Summer 2019

DEPARTMENT OF ELECTRICAL AND COMPUTER ENGINEERING

Mentor, A Siemens Business

ECE 595 Final Project

IEEE-754 32-BIT FLOATING-POINT FMA HYPERCUBE FOR 6X6 MATRICES

ORIGINAL DESIGN AND VERIFICATION BY ALEX OLSON

Introduction

Matrix multiplication is a highly parallel problem in computing that is seeing recent increase in demands due to the recent explosion of machine learning. However, training models with large data sets can be a painfully slow process which results in wasting of valuable time, lots of time, when models are wrong. In order to alleviate this pain, there has been a lot of interest in developing specialized hardware that can speed up these processes. In order to help myself gain experience in this field, I decided to create my own design which I envisioned as a matrix processor where a register of the processor could hold an entire matrix (all registers of a fixed size). Once the matrix registers are loaded, a control unit would then execute commands to add or multiply two matrix registers and put the resulting matrix into another register.

But why floating-point? Another challenge between machine learning and computation is that a set of data has a sampling range that could be distorted by something other than evenly distributed, discrete points such as integers. Floating-point provides an excellent range of small, large, positive, and negative numbers within a single set. I also wanted a reason to practice working with floating-point numbers a bit more to help make them less mysterious to me. The testbench standardizes on 32-bit floats because it is the lowest bandwidth that leverages the system tasks *$bitstoshortreal()* and *$shortrealtobits()* for verification simplicity. However, the design standardizes on a parameterized float size which may be adjusted to the desired level of precision.

Design

The design under verification is a 6x6 matrix multiplier that operates on 32-bit IEEE-754 floating-point values using a fused-multiply accumulate (FMA) that only performs a single rounding for increased performance and accuracy [1]. The FMAs are clustered in the shape of the solution matrix (also a 6x6) but are connected in a way that allows data to flow in from all side of the matrix (**Figure 1**). Each FMA node of the cluster is theoretically connected to four neighbors, forming a processing hypercube [2]. Odd and even matrices will have different partitioning where even matrices that are symmetrical will partition easily into smaller matrices, but an odd matrix will have an offset on its symmetry. **Figure 2** is an example of this partitioning on an odd 3x3 matrix, that also happens to be the size of the DUV for emulation. An example of the process on a 6x6 is shown in **Figure 3**, which also demonstrates the difference in partitioning of even matrices.

