FAULT DE-INTERLEAVING FOR RELIABILITY IN HIGH-SPEED CIRCUITS

By

Kevin Dick

Thesis

Submitted to the Faculty of the

Graduate School of Vanderbilt University

in partial fulfillment of the requirements

for the degree of

MASTER IN SCIENCE

in

Electrical Engineering

May, 2011

Nashville, Tennessee

Approved:

Professor Jeffrey D. Black

Professor William H. Robinson

ACKNOWLEDGEMENTS

I would like to first thank my advisor Professor Jeffrey Black who helped guide me through the process. Professor William H. Robinson provided great feedback and suggestions on how to improve this thesis. I would also like to thank Dolores Black who has been very helpful to me in times of need. This would not have been possible without Andrew Sternberg who helped to set up and run the simulations. John Ahlbin and Andrew Sternberg kept polarbear working so I could run my simulations. I would like to thank all of the professors and students from the Radiation Effects and Reliability Group who have been very kind when I have had questions. I would also like to thank my family for their loving-kindness and support.

TABLE OF CONTENTS

Page

LIST OF TABLES

Table Page

LIST OF FIGURES

Figure Page

CHAPTER I

INTRODUCTION

1.1 Statement of Research Problem

The majority of modeling and simulation of single event transients (SETs) and their effects is done at a transistor level, known as micro-modeling. These simulations give insight into how a transistor will behave when struck by SETs. The micro-modeling of single-particle effects in Integrated Circuit (IC) devices can simulate charge collection, charge deposition/generation and ionic interaction with the semiconductor material. Micro-modeling works very well on a small circuit, when there are a low number of transistors. However, modeling large circuits on a transistor level can be time- and cost-consuming. System-level modeling and simulation allow for large circuits to be simulated with less time and less cost. Under the correct conditions, the results of (system-level) macro-modeling can approximate the results of micro-modeling.

Simulating SETs can be done by using transient fault injection. Transient fault injection is the method of injecting a fault on a certain node and observing what happens as it propagates through the circuit. This method can be used to study the effects of SETs that last longer than one clock cycle [1] and how they affect high-speed applications such as the radio frequency (RF) mixer in a software-defined radio [2-3].

The goal of this research is to study the effects that SETs lasting longer than one clock cycle have on a parallel pipelined multiplier, implemented with parallel and serial addition. The approach uses a transient injection methodology to evaluate circuit architectures and then mitigates the effect of SET-induced errors by spacing them out in time so that they all do not occur together in a burst.

1.2 Overview of thesis

The flow of the thesis is to discuss: (1) the background of SETs, (2) the simulation of SETs and burst-error mitigation approaches used in communication theory, (3) the simulation of SETs in a parallel full-adder-implemented multiplier and a serial full-adder-implemented multiplier, and (4) the application of the multipliers to the mixer of a software-defined radio. Chapter II discusses the software-defined radio, generation of SETs, the effects they have on the circuits, the effects of SETs that last longer than one clock, and the approach that is used in communication systems to mitigate a burst of errors. The simulation of SETs will discuss the system-level modeling approaches for propagating and capturing SETs. A similar approach to mitigating errors in communication systems will be used to mitigate SETs lasting longer than one clock cycle. Chapter III discusses two 4x4 multipliers, one implemented with parallel adders and another with serial adders. The method of setting up each multiplier is discussed along with simulations demonstrating the operation of each multiplier and the effects of SETs. Both multipliers will simulate long (i.e., lasting longer than one clock cycle) SET effects in a mixer of a software-defined radio. Chapter IV discusses the results for a

16x16 multiplier with parallel-implemented full adder circuitry and a 16x16 multiplier with serial-implemented full adder circuitry. The same method for the 4x4 multipliers is used to build the 16x16 multipliers, and the results from the simulations are given along with a discussion of their effectiveness and future implication. Chapter V summarizes the conclusions from the research and simulations.

CHAPTER II

BACKGROUND

2.1 Software-defined Radio

The software-defined radio is a radio communication system where waveforms are defined using software instead of hardware (Figure 1). The ideal software-defined radio would receive an analog signal, convert it to a digital signal, run the signal through the required applications, and then convert the digital signal back to an analog signal.

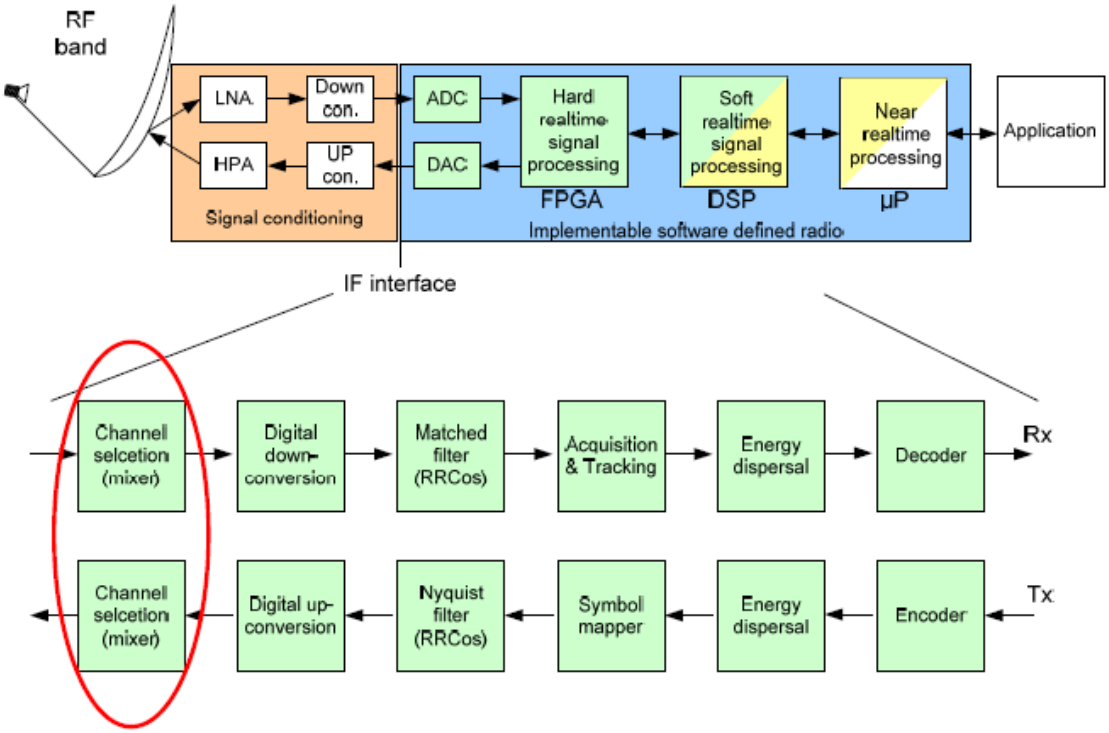
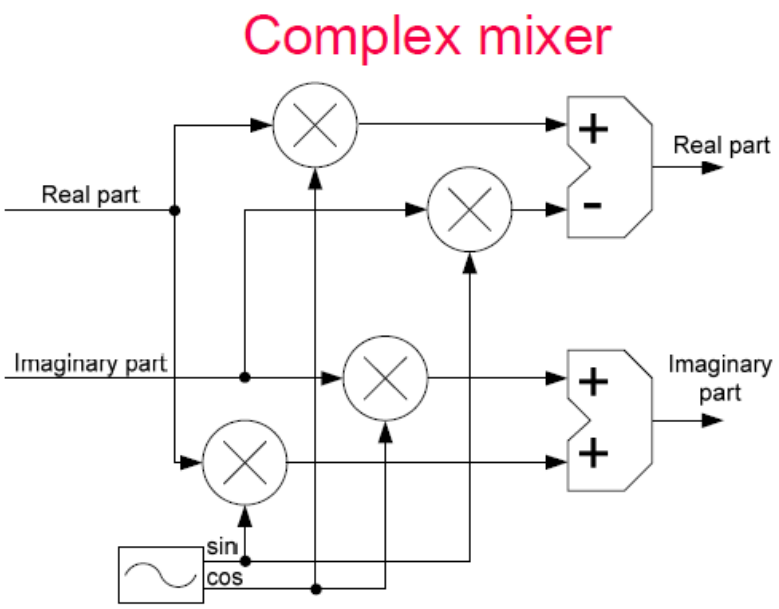


