**Approximate Squaring Circuit Report**

**Abstract**

The general idea of the squaring circuit comes from the usual 8x8 multiplier except with additional simplifications. If we consider an 8x8 multiplier with both inputs being the same vector, various simplifications can be performed to reduce the number of partial products from 8 down to 5. After this reduction we can use several Full Adders and Half adders to add up bits of the partial products to eventually reduce the 5 partial products to 2 partial products. The final 2 partial products can be summed together with one ripple carry adder. The process of reducing 5 partial products to 2 is based off of the Wallace tree multiplier which uses the same concept but on a full multiplier with 2 different input vectors rather than a squaring circuit with one input vector.

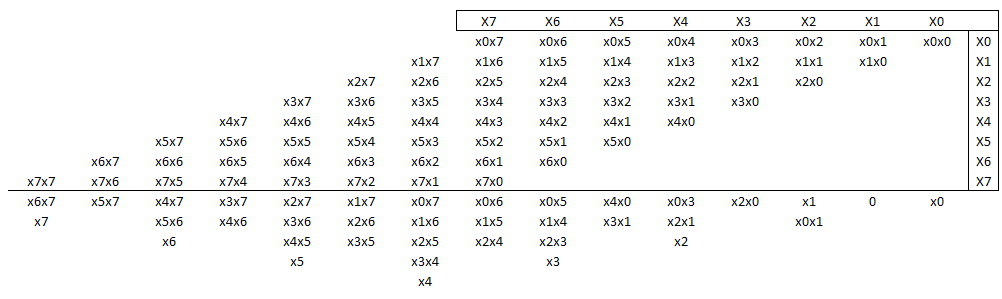
**Report Contents**

The next sections of this report will cover the following

1. Partial Product generation in squaring circuit
2. Partial product reduction with Wallace tree
3. Error analysis
4. Verilog Test Bench
5. Izhikevich Pattern Performance
   1. Patterns generated
   2. Error Metrics
   3. Error Results

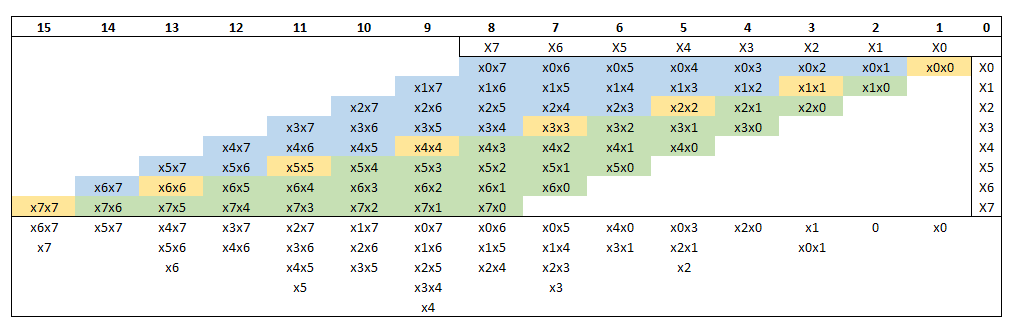
**Partial Product generation**

Partial Products can be generated normally as a regular multiplier by ANDing each input bit with the other and then shifting as in the fig. 1. However, summing the partial products us to simplify the partial products from 8 to 5 as expressed in Fig. 2.



***Figure 1***

In Fig.2 partial products generated come in pairs (blue and green). Since they’re basically the same AND operation performed on the same bits summing them will always produce a 0 output. Taking column 2 as an example. Adding x0x1 and x1x0 can be done using a half adder. The sum will equal (x0x1 XOR x1x0) which always produce 0. The inputs, however, have the potential to produce a carry which equals (x0x1 and x0x1) which reduces to (x0 AND x1). Following this line of logic, all partial products marked in blue have pairs marked in green. These pairs only generate carries for the next column and no sum bits. Only the yellow partial product bits produce a sum bit equal to the xi.