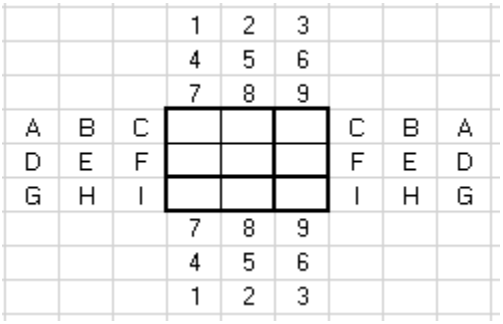

Figure 1 – 3x3 with Input Matrices Flowing in from All Sides

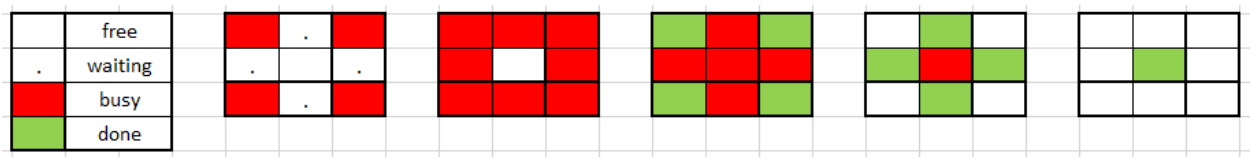

Figure 2 – 3x3 Cluster States

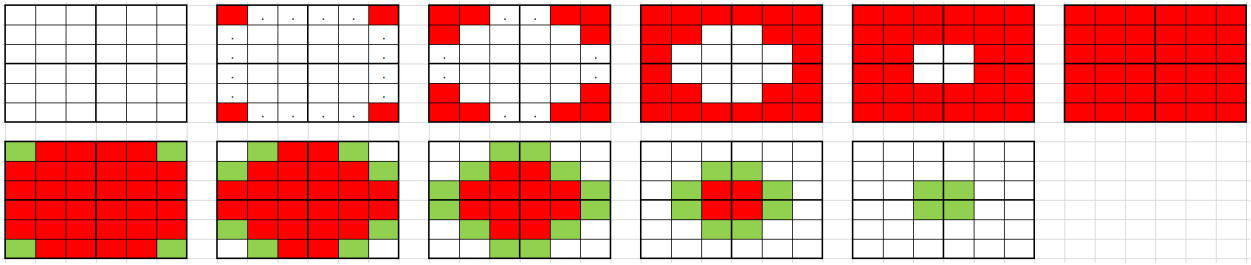

Figure 3 – 6x6 Cluster States

The design currently supports 4 different operations: NOP, LOAD, STORE, and MULTIPLY. The NOP operation is fairly mundane where a clock cycle gets burned, and hopefully nothing else happens. The LOAD and STORE operations are complementary to each other for getting data (matrices) into and out of the design. When a LOAD is initiated through an interface transaction, a packet of data is sent from the testbench that includes the operation, register address, and the literal matrix to be loaded into the register. The load hardware in the design extracts one element of the matrix per clock cycle and fills the internal register until full. A 6x6 matrix therefore takes 36 clock cycles to fill the internal register after the LOAD begins. The STORE operation is similar, only it transfers out the matrix, one element per clock cycle. Although this process is time consuming, the advantage is that a 32-bit bus interface in and out of the unit increases the portability of working with anything else that is capable of that bandwidth.

The LOAD and STORE operations interact with the registers by writing and reading to them respectively. However, the MULTIPLY operation also has a concurrent read and write relationship with the registers. A dispatcher module reads elements out of a specified matrix register and pushes the elements into the arithmetic unit of the design. A collector waits on the receiving end and *collects* the elements into a buffer as the ready signals go off as shown in the green patterns above (**Figure 2**). Once the center of the matrix is collected, the buffer is full and elements are written out to the registers, completing the matrix multiplication. The dispatcher and collector are under the control of a *controller* module, which only begins the multiplication process once it gets ownership of the requested addresses. The register file takes requested addresses and sends back a *ready* signal if the requested address is not being used by a different part of the design. The load/store modules and the dispatcher/collector modules both follow the signal handshake in laid out in **Figure 4** to get a mutex lock on the register. This way, you are able to do concurrent loads and stores while arithmetic processing is happening, as long as there are no address collisions. If a module cannot get exclusion of a resource, it stops for a few cycles before making another attempt.