Figure 1: Software-define radio block diagram (after [4])

Most software-defined radio receivers have a variable-frequency oscillator, filter, and a mixer. The mixer for the software-defined radio, circled in Figure 1, deals with two different inputs. One of the inputs is the carrier wave, and the other is the data signal. The signals in the mixer are often multiplied together to add and subtract the frequencies of the signals. A complex mixer is shown in Figure 2. The components for the complex mixer are a multiplier, a look up table for the cosine and sine functions, and an address counter.



$$(\mathrm{Re} + j\,\mathrm{Im}) \cdot e^{j\omega t} = (\mathrm{Re} + j\,\mathrm{Im}) \cdot (\cos\omega t + j\sin\omega t)$$
$$= (\mathrm{Re} \cdot \cos\omega t - \mathrm{Im} \cdot \sin\omega t) + j(\mathrm{Re} \cdot sin\omega t + \mathrm{Im} \cdot \cos\omega t)$$

Figure 2: Complex mixer (after [4])

2.2 Single Event Transients (SETs)

SETs are the result of the unwanted generation of electron-hole pairs (EHPs) in the semiconductor [5]. The interaction of an ionized particle with the semiconductor is called a single event (SE). The generation of EHPs is the result of ionizing particles interacting with the semiconductor through Rutherford's scattering. The ionizing particles can be galactic cosmic rays, energetic protons, neutrons, or alpha particles [6]. The ion slows down and deposits energy as it moves through the lattice of the semiconductor.

The slower the ion moves, the more energy it deposits. The linear energy transfer (LET) is a measure of how much energy is produced by the ion as it travels through the semiconductor. This transfer of energy from the ionizing particle to the semiconductor leaves a path of EHPs. This SE occurrence is spatially and temporally random [7].

SEs on a circuit can result in charge collection on nodes in that circuit. The collection of charge on a circuit can cause a single event upset (SEU) or a SET, both potentially providing an unwanted change in the information at a given node. An SEU occurs when a change of state happens in a memory circuit in which it does not recover. An SET occurs when a subatomic particle deposits charge at a node resulting in a change in the transient voltage. The effect of an SET in digital electronics depend on the vulnerability of a node, active combinational logic path, the length of the latching window, propagation delay along the path, and pulse shaping [8]. A node will be affected by an SE depending on how the SE deposits charge, the mechanisms of its collection, and how the information is propagated through the circuit. The active combinational logic path is determined by the operation of the circuit. The length of the latching window is based on the timing characteristics of the latch at the end of the combination logic path and determines whether the SET is latched or not. A soft fault is an upset that occurs within the circuit. An error is when the soft fault causes a corruption of the output. The probability that the soft error is observable at the system output is given by the equation [9]:

$$P_{C,N}^{\mathrm{SF}} = P_N^{Pulse} \cdot D_{C,N,L}^{Prop} \cdot P_{N,L*}^{Storage} \quad (1)$$

where,

P_N^{Pulse} is the probability given a SE strike depositing Q_{coll} at a random location anywhere within the total circuit area, that node N will generate an output perturbation above the logic noise margin and thus produce an erroneous logic signal,

$D_{C,N,L}^{Prop}$ is a deterministic measure that, given an erroneous signal originating at node N during clock cycle C, the signal will propagate to a latch L, and

$P_{N,L}^{Storage}$ is the probability that a randomly arriving logic signal along an active path from N to L will corrupt the latch L and $L*$ is the latch for which $P_{N,L}^{Storage}$ is a maximum among multiple active paths from N to multiple latches.

As the size of transistors scale down, the more likely SETs will affect a circuit. Each node stores charge, and as the size gets smaller, the storage charge reduces, meaning the same SET will have a greater impact on that node. If the charge for the node decreases, then the voltage must increase, according to the formula:

$$\Delta q = C * \Delta V . \quad (2)$$

The charged particle does not scale down as the transistor size scales down. This means that if the capacitance for each node decreases, then the voltage resulting from an SE must increase, leading to increased circuit vulnerability.

2.3 SET Simulation in High-Speed Circuits

A system-level modeling approach is helpful for understanding how an SET will affect the output of a system. When an SET occurs in high-speed circuits, they can affect the output of a system over multiple clock cycles.

2.3.1 System-Level Modeling

System-level modeling uses behavioral or rules-based techniques to model electrical systems, in order to determine if the system is functional. This technique is most effective in large digital circuits. Mathematically-described parameters are used to analyze single event effects (SEEs). An SEE is caused by single, energetic particles that have effects on the circuit. These systems can even be used to simulate time-dependent effects.

There are several reasons to use system-level modeling. One reason is a case in which the regularity assumptions do not hold. The function of one logic cell cannot be applied to the whole. Another reason is that out of all the possible paths, there are only certain paths that will be active at the time of the SET. The paths that are active depend on the timing and the inputs. A third reason is that a single hit can cause multiple errors (Figure 3).

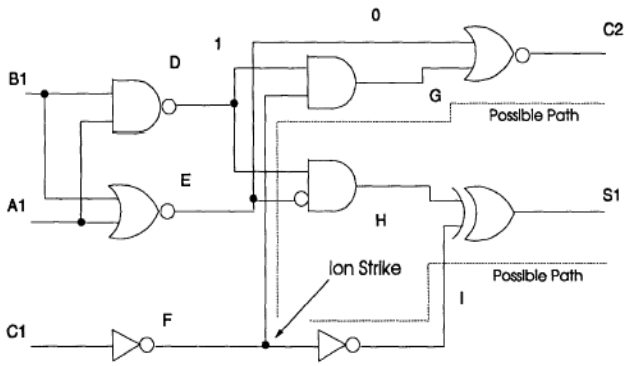


Figure 3: Circuit showing multiple paths following a SE hit (after [10])

The SEU_TOOL is a multimode approach to simulating circuits [9]. The SEU_TOOL looks at the probability of each node causing a fault and the observability of the soft error. An analysis was done on a large circuit, shown in Figure 4, to determine the relative size of the error cross section of the different components. The arithmetic logic unit (ALU) was found to have the largest error cross section (Figure 5).