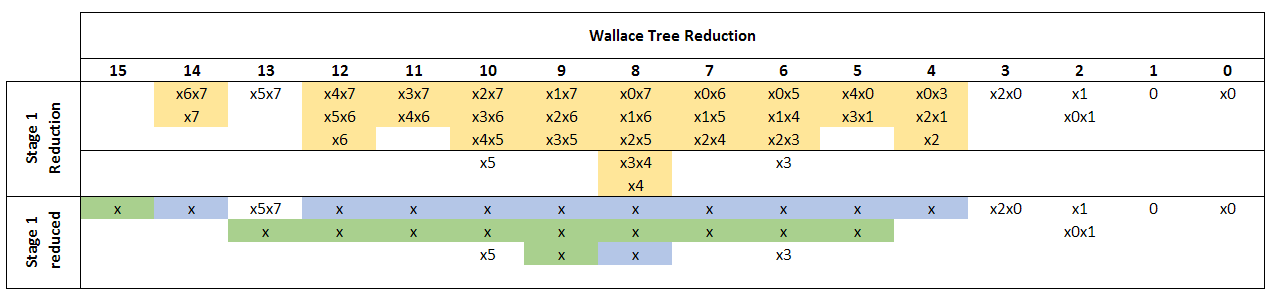
**Figure 2**

**Partial Product reduction with a Wallace Tree**

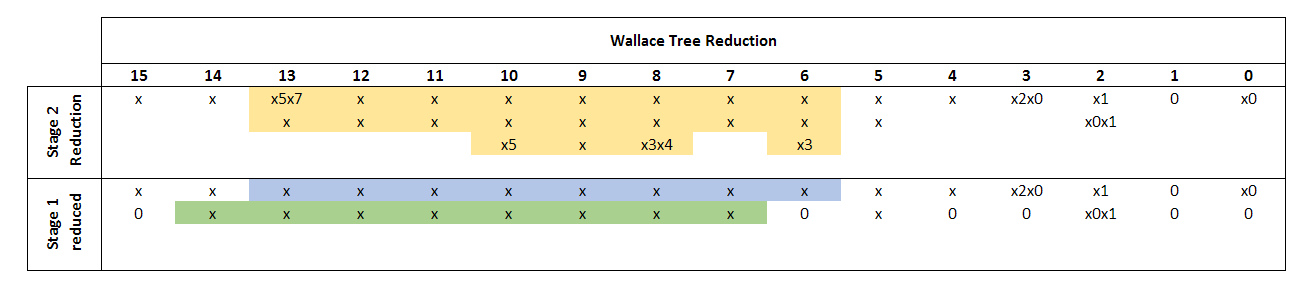
Partial product reduction is similar to simplifying binary addition operations by adding up columns of the partial products from right to left arbitrarily. It works because ultimately you are still adding up the partial products and producing sums and carrys that will ultimately produce 2 partial products added together using a regular ripple carry adder.

Partial product reduction happens in 2 stages, the first stage presented in Fig.3 reduces columns 4 through 14 using Full Adders, and Half Adders. The bits that will be reduced in this stage are marked in yellow. All reduced bits produce a sum bit sent to down to the same column (marked in blue) and a carry bit sent to the next column (marked in green). To maximize the number of bits reduced, the 5 partial products are segmented into two sets of 3 or 2 partial products, and then reduction is applied to both segments simultaneously. Since the ultimate goal is to bring the 5 partial products down to 2, all columns containing 3 partial product bits are reduced. Some columns with only 2 bits are also reduced (col. 5,11,14) however, this is done to limit the reductions that will be done in the next stage. All unreduced bits are sent to the next stage unchanged.

Fig. 4 Shows that the second stage of the Wallace tree reduction follows the same pattern except that its inputs are bits from the first reduction stage.



**Figure 3 First stage Wallace tree reduction**



**Figure 4: Second stage Wallace Tree reduction**

**Error Analysis**

The primary disadvantage of this squaring circuit is that it does not accept signed numbers, therefore, all signed numbers have to be converted into unsigned numbers prior to sending them to the squaring circuit. This does add additional hardware because the act of negating a 2’s complement number into an unsigned number does requires a ripple carry adder. To get around this one can approximate the negative 2’s complement number by ignoring the necessary one addition needed after inverting all the bits in the negative number. Since the purpose of this circuit is squaring we are certain that the result must be positive, so there is no need for any additional circuitry to correct the sign of the output based on the input. However, using negation without addition for negative numbers will add a measurable error to the final result. The analysis of the error is as follows:

Let be a negative binary 2’s complement number.