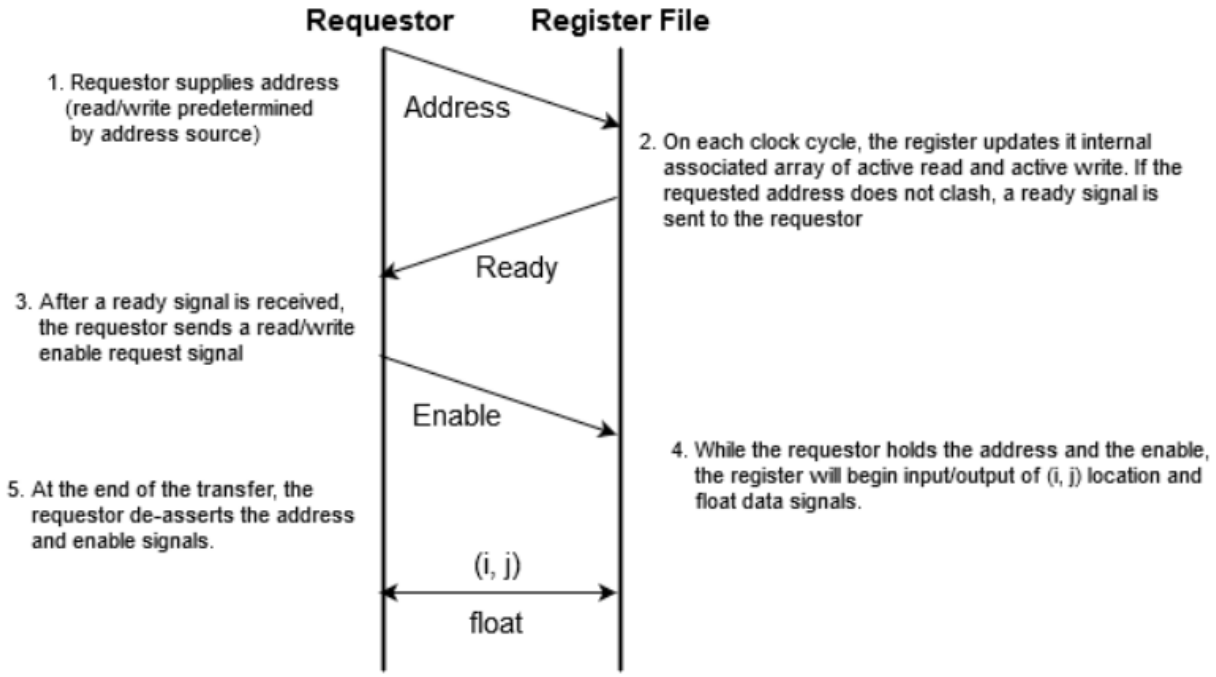

Figure 4 – Register File Handshaking

The execution cluster is made up of a network of interconnected FMA units. The FMA units were designed with this in mind. An FMA unit (**Figure 5**) can load either one or two floats at a time. In the case where two are loaded in, an extra state gets skipped and a cycle gets saved. Once the FMA has two numbers to multiply, it does so and goes through the normalize, accumulate, and rounding process before reaching the output state. The FMA counts the number of accumulations it performs, and when that size reaches N (N=6 for this version of the design), the FMA signals its output ready and sends out the float for the collector to receive. During the output state is also when the two input floats get passed to their neighbors. The FMA will hold these numbers until its neighbor sends a 'not busy' signal, which is how it knows that its neighbor finally got its result and it can return back to an idle state. Once back to the idle state, the FMA broadcasts 'not busy' to its neighbors and is able to start the process again. These transitions are shown it detail in **Figure 6**.