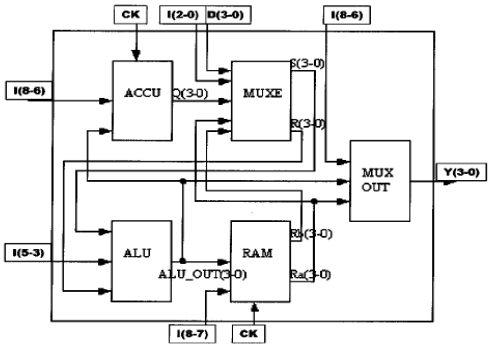


Figure 4: Block diagram of circuit (after [9])

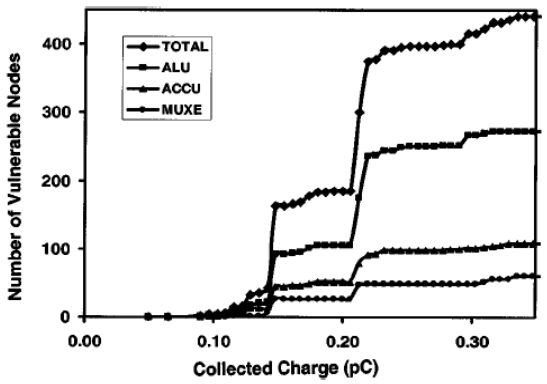


Figure 5: Relative size of error cross section (after [9])

Fault injection consists of introducing SETs into the system at a given node, and time to observe if it leads to an error. This is useful for predicting the processor error rate for a program, which allows for full application to be tested instead of representative benchmarks. With enough samples, the average number of fault injections to produce an error can be determined. The cross section of observable single event upsets (SEUs) is given by [11]:

$$\sigma_{CEU}(application) = \tau_{CEU} \times \sigma_{SEU}, \tag{3}$$

where τ_{CEU} is a circuit-dependent constant.

A simulation was performed, using fault injection and actual data taken for code emulating upsets (CEU) [7]. Figure 6 shows the results without the use of the cache. Figure 7 demonstrates the results with the use of the cache. Cache is memory that can be accessed very quickly in a couple of clock cycles. Information can be read from or written to the cache. Storing information in the cache instead of having to compute it again or access it from its original storage location improves the overall performance of

the system. The predicted and measured values do not follow more closely with the use of the cache, because the instruction set did not address the cache memories [11].

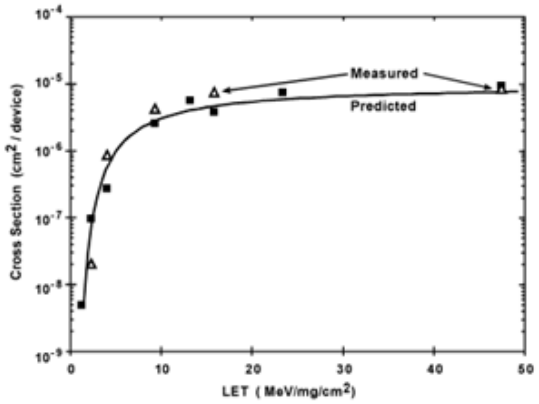


Figure 6: CEU prediction and measured results without the use of the cache (after [11])

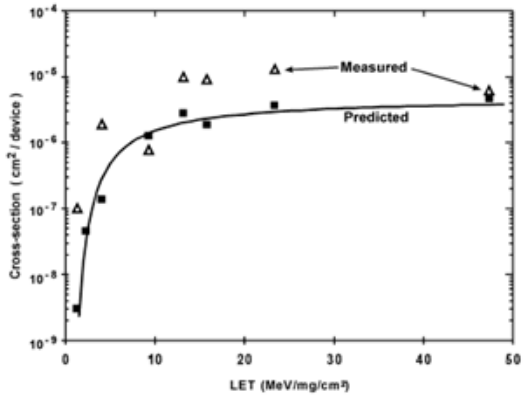


Figure 7: CEU prediction and measured results with the use of the cache (after [11])

2.3.2 SETs Longer Than One Clock Cycle

In circuits applying faster and smaller devices, SETs lasting longer than one clock cycle may be a potential problem. SETs lasting longer than one clock cycle can affect consecutive values for a single node. Instead of affecting just one value of information on a node, it could affect two or more bits of information on that one node. This is a concern for circuits, like the software-defined radio, that are producing radio frequency signals. This allows the effects of the SET to extend to events over multiple clock cycles.

With SETs lasting longer than one clock cycle the mitigation technique of triple modular redundancy can be used, but it is expensive to implement in terms of the size and power for the circuit. New mitigation techniques will probably need to be developed to handle these cases. Simulations can help determine which techniques will be effective.

To simulate SETs lasting longer than one clock cycle, two NOR gates can be applied for each output in the element of the logic library (Figure 8). Logic library elements, such as the NAND gate and a full adder, have built-in circuit delays. This means that each element will have a delay from the input to the output. The two NOR gates, however, are primitive gates and have no delay between the input and the output. The first NOR gate has the output for that node and an input signal **SETH**. If the input signal **SETH** is logic high then the output for that node will be driven high for as long as the input signal **SETH** is held high. If the input signal **SETH** is logic low, then the output will give the normal value. The inputs for the second NOR gate are the output from the first NOR gate and an input signal **SETL**. If the input signal **SETL** is driven logic high, then the output for that node will be low for as long as the input signal **SETL**

is held high. If the input signal **SETL** is logic low, then the node will give the normal value.

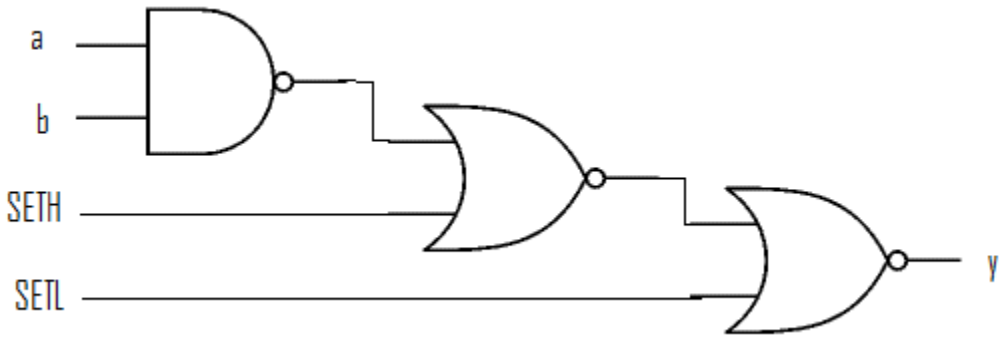


Figure 8: NAND gate with fault injection

The NAND gate will operate normally if both the input signals **SETH** and **SETL** are held low, which can be seen below (Figure 9).

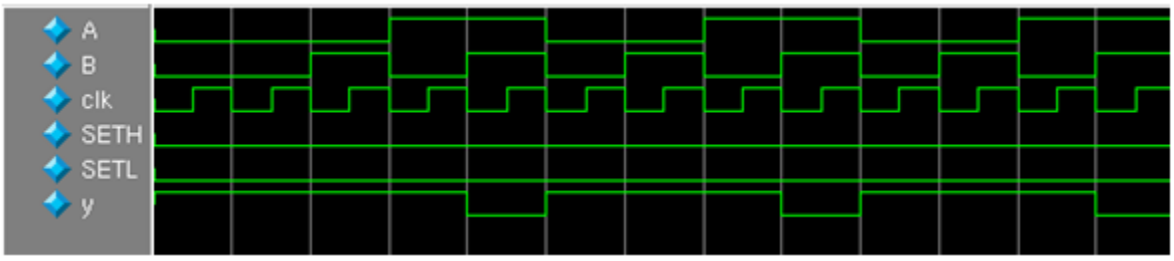


Figure 9: Normal operation of NAND gate

When the input signal **SETH** is set to logic high the output of node ***y*** is driven high and will last as long as the input signal **SETH** stays high (Figure 10). On average 25 percent

of the inputs will be upsets when the input signal **SETH** is set to logic high. This is because out of the 4 possible combinations of inputs, only 1 of them will lead to a low output.