To negate it without adding one the following result ***Y*** will be produced:

Squaring the result:

Let ***E*** Be the error in the result

Let RED be the ratio between the error and the exact result

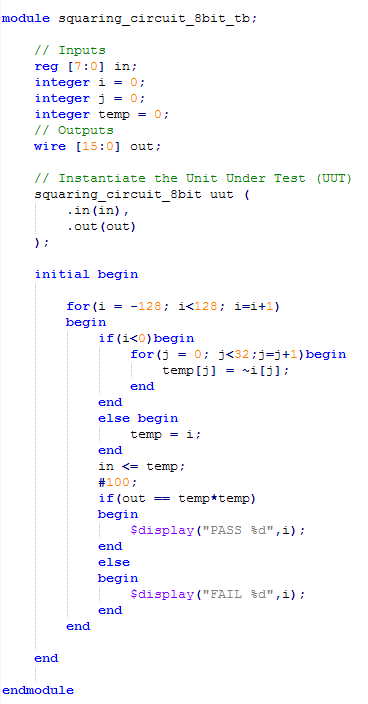
The worst-case RED will occur at A = -1 which will produce a ***RED = 100%***

Taking the limit as A approaches infinity

This implies that RED becomes negligible at higher A values, however we are constrained to 8 bits in this squarer so the minimum RED will be at ***A = -128*** which will produce a ***RED = 0.78%***

**Verilog Test Bench**

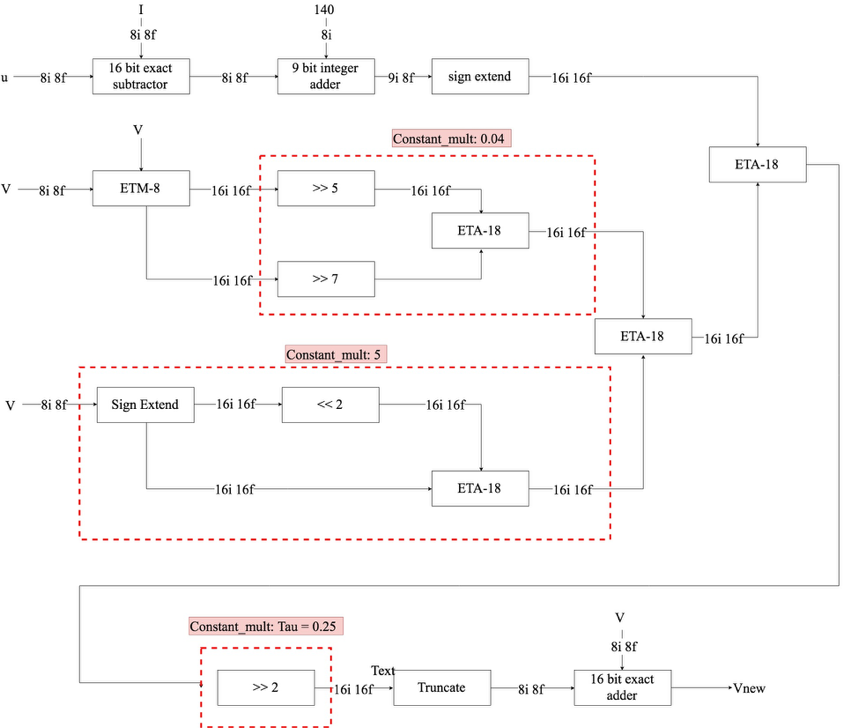
To following test bench was used to verify the accuracy of the squaring circuit while considering the approximations made for negative inputs. All inputs passed verification.