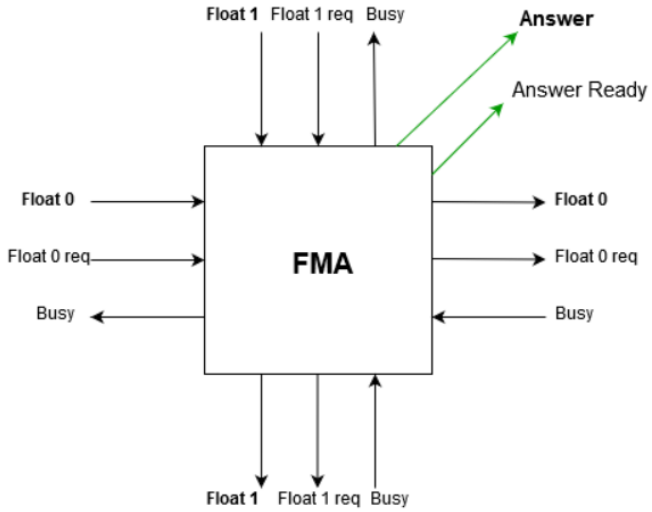

Figure 5 – Fused Multiply Accumulate Unit

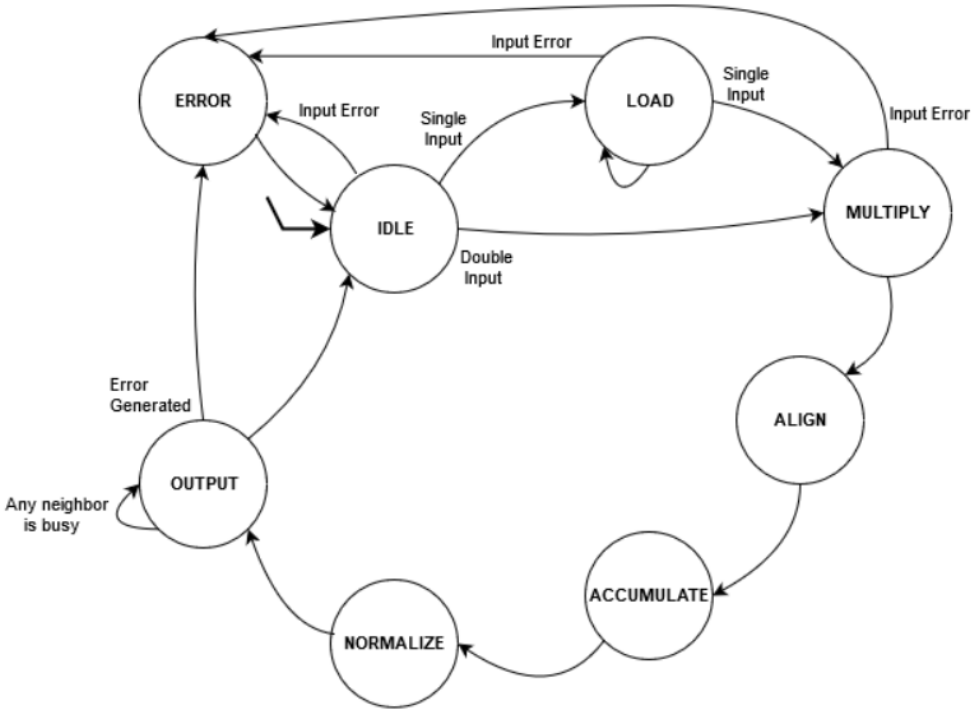

Figure 6 – FMA State Transition Diagram

The FMA units are arranged in a cluster as pictured in **Figure 7**. For odd numbered matrices such as the 3x3, the partitioning is offset because the FMA units that land on the line of symmetry theoretically have the option of getting the same number from either neighbor. In practice, only one side is given precedence and the matrix ends up having an offset partitioning. The advantage of using this method allows more units to be working at the same time, as well as only needing to distribute data to the perimeter of the network where the information gets passed around once it's inside. This saves unnecessary repetitive distribution of the same element to multiple FMA units.