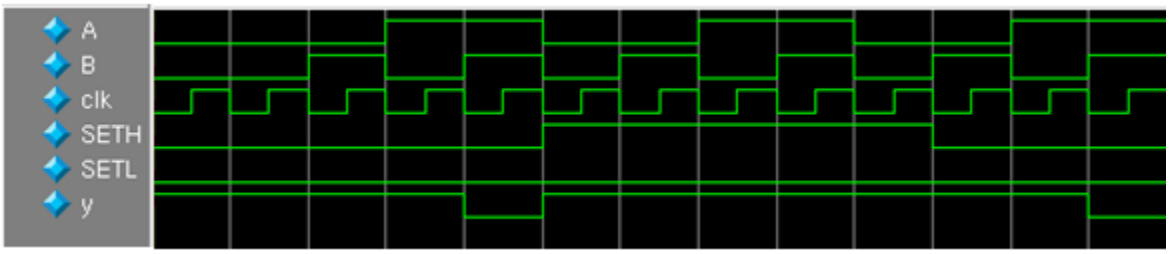


Figure 10: SETH held high for 5 clock cycles

When the input signal **SETL** is set to logic high, the output of node ***y*** is driven low and will last as long as the input signal **SETL** stays high (Figure 11). On average, 75 percent of the inputs will be upset when the input signal **SETL** is set to logic high. This is because out of 4 possible input combinations, 3 of them will lead to a high output.

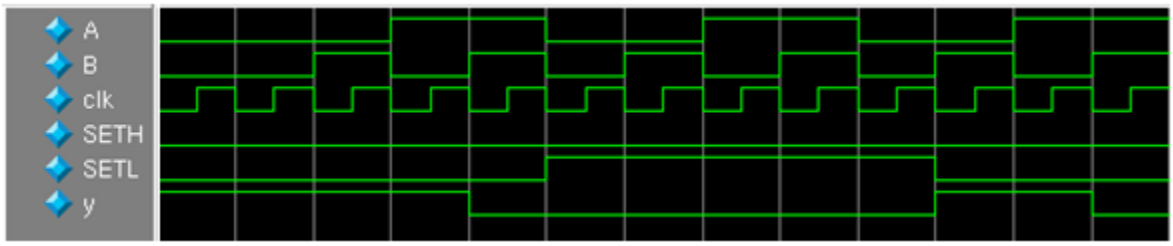


Figure 11: SETL held high for 5 clock cycles

With the previously-described circuit modifications, any node in a circuit can be simulated as an SET lasting longer than one clock cycle. This does not just flip the bit of the outputs but holds the node either high or low to simulate an SET strike that lasts longer than one clock cycle. When the input signal **SETH** in the first NOR gate is high, no matter what the inputs are for the NAND gate, the output, ***y***, will be high. The output will remain high as long as the input signal **SETH** is high. When the input signal **SETL** in the second NOR gate is high, no matter what the inputs are for the NAND gate the output, ***y***, will be low. The module and the testbench for the NAND gate are in Appendix A and Appendix B.

2.4 Mitigating Bursts of Upsets in Communication Systems

Communication systems operating in environments like the ionosphere can experience bursts of errors that can cause serious problems. Communication systems use a process called interleaving/de-interleaving to spread the burst of errors out and mitigate them. If there are too many errors that occur in a word, then the word cannot be correctly decoded. Therefore, interleaving (rearranging the bits) is performed before the code is transmitted. When the code is received, de-interleaving is performed to obtain the original code.

Figure 12: Interleaving/De-Interleaving

2.4.1 Error Interleaving/De-Interleaving

In general, errors are easier to mitigate if they do not occur in bursts. Analog filters on the output of the software radio should remove anomalies that occur in one clock cycle and do not persist. But when SEUs occur in bursts, they may overwhelm the filter's abilities. Similarly in communications, burst of errors can overwhelm forward error correction (FEC) capabilities of communication signals [3]. The burst errors can be due to fading channels. There have been many studies done on the effects of errors in communication signals and how they can be minimized [12]. The process of interleaving and de-interleaving can spread out the burst of upsets so that they can be handled individually. Suppose there is a stream of binary words, $a_1a_2a_3a_4b_1b_2b_3b_4c_1c_2c_3c_4d_1d_2d_3d_4$. A binary word is a group of bits that occupy a single storage address and that the computer treats as a unit. A binary word would be $a_1a_2a_3a_4$. The process of interleaving takes the bits and converts them into matrix form and then reads them out serially. The bits can be read into a matrix form:

$$a_1a_2a_3a_4$$

$$b_1b_2b_3b_4$$

$c_1 c_2 c_3 c_4$

then read out serially as $a_1 b_1 c_1 a_2 b_2 c_2 a_3 b_3 c_3 a_4 b_4 c_4$. When the code is received, de-interleaving allows the upset to be spread out instead of occurring in a burst. A burst of upsets can be a problem in certain environments, and the effectiveness of mitigation techniques are shown in the simulations in Section 2.4.2. Below is shown a code and the process of interleaving and de-interleaving with a burst of 4 errors, $b_2 c_2 d_2 a_3$, represented by underscores. The size of the interleaving process determines how far apart the errors are located.

Code: $a_1 a_2 a_3 a_4 b_1 b_2 b_3 b_4 c_1 c_2 c_3 c_4 d_1 d_2 d_3 d_4$

Code after interleaving: $a_1 b_1 c_1 d_1 a_2 b_2 c_2 d_2 a_3 b_3 c_3 d_3 a_4 b_4 c_4 d_4$

Received code with a burst of errors: $a_1 b_1 c_1 d_1 a_2 \_\_\_\_ b_3 c_3 d_3 a_4 b_4 c_4 d_4$

Code after de-interleaving: $a_1 a_2 \_ a_4 b_1 \_ b_3 b_4 c_1 \_ c_3 c_4 d_1 \_ d_3 d_4$

A burst of random changes or scintillations in the signal can be created in the ionosphere and can even make the signal unrecognizable. If a system is not designed to handle these scintillations properly, then the performance of the system will suffer [13]. There are various degrees of scintillations. Some are slow; while alteration to the signal occurs, it does not make it completely unrecognizable. The faster scintillations can make the signal unrecognizable. The effectiveness of any digital communication system in filtering out or reducing the effects of these scintillations in the signal is important for its ability to get an accurate reading of the signal.

2.4.2 Simulating the Effects of the Scintillations of Signals

Going up to the ionosphere and running tests to study the scintillations of signals is not really feasible. Simulating the effects of the scintillations of signals can be done through software products like Matlab® and Modelsim®. These software packages allow us the ability to represent the scintillations of signals and find ways to mitigate the effects of the scintillations.

The simulations in this section were performed to see what benefits different encoders, decoders, and environments have on the reduction of errors in the communication signals. Appendix C contains an example of a simulation performed in Matlab® with a 1-to-2 encoder and a, Viterbi decoder using an environment that produces single errors as well as burst errors. A Viterbi decoder uses the Viterbi algorithm [23], which uses the convolutional code based forward error correction (FEC) technique to decode bitstreams in a signal.

