as they can be ever consuming and completing them can really be a time sink.
- You should use HLS (i.e., `assign Z=A*B` to generate the multiplier to get thigns started, however, you are welcome to use the Wallace multiplier for efficiency found within the repository.
- ECEN 5080 students are also required to implement floating-point square root in addition to floating-point division. ECEN 4233 students just have to implement floating-point division.
- I do expect you to document all combinational and sequential logic you have in your design, but try to summarize the detail of your design.
- Your final design must pass all the TestFloat [2] validation values (see the HSCA repository).
- You may use any hardware element discussed in class provided you discuss it completely in your project. For this year, let's try to summarize as much as you can to make it succinct and easy to produce. I am putting more emphasis this year on the implementation.
- You do not have to worry about IEEE 754 exceptions and can be added on a case-by-case basis for extra credit (e.g., $\pm\infty$).

B. Steps

Sometimes, I get asked what steps are necessary in this project to meet the given constraints and although I encourage adaptability and individuality, the question is valid and something I should address. These are the suggested steps - feel free to improvise or skip steps, if necessary. There are no right or wrong ways to complete the project. It is important to emphasize that this process is what digital designers typically encounter when implementing a unit at companies like Intel, AMD, ARM, NVIDIA, and others. Users demand accountability that any unit you utilize will function, as expected. Therefore, it is a good thing to learn the process you utilize in this project.

1) The main repository for all the files is available here: https://github.com/stineje/HSCAProjectS24/.
2) Inspect the C or Java code and learn how to compile and run it repeatedly.
3) Modify the C or Java code to determine the sizes of your units inside your main functional unit. Make sure you choose a data type for your results that satisfy the given input and output precision.
4) Run an adequate number of vectors to make sure your design meets the accuracy for the input and output operands. Save these vectors, so you can utilize them to test your SystemVerilog test benches.
5) Put together a block diagram of your design with the sizes you determined in previous steps.
6) Stub out your SystemVerilog along with a testbench for your design. Make sure your testbench works!
7) Once you have a reasonable block diagram, start implementing lower-level blocks in SystemVerilog where you try to use these lower-level blocks to validate their

functionality. It helps to fully test lower-level blocks as these blocks are sometimes the most problematic within a design. Also, its easier to exhaustively test a lower-level design, as it usually has a low number of input and output ports.

8) Start assembling the upper-level SystemVerilog blocks and testing whether they match the results from your C or Java code line by line. It is advisable to start out with a simple design (e.g., only a multiplier) and work upwards. Always verify the operation of your design before starting on working a new block.
9) Finalize your final SystemVerilog block and testbench. You may have several small iterations here to figure out the input and output signaling as well as handling the control structure.
10) For the two's complementator part of Goldschmidt operation, I would recommend using the one's complement for efficiency and optimization as mentioned in class.

C. Rounding

Rounding is not easy for multiplicative divide. It is even more difficult to describe, but the main idea is that we add an extra constant to the quotient and use the remainder to see how far we are from the rounding. Fortunately, I am only asking you to perform two rounding functions: round-to-nearest-even (RNE) and round-to-zero (RZ). We will use the techniques published here [3], [4]. This process amounts to rounding to one additional bit or $pc{+}1$ bits. This will provide the necessary mechanism to produce a result that has an error no worse than ± 0.5 ulp, or 2^{-pc}.

The rounding function assumes that a biased trial result q_i has been computed with $pc+1$ bits, which is known to have an error of $(-0.5, +0.5)$ ulp with respect to pc bits. The extra bit, or guard bit, is used along with the sign of the remainder, a bit stating whether the remainder is exactly zero, and the final result is chosen such that it has three possible results, either q, $q-1$, or $q+1$. Again, for RNE and RZ this will be trivial. The rounding details are shown in Table II

This rounding is complicated and you just have to make sure you add the right constant to make sure the rounding mode is accurate. Technically, this forces a 1 into the Least Significant Bit (LSB) to guarantee the precision needed [4]. This amounts to the following additions for q, $q-1$, and $q+1$ for q, qp, and qm, respectively (assuming a 32-bit multiply – you can change to reduce area of your multiplier).

```
assign q_const = {32'h0000_0040};
assign qp_const = {32'h0000_0140};
```

Guard Bit	Remainder	RN	RZ
0	$= 0$	trunc	trunc
0	$-$	trunc	dec
0	$+$	trunc	trunc
1	$= 0$	RNE	trunc
1	$-$	trunc	trunc
1	$+$	inc	trunc

TABLE II. ACTION FOR ROUND FUNCTION

```
assign qm_const = {32'hFFFF_FF3F};
```

You need to store these three values during the final iteration and choose the correct value based on Table II. These outputs are not necessary during the normal Newton-Raphson iteration.

What is interesting about Table II, is the RNE case in the middle of the table. It should be noted that for division where the precision of the input operands is the same, which is true in our case and for most IEEE 754 units, the exact halfway case cannot occur. Therefore, you do not have to check this option which makes things abundantly easier.

To compute the remainder during the rounding phase, you can reverse the table to make things easier. Some researchers call this process back multiplication [4]. That is, instead of performing the operation

$$b \times q - a \ ,$$

is implemented instead as:

$$a - b \times q \ .$$

That is, the sign of this remainder is thus the negative of the true remainder sign. The remainder can get tricky because of the range of the quotient. For example, the input operands should be $[1, 2)$, however, the output can be in the following range $[0.5, 2)$. This range will affect your remainder computation and should be accounted for.