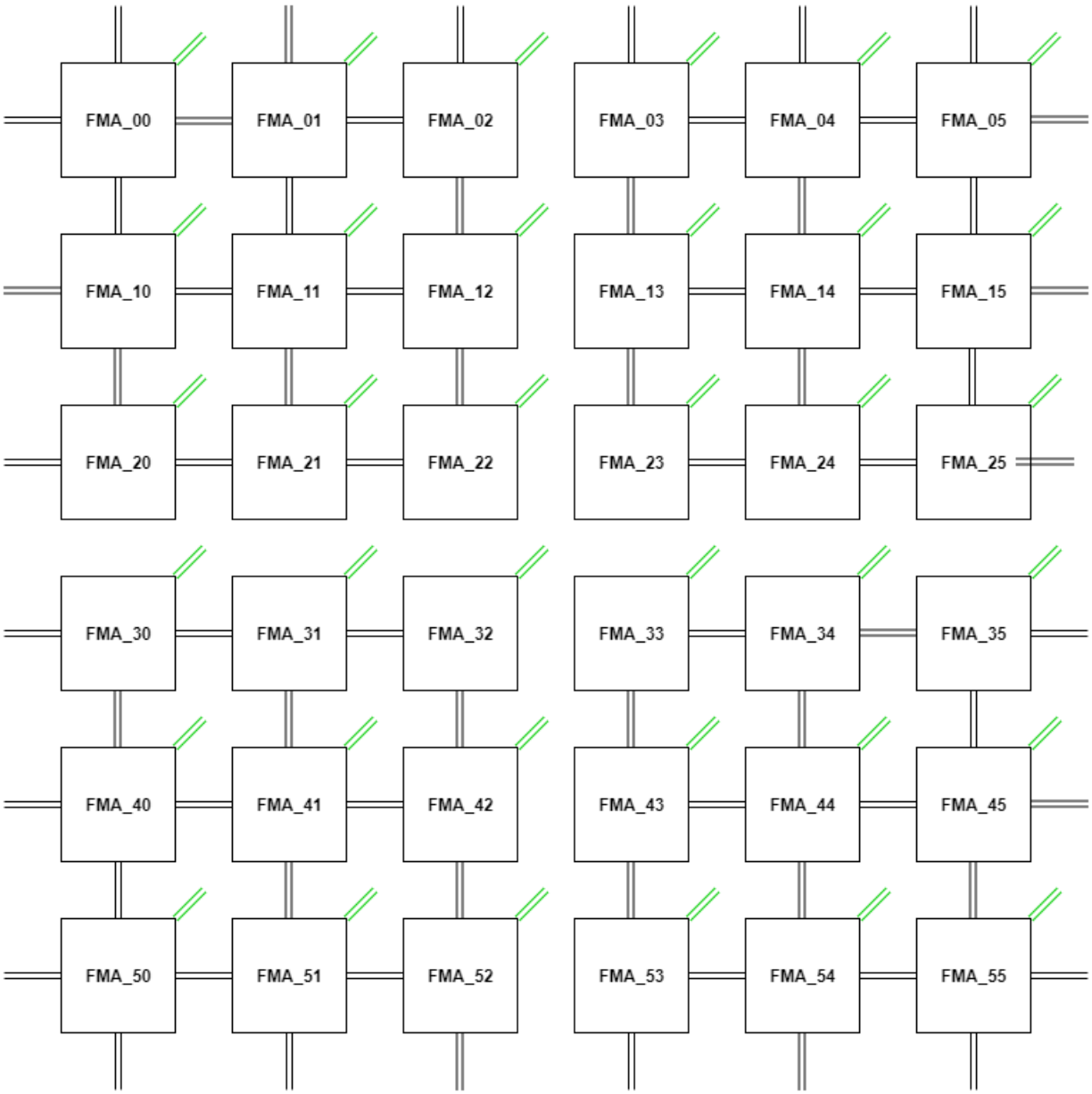

Figure 7 – FMA Units Arranged in a Cluster

Note about parameter choices:

When I first started creating this design, my goal was a parameterized NxN processor where N may be chosen at compile time. In order to finish in time for the competition, I scaled the matrix processing cluster down to 6x6, but there are still parts of the design (i.e. registers, load and store modules) that are parameterized as NxN and set to N=6 at compile time. My future plans for the design are to extend the other modules to handle parameterized matrices decided at compile time.

Verification

The verification of the design was performed on the Veloce emulator along with QuestaSim with velhvl set to *puresim* during simulation and set to *veloce* for emulation. A smaller number of tests were performed in simulation to show that the testbench was driving in cases and getting results. The number of tests was scaled up for the emulator.

The verification plan attempts to capture all of the standard running cases as well as checking for the execution of individual units. The first test sent into the design simply loads all of the internal matrices with an incremental string of numbers, and then subsequently store the numbers back out to the testbench where the sequence is checked. After this, a few preloaded matrices with values designed to check individual units. If a particular unit was not connected correctly, the resulting matrix should be able to pinpoint which unit out of the cluster is faulty.

Once the verification has reached this point, it is time to start checking special cases. First, the multiply by +1, -1, and 0 are all checked for appropriate results. Next, a few matrices and their inverses are multiplied to show that an identity matrix (1's on the diagonal, 0's everywhere else) is the resulting matrix. After that, very small matrices are multiplied together to produce underflow and very large matrices are multiplied together to produce overflow. The bulk of the tests come from scalable random test generation where various types and sizes of matrices are multiplied together at random. Matrix registers 0:2 are filled with generated matrices. They are multiplied randomly together (along with a chance that their results are used in subsequent multiplications) until matrix registers 3:7 are full. At this point, all registers are 'stored' out to memory and the reference model checks for errors. This process is scaled up to 500,000 iterations for emulation. Another add-on for emulation was another 500,000 back-to-back multiplications of the same two registers to try and see how quickly tests could be driven into the design.

The testbench used to test the design was an object-oriented testbench with objects for the checker, driver, testcase factory, scoreboard, and top-level connections shown in **Figure 8**. In this model, the driver calls some various matrix generation tasks derived from a parent stimulation class test factory and drives them to both the design and the checker (using a mailbox for the checker). The testcases are generated at runtime from command line arguments passed in through a Makefile. When a store is detected, the result from the DUT is differed with the reference model, and the total error is accumulated. If that error is above a defined threshold, the checker sends a fail score to the scoreboard, and a pass otherwise. The scoreboard tracks the results and prints the tests passed % at the end of the test. All testbench/DUT interactions are through Bus Function Model interface transactions.