The input data of the code lets the user select the number of samples for the input. Then a random set of 0's and 1's is created according to the value the user entered (Figure 13).

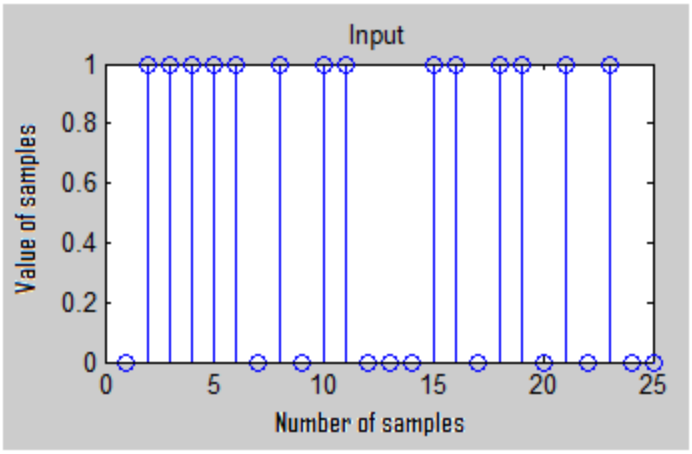


Figure 13: Random input signal

The random set is then encoded. Depending on the encoder selected, different numbers of outputs will be produced. This 1-to-2 encoder will produce twice as many outputs as inputs. The encoder works by taking the inputs and putting them through a shift register, which then takes selected inputs and performs a modulo-2 addition on them to get the outputs (Figure 14).

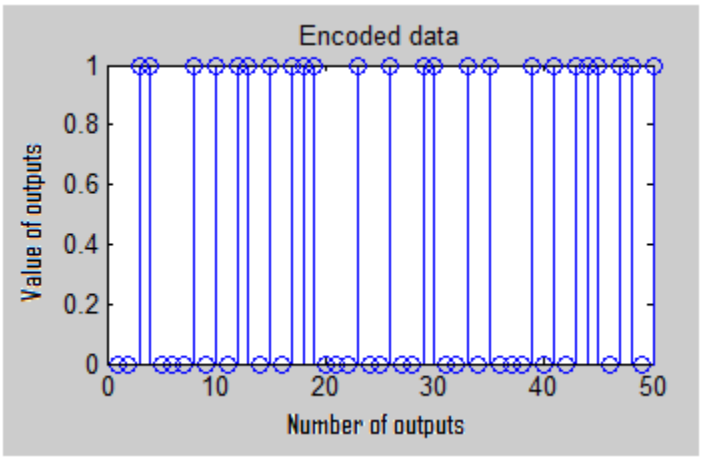


Figure 14: Encoded signal for a 1-to-2 encoder

The noise is produced by randomly generating the same number of values as the output of the encoder. The random numbers are then changed into either 0's or 1's depending on their value (Figure 15). The code goes through a loop and compares the random numbers to the values. Those values that are greater than a selected threshold for the noise are changed to 0's and those values that are between the threshold and another selected threshold are changed to 1's, to simulate the noise. The selected values are set by the user depending on the percentage of noise that is desirable for simulation. A burst happens when the random number falls below both thresholds. The burst generates a random set of 16 values between 0 and 1, which then are changed into 0's and 1's through a comparison of the values to a selected value. Those above it are changed to 0's and those that are below it are changed to 1's. These values are put into the noise by overwriting the next 16 values.

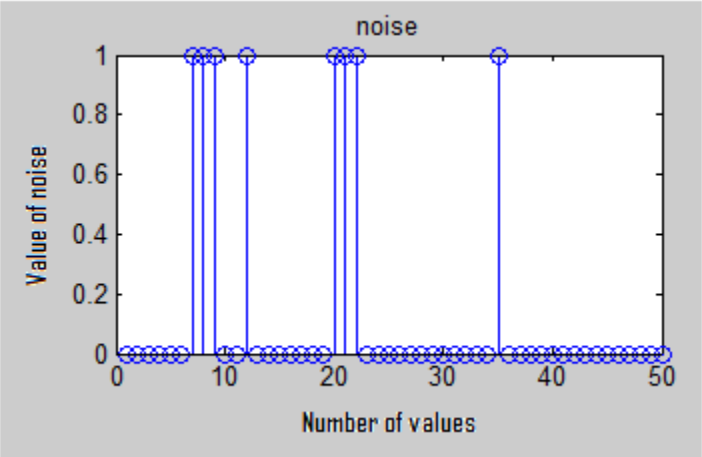


Figure 15: Random burst of noise

The noise and encoded signal are then added together (Figure 16). This is accomplished by performing a modulo-2 addition (Figure 17). The signals are first added together, and then those values that result in a 2 are changed to 0.

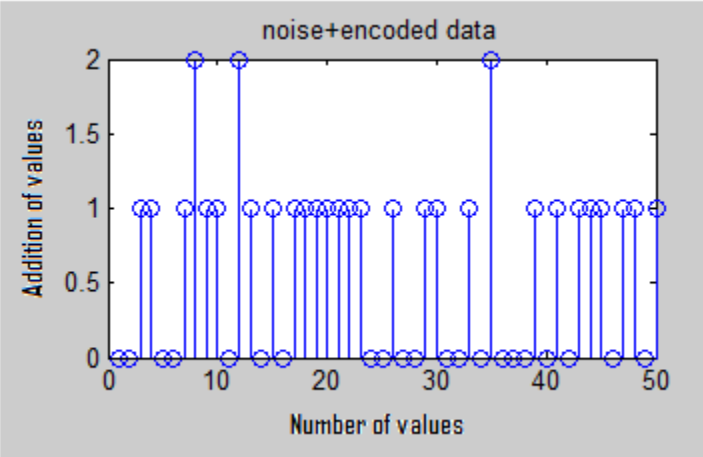


Figure 16: Noise plus encoded signal

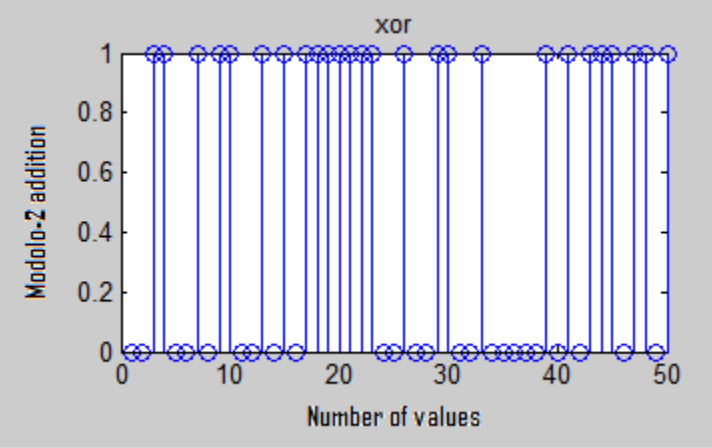


Figure 17: Modulo-2 addition

The data is then quantized to prepare it for the decoder. The Viterbi decoding is performed on the data (Figure 18). The process of encoding and decoding is used to eliminate some of the errors. The bit error rate is found by comparing the input signal to the decoded signal. The delay in the decoded signal is taken into account, and the number of times the signals differ is recorded and divided by the total number of compared points to get the error ratio.