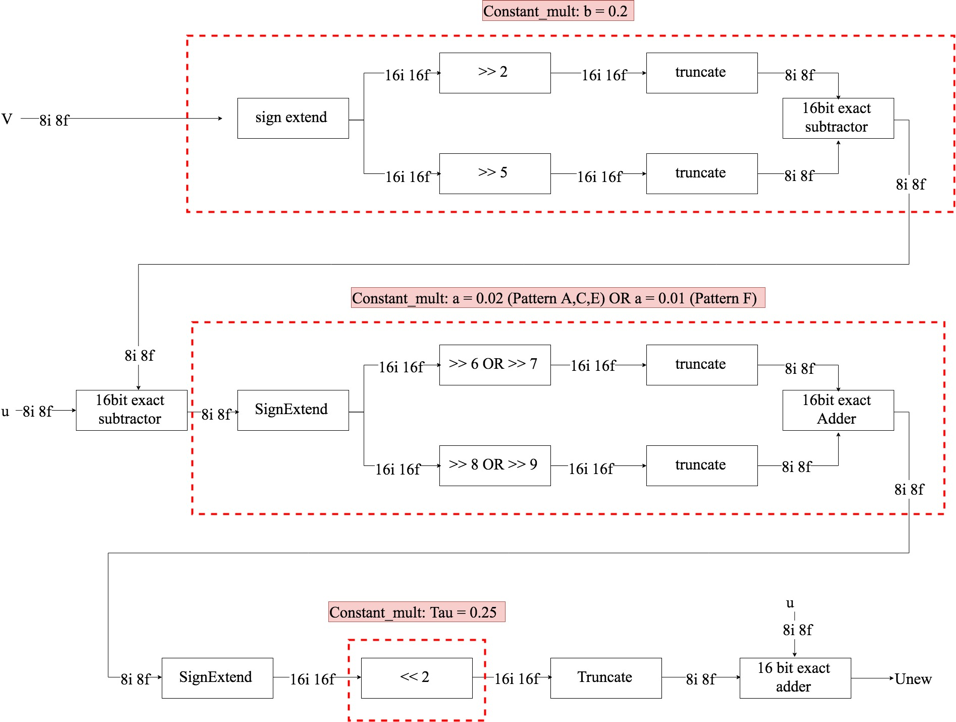
**Figure 5**

**Izhikevich Pattern Performance**

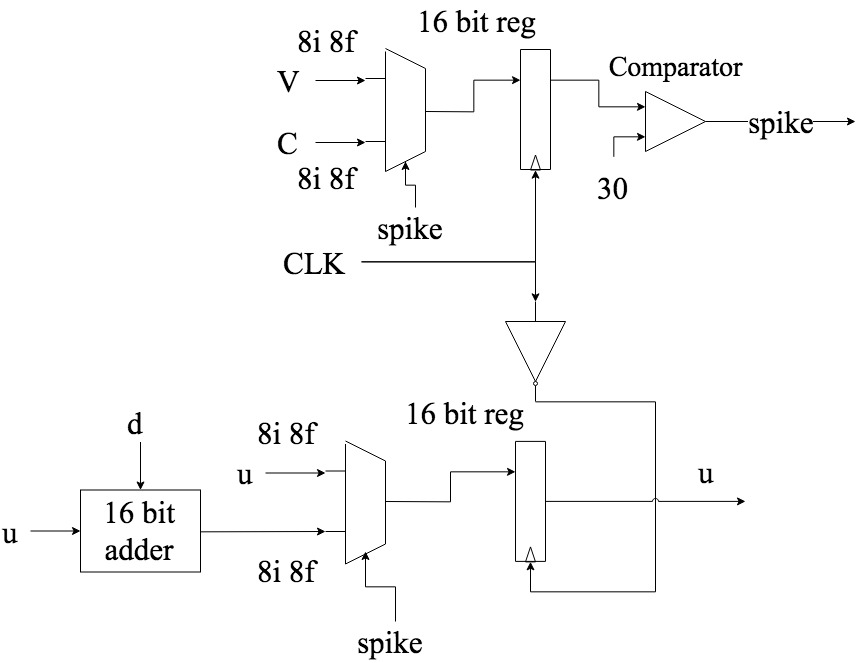
To further verify the functionality of the squaring circuit a Matlab testbench was built using the Izhikevich neuron architecture specified in fig. 6 and fig.7. The testbench was built using Matlab’s HDL Verifier package to simulate other hardware elements not yet built in Verilog. In fig. 8, four patterns were generated by the approximate Izhikevich model and are compared to the same patterns generated by the exact Izhikevich model.