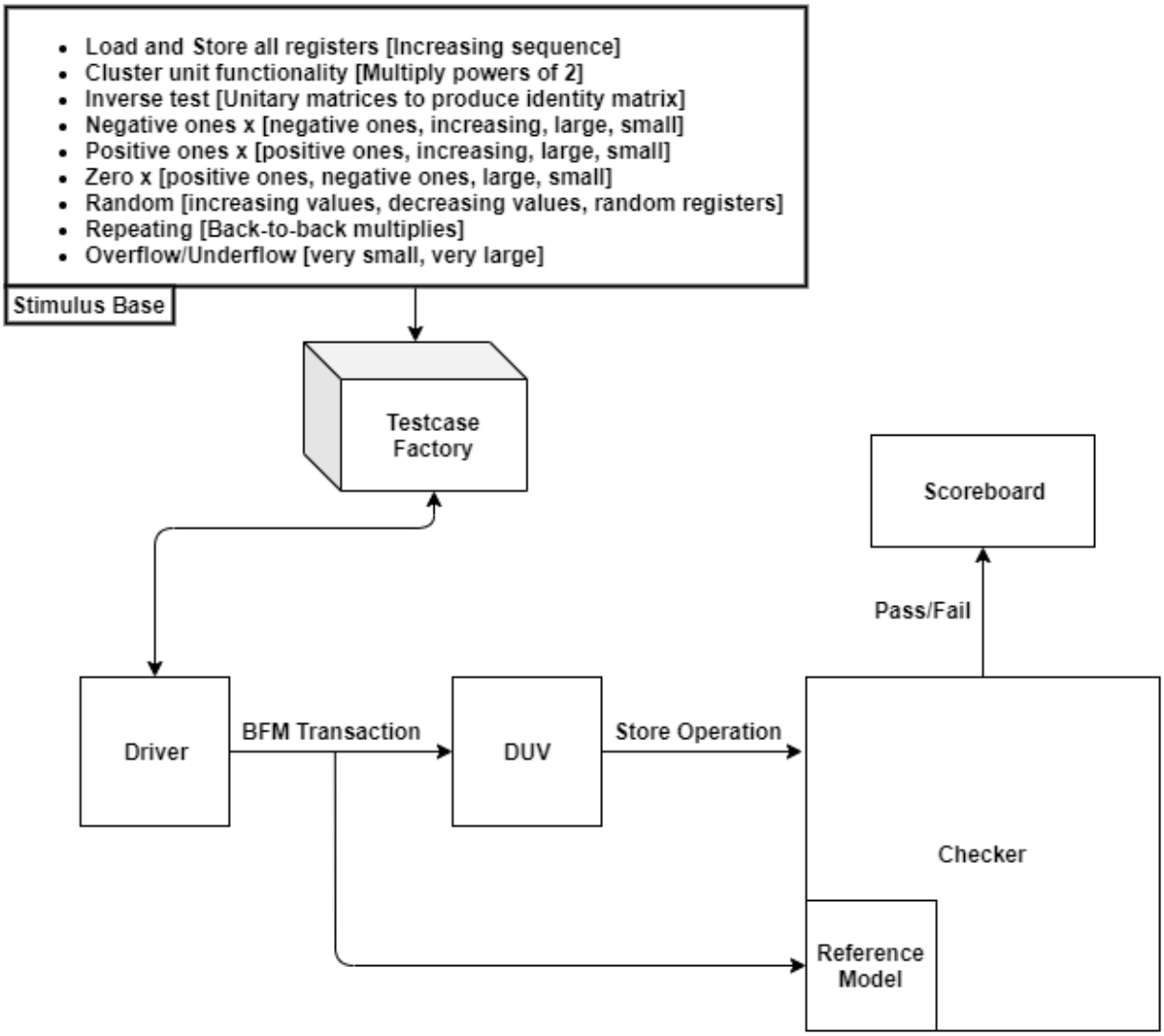

Figure 8 – Testbench Architecture

Emulator Configuration and Results

The results of the verification were achieved on the Veloce solo emulation platform. The emulator was operating in TBX mode with the use of a SystemVerilog interface that drove the required packed data to and from the design. There was the special addition of a pragma for a *ram _block* regarding the collector's buffer. The interface contained three more pragmas: *partition_interface_xif* that allowed the interface to be compliant with the emulator and two *tbx xtf* pragmas for the interface tasks regarding waiting for a reset and sending operations into the design. Since the testbench driver is the object that calls these tasks, it get *velanalyzed* along with the RTL with the option to *-extract_hvl_info*. The veloce.config file declares the interface bfm as *partition_module_rtl* and the DUT top and packages as *partition_module_xrtl*. The results from the emulator running the tests achieved a 99.0% pass rate for

randomly driven tests and a speedup of 29.5x was achieved for the repeat multiply test. The complexity and number of moving pieces in the design really lends itself to the parallelism available in the Veloce emulator. It will be interesting to see a comparison of speedup as the size of the matrices grow and the performance achieved by the emulator.