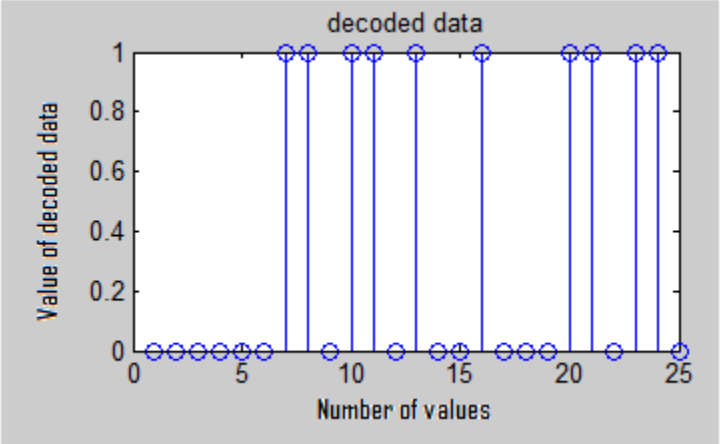


Figure 18: Decoded signal with shift

Simulations for 10,000 samples were performed for three different types of encoders (1-to-2 encoder, 2-to-3 encoder and 3-to-4 encoder) (Figure 19 and Table 1). Table 1 lists the encoders along the right hand side of the table. The inputs errors are listed on the top under the title and the output errors are listed under each input error corresponding to the encoder that was used on the right hand side. Simulations were also performed for signals with no burst of errors in the noise and those that did have a burst of errors in the noise. The selected input error was then compared to the output error in

order to determine how effective the encoder and decoder were at reducing the amount of errors.

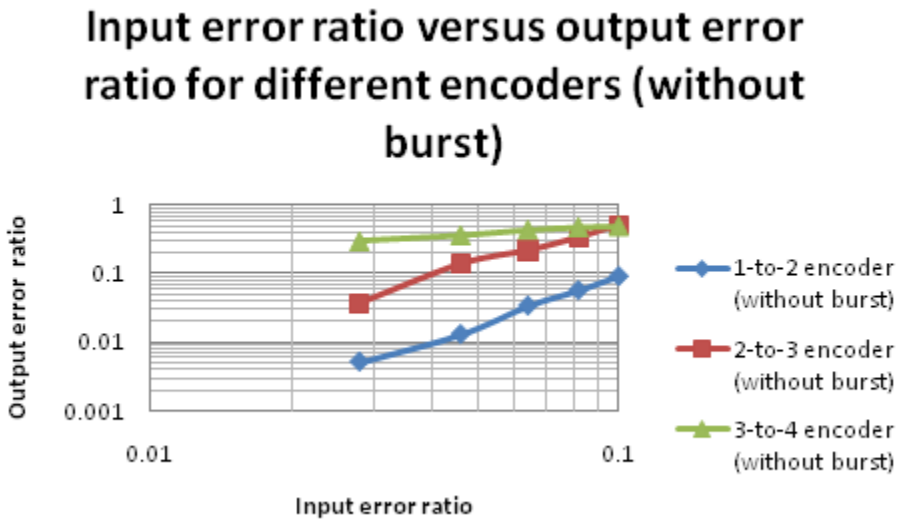


Figure 19: Input error ratio versus output error ratio

Table 1: Input error ratio versus output error ratio for different encoders (without burst)

Input error ratio versus output error ratio for different encoders (without burst)						
	Input error					
	0	0.027	0.045	0.062	0.081	0.101
1-to-2 encoder	0	0.005	0.013	0.034	0.056	0.091
2-to-3 encoder	0	0.037	0.137	0.215	0.333	0.507
3-to-4 encoder	0	0.293	0.359	0.435	0.479	0.496

The results for the encoders without burst are compared to those with burst in the noise and a significant difference can be seen in the effectiveness in the reduction of output errors for the simulations without an input burst of errors. The simulations are

done for 10,000 samples in order to give the random burst of noise to a chance to stabilize and give consistent results (Table 2). The number of errors with the burst has about 3,000 more errors in it than it does without the burst.

Table 2: Number of errors for encoders with and without bursts

Number of errors for encoders with and without bursts	
Number of samples	10000
1-to-2 encoder (without burst)	6
2-to-3 encoder (without burst)	37
3-to-4 encoder (without burst)	126
1-to-2 encoder (with burst)	670
2-to-3 encoder (with burst)	1433
3-to-4 encoder (with burst)	3106

CHAPTER III

4x4 BIT MULTIPLIERS

3.1 Introduction

To demonstrate how circuit design architecture can mitigate upsets occurring in bursts by spreading them out, two different implementations of multipliers will be used. Both multipliers will have parallel inputs and parallel outputs, but the internal structure is different. One is implemented with the full adders in a serial fashion (Figure 20). This means that each full adder forwards the carry bit to the next clock cycle and the carry out is connected to the carry in of the same full adder.

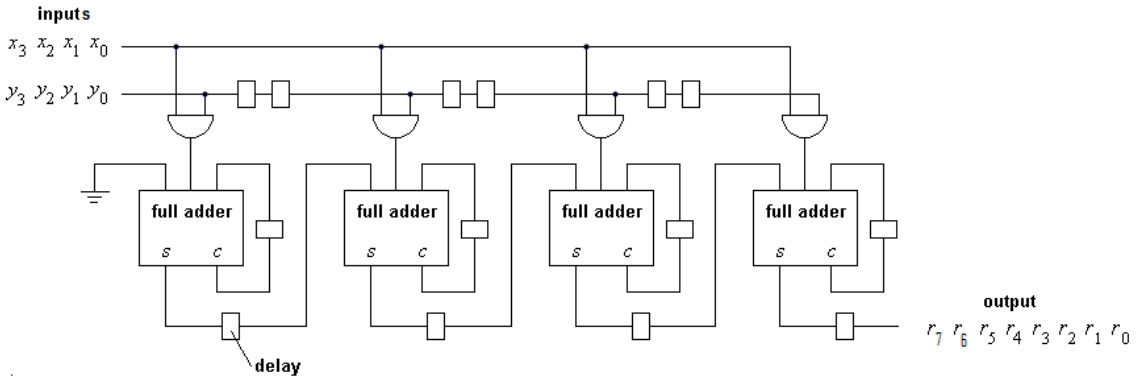


Figure 20: Serial multiplier with the carry-out connected to the carry-in on the same full adder

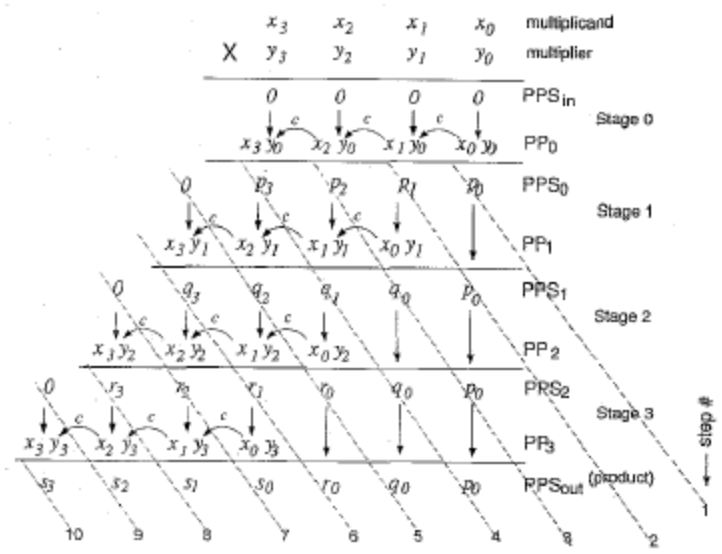


Figure 21: Data flow for serial-implemented multiplier (after [15])

Figure 20 and 21 shows the data flow for a 4x4 multiplier. Each step is represented by a dotted diagonal line and lasts one clock cycle. On the first step, the first result is calculated by sending the two least-significant bits (LSB) of the multiplier and the multiplicand through an AND gate, and the result is added together with the carry-out from the first full adder, which is initialized to zero. The carry-out bit of the first full adder is carried over to the next clock cycle, which is the carry-in bit on the first full adder. The sum bit is carried over to the next clock cycle, which is the input of the second full adder (Figure 22).