III. GRADING

Every year I get asked on the grading that I utilize to ascertain what grade you get for this project. My goal is to also teach you, besides the theory, that verbal communication is vital to your success. I suggest that you spend at least one week for the write-up with this project. I am a fair grader, but I have certain biases, so please make note of the following items:

- Please note that there is no correlation between how much work you complete and the grade of your project. I have had students in the past who have worked tirelessly on the project, but ended up with a C, because they failed to document anything. That is, they had a wonderful design, but gave up with the design and handed in an inaccurate picture of their work. Therefore, I encourage you to start early and commit regular updates to this project.
- I am here to help you, but I cannot spend time debugging your project. I would highly advise working from the ground up and assembling parts after you test them. Always test your output against a known value, preferrably from the C code.
- Your project will be based on the following break down:
  - 70% Bi-Weekly status reports (see below in Section IV).
  - 20% Final functional block diagram and SystemVerilog code.
  - 10% Testing : final testing against TestFloat (see below).

- A working SystemVerilog model of your design showing several vectors accurately computing your answer (including a final test bench and DO files). There is no need to show me all the lower level test benches - a final test bench is sufficient. All files **must** be handed in with your project as one compressed file (e.g., zip), so it may be verified for operation.
- Please hand in all HDL to Canvas by the deadline. There are no exceptions to this other than medical and legitimate excuses.

IV. Meetings/Bi-Weekly Status Report

I have had excessive problems with a small number of students in this class in that they wait until the last week to implement anything (Note: not all students do this but I would say about half abuse the deadline). I have tried multiple ideas including suggested milestones, however, nothing has worked. Then, I get a barrage of Emails similar to "things are not working and I am not sure why?". I also get a flurry of "I wish you did something to keep us on track!". My thought process here is to help you with this option and this is the third time I am trying this (the first/second time was not as good as I had hoped, but I am hoping that the third time is the golden ticket). For this reason, I am forgoing the report to have bi-weekly status reports. In summary, I am using the suggestion from a colleague/friend to have weekly status reports instead which has worked for him.

You will be assessed on this bi-weekly status report. I will require that you have a one page summary on what you have worked on, what works, what you have done. You can bring diagrams or show me anything you have done and I can assess your grade weekly. If you skip a week, I will give you 0 points for your grade that week. I would suggest visiting with me during office hours, but you can visit with me anytime that we are both free. However, you must visit with me each week to get a grade; otherwise you will get a 0 for that week. I will not conduct the meeting over Zoom unless you are a remote student or have legitimate problems attending in person. You **must** meet with me weekly to get full credit. Remote students should contact me to arrange a weekly Zoom or Teams discussion. These meetings must be completed by Friday at 5 PM each week starting the week of Monday, April 1, 2024. This amounts to about 2 mandatory meetings for the semester, however, you are welcome to visit with me more than 2 times during the semester. I also suggest that you do not try to visit with me last minute (e.g., 4:45 PM on a Friday).

Please note that a significant amount of your grade is based on this meeting. I would suggest that you be be prepared for each meeting with the 1 page status report you hand into me. I will give you a grade at the end of our meeting so you have some feedback if you are underperforming or are on track for completing the design. Again, I would highly suggest you prepare for our meeting and it should last about 5-15 minutes at maximum. If you are not prepared, your grade will represent this.

V. Hints

The best way to complete this project is to work from the ground up. That is, build the version first for a small size (e.g. 16 bits) or a simple design by adding more functionality. Hardware designers sometimes make mistakes by initially trying to put too much into a hardware design. It is advisable to make the logic simple and expand the logic incrementally.

I would highly advise creating all the combinational logic first before you go on to the sequential logic, if necessary. Sequential logic can change due to the large number of signals that may need to be controlled. Therefore, a three step process is recommended:

- Draw the complete block diagram of all combinational elements on a nice large piece of paper. Some students have commented that legal paper makes a good way of designing logic blocks.
- The baseline project will be discussed rather regularly in class, therefore, paying attending to the discussion on different days may clear up any confusion on its implementation.
- You are welcome to use structural, RTL or HLS-level coding for your SystemVerilog.
- Make a block diagram of your control unit. Your control logic will probably be synchronous-based, therefore, it is probably a good idea to have the complete control tabulated for each cycle. You do **not** have the control implemented, but it should be integrated within your testbench. That is, you can hard code the control logic into your testbench and test whether your datapath will work. Some students have implented the control unit to make things easier, but the choice is yours.
- I will provide a set of tests that you should use using the University of California-Berkeley (UCB) TestFloat test suite [2]. You will need to have your testbench read these values in and test against your unit (a sample testbench will be provided).
- There are many ways to get extra credit in this project by implementing extra features. However, every year I have students spend way too much time trying to get extra credit working and not enough time on the documentation. I would highly suggest starting the baseline project and co-implementing the extra credit to allow you time to complete everything in an orderly fashion.

References

[1] "IEEE standard for binary floating-point arithmetic," *ANSI/IEEE Std 754-1985*, pp. 1–13, 1985.

[2] J. Hauser, "The SoftFloat and TestFloat Validation Suite for Binary Floating-Point Arithmetic," Tech. Rep., University of California, Berkeley, 1999, Available at http://www.jhauser.us/arithmetic/TestFloat.html.

[3] E.M. Schwarz, "Rounding for quadratically converging algorithms for division and square root," in *Conference Record of The Twenty-Ninth Asilomar Conference on Signals, Systems and Computers*, 1995, vol. 1, pp. 600–603 vol.1.

[4] S.F. Oberman, "Floating point division and square root algorithms and implementation in the amd-k7/sup tm/ microprocessor," in *Proceedings 14th IEEE Symposium on Computer Arithmetic (Cat. No.99CB36336)*, 1999, pp. 106–115.