Figure 9 – Testbench Results

With the 99.0% pass rate expressed above, it is apparent that the emulator is still discovering some bugs. The main bug that was discovered was in the scenario where a product that is smaller (difference of at least 30.0) than the amount in the accumulator and the FMA unit doesn't normalize the two numbers quite right and the result is a binary magnitude larger than it should be. Another bug that was discovered was related to the proper accumulation of negative numbers within the FMA unit as well. **Figure 9** demonstrates that achieved speedup of the resulting testbench and design on the emulator. The largest set consisted of over 2,000,000 matrix load, multiply, and store operations. Running in simulation took 7,214 seconds while running the same test on the emulator only took 244 seconds resulting in a speed of about 29.5x.

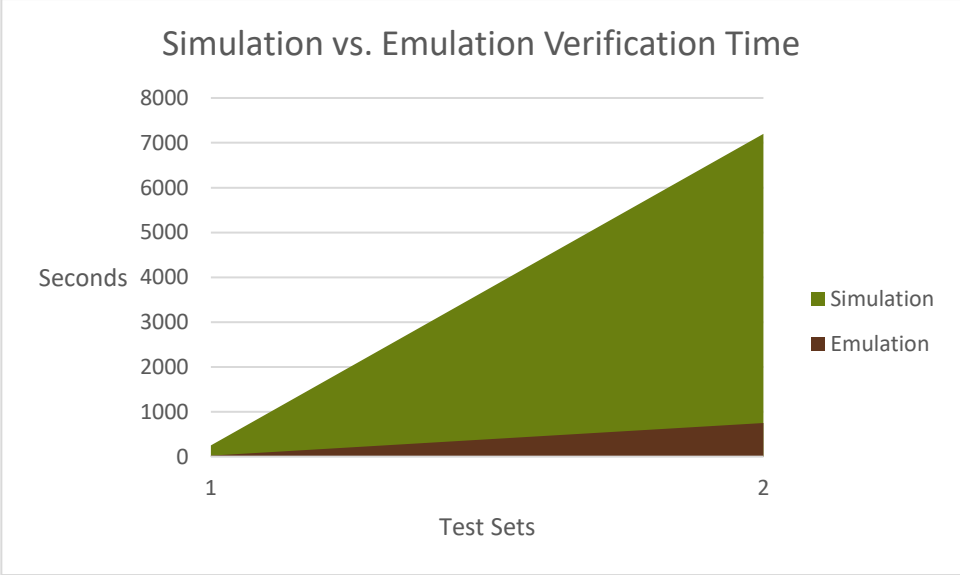

Figure 9 – Emulation Speedup

Conclusion

The emulator was a very interesting piece of technology to work with. It allowed me to really explore some areas of digital design that I was interested in. It was a lot of fun going back and forth between RTL design and creating the tests to verify my ideas all while getting to work on a computer nerd's dream machine. I hope to expand my design to compile a parameterized matrix size as well as add some more operations. For starters, I am planning on adding matrix scaling and addition/subtraction. I also want to add better capability to use the emulator to drive in tests at a faster rate. Part of what I learned when working with the emulator was the significance of instruction memory. The emulator helped show me

the performance bottleneck created when waiting for another instruction to arrive. Having an internal instruction memory would greatly increase performance when not loading/storing from memory. I think that last lesson is a valuable one to understand.

Citations

[1] IEEE Standard for Floating-Point Arithmetic," in *IEEE Std 754-2008* , vol., no., pp.1-70, 29 Aug. 2008

[2] Kumar, V., Grama, A., Gupta, A. and Karypis, G. (1994). *Introduction to Parallel Computing: Design and Analysis of Algorithms*. 1st ed. Redwood City, California: The Benjamin/Cummings Publishing Company, Inc.