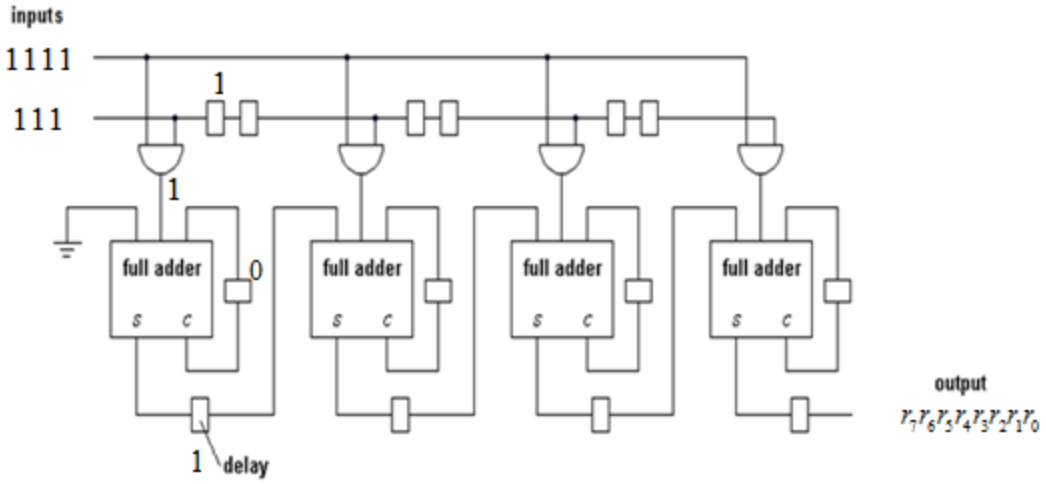


Figure 22: First clock cycle for the serial multiplier with 4-bit inputs of all 1

The second full adder adds the input (from the first full adder) of the second full adder to the input from the AND gate (which is zero) and the carry-in (which is zero). The results of the AND gate and the carry-in are zero because the nodes in the multiplier were initialized to zero. The LSB of the multiplicand is shifted through the delay which is a D-type flip-flop (DFF). The LSB of the multiplier and the second bit of the multiplicand is sent through the AND gate and added together with the carry-in bit of the first full adder (Figure 23).

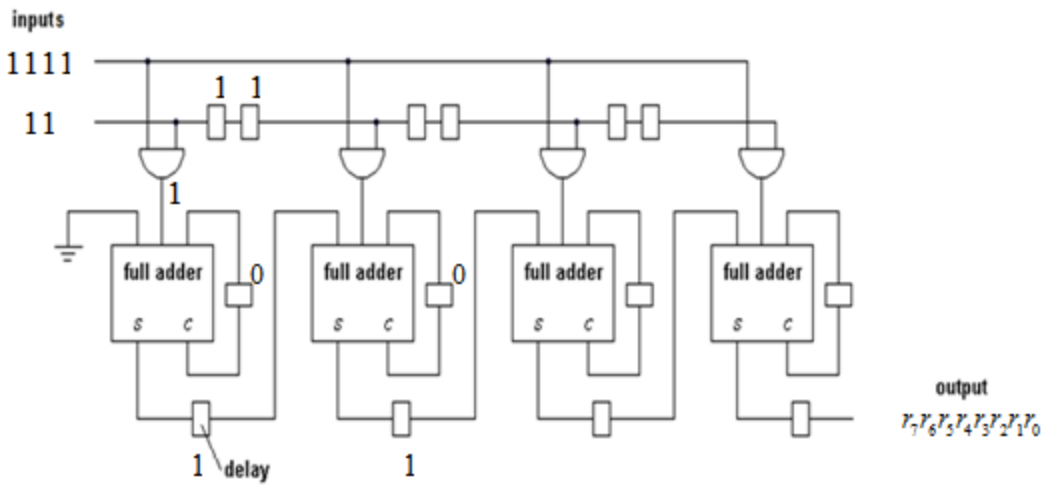


Figure 23: Second clock cycle of the serial multiplier

The result from the second full adder is sent to the input of the third full adder, and the carry-out of the second full adder is the carry-in of the second full adder. The LSB of the multiplicand is sent through the next delay and the AND gate of the second full adder with the second bit of the multiplier. The output of the AND gate is added together with the carry-in of the second full adder and the result of the first full adder. The LSB of the multiplier is sent through the AND gate with the third bit of the multiplicand. The result of this AND gate is added together with the carry-in of the first full adder and the ground input (Figure 24).

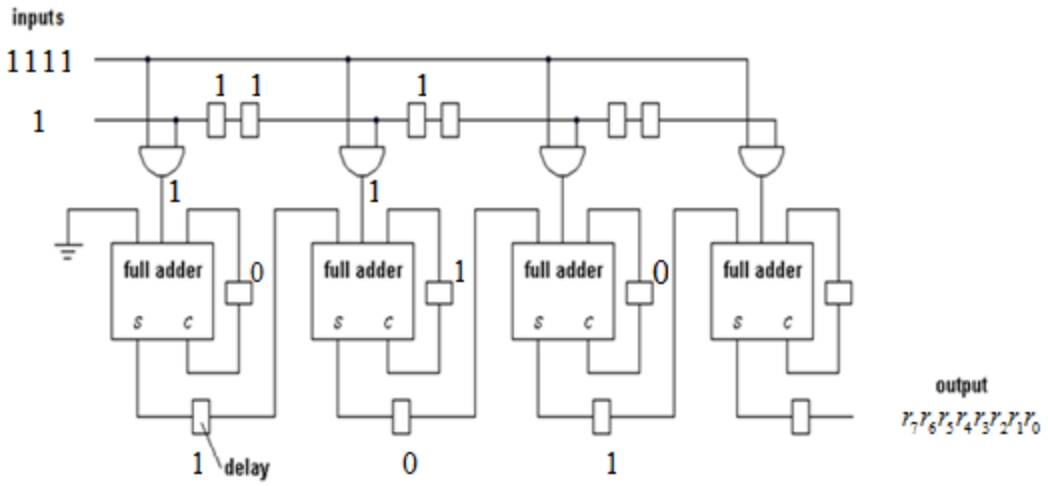


Figure 24: Third clock cycle of the serial multiplier

The fourth step takes the result from the third full adder and adds it together with the carry-in of the fourth full adder and the output of the fourth AND gate. The output of the fourth full adder is the answer. The first result is the LSB of the result. The result of the second full adder is added together with the result of the third AND gate (which is zero) and the carry-in of the third full adder. The result of the first full adder is added together with the result of the AND gate (which is the second bit of the multiplicand and the second bit of the multiplier) and the carry-in of the second full adder. The fourth bit of the multiplicand and the LSB of the multiplier is sent through the AND gate and added to the carry-in of the first full adder and the input ground (Figure 25).