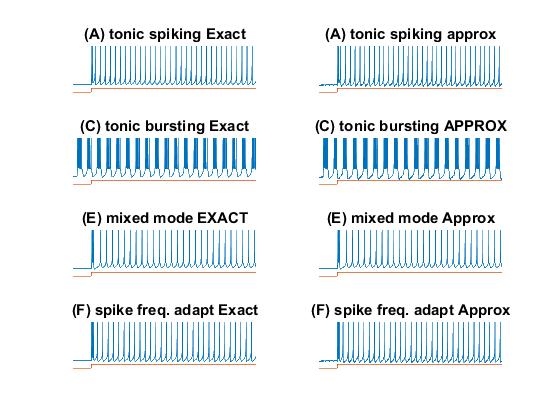
**Figure 6: V Architecture**



**Figure 7: U architecture**



**Figure 8: Spike detection circuitry**



**Figure 9: Four Spiking Patterns Generated**

**Izhikevich Error Metrics**

The following error metrics were used to evaluate the performance of the squaring circuit in the Izhikevich model.

1. **ERRp**

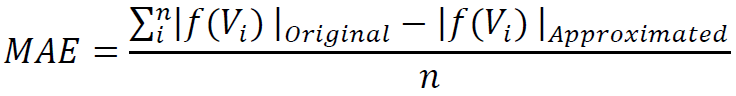
This metric is used to evaluate the Izhikevich model’s performance in the static case as in fig.10 where the current I and the recovery variable u are set to 0. ERRp is the difference between the minimum point of the parabola generated by the approximate and the exact model.



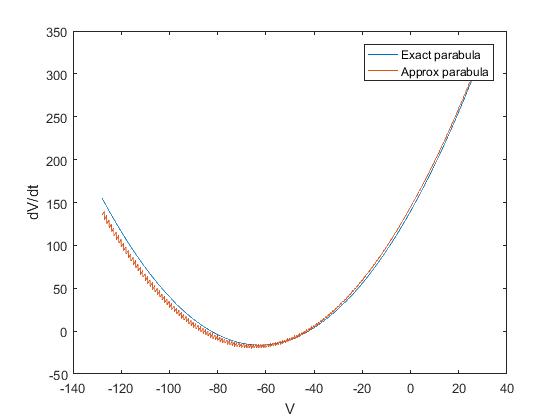
**Figure 10: Izhikevich model in the static case**

1. **MAE**

MAE Is the difference between the exact parabola equation and the approximate parabola



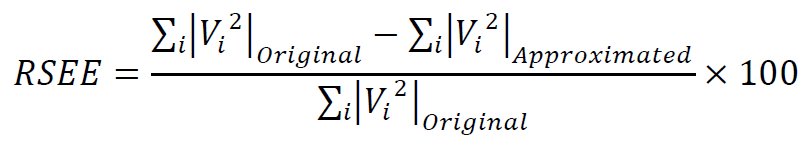
**Figure 11: MAE equation**

****

**Figure 12: Simulated parabolas**

1. **RSEE**

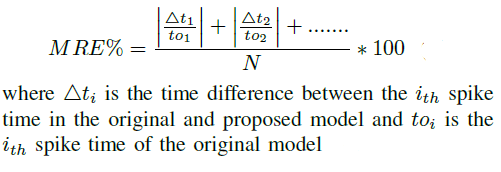
Because a minor shift in the generated patterns would yield a very high mean square error, RSEE is used instead to compare the patterns. RSEE evaluates the relative spike energy which is the difference in spike energy between two generated patterns.



**Figure 13: RSEE equation**

1. **MRE**

MRE calculates the difference in the spike time between the original model and the proposed model.



**Figure 14: MRE error**

**Izhikevich Error Results**

The following table details the evaluated errors previous mentioned. Three things to note:

* The constant 140 has been increased to 145 to compensate for the underestimation resulting from squaring negative numbers.
* RSEE and MRE were evaluating by generating each pattern for 1000ms
* The approximate model may not generate exactly the same number of spikes as the exact model, additional spikes generated at the end of either were ignored when calculating MAE

|  |  |
| --- | --- |
| MAE | 0.691 |
| ERR | 3.94 |

|  |  |  |
| --- | --- | --- |
| Pattern | RSEE % | MRE % |
| A | 6.57 | 4.47 |
| C | 13.59 | 9.37 |
| E | 13.07 | 19.76 |
| F | 5.12 | 1.45 |