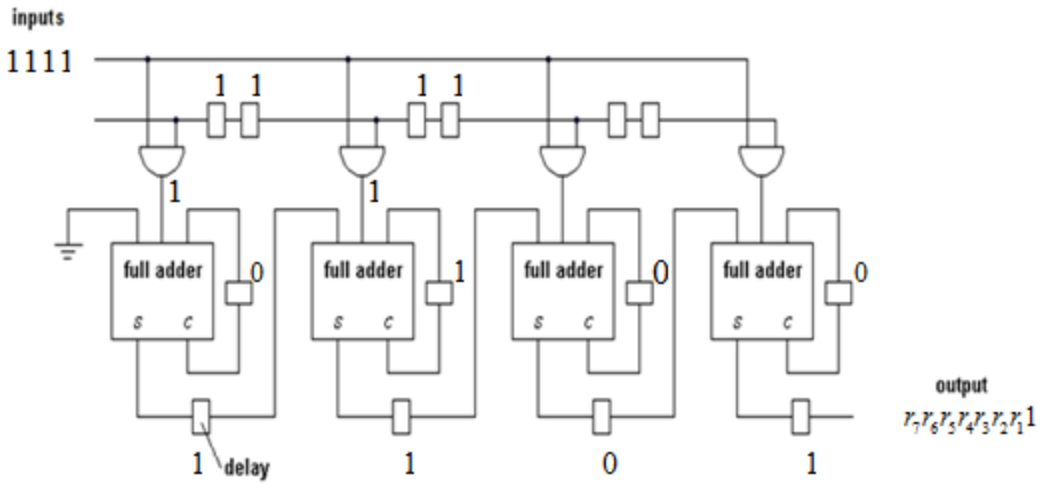


Figure 25: Fourth clock cycle for the serial multiplier

The process continues until all of the bits in the result have been calculated. For a 4x4 multiplier this will take 11 clock cycles to complete [15]. For the serial multiplier, everything must be initialized to zero so there are no unknown nodes. This is because the carry-out for the adders in the serial multiplier is tied to the carry-in of the same adder (Figure 20). If the nodes are not initialized to zero before the inputs are fed through, then it could give incorrect results. This can be done by setting the inputs to zero until they have cycled through the whole multiplier or by clearing all of the DFFs in the multiplier and letting it cycle through the multiplier. Since the serial multiplier takes 11 cycles to complete, to perform a multiplication every cycle requires 11 multipliers in parallel. Each set of inputs is given to a different multiplier and then after it cycles through it comes back to the first multiplier and starts over. This architecture can confine the errors caused by the SET to be restricted to one clock cycle of the results unless the SET occurs at the end of one calculation and extends to the next. The outputs from the multipliers are

selected using multiplexers (Figure 26). The formula for finding out the number of clock cycles for a specific input is:

$$cc = 3b - 1, \tag{4}$$

where *cc* is the number of clock cycles and *b* is the number of bits in the inputs. So, for 16-bit inputs, the number of clock cycles delayed before all of the bits in the result are calculated is 47.

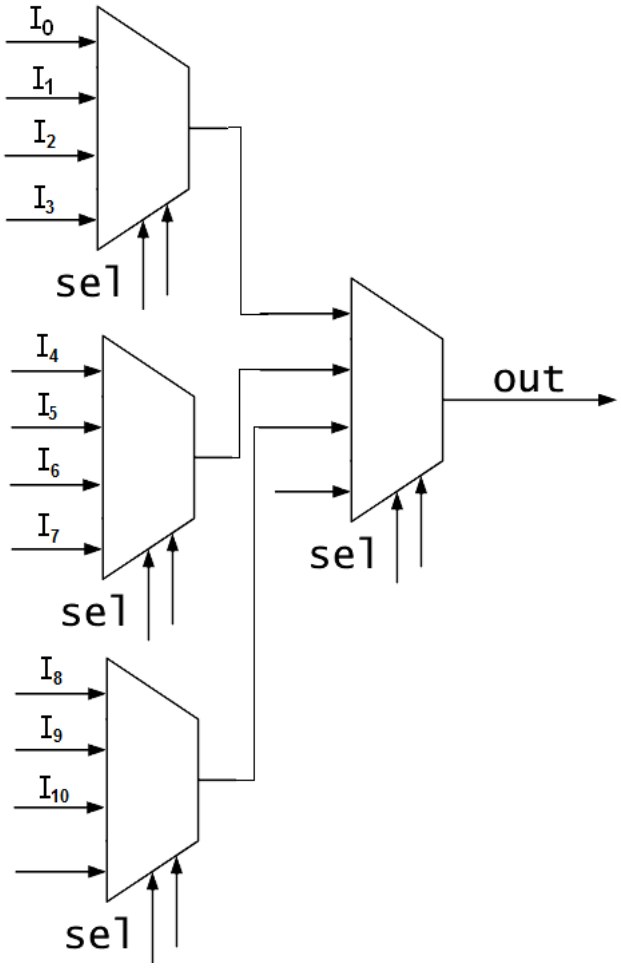


Figure 26: Output selection for serial multiplier

In the other multiplier design, the full adders are implemented in a parallel fashion (Figure 27 and 28). This means that for each clock cycle the full adders have to propagate the carries through all of the full adders thus slowing down the circuit. In this design each partial product is a clock cycle. For 4-bit inputs, there will be a four clock cycle delay between the inputs and the result. The first clock cycle will calculate the first partial product by sending the LSB of the multiplier and each bit in the multiplicand through an AND gate. The result of the LSB of the multiplier and the multiplicand is the first bit in the result. All of the results are then passed to the next clock cycle. The second clock cycle calculates the second partial product by sending the second bit in the multiplier and each bit in the multiplicand through an AND gate. Then it adds this result to the first partial product. Each addition has to wait for the previous one to finish so that it can add the carry out from that result. Each of the results is passed to the next clock cycle. This is what makes this process slower than the serially-implemented full adder multiplier. Figure 28 shows the schematic for the first two clock cycles. The third and fourth clock cycles are the same as the second.

TABLE 4-3: **Four-bit Multiplier Partial Products**

				X_3	X_2	X_1	X_0	Multiplicand
				Y_3	Y_2	Y_1	Y_0	Multiplier
				X_3Y_0	X_2Y_0	X_1Y_0	X_0Y_0	Partial product 0
			X_3Y_1	X_2Y_1	X_1Y_1	X_0Y_1		Partial product 1
			C_{12}	C_{11}	C_{10}			First row carries
		C_{13}	S_{13}	S_{12}	S_{11}	S_{10}		First row sums
		X_3Y_2	X_2Y_2	X_1Y_2	X_0Y_2			Partial product 2
		C_{22}	C_{21}	C_{20}				Second row carries
	C_{23}	S_{23}	S_{22}	S_{21}	S_{20}			Second row sums
	X_3Y_3	X_2Y_3	X_1Y_3	X_0Y_3				Partial product 3
	C_{32}	C_{31}	C_{30}					Third row carries
C_{33}	S_{33}	S_{32}	S_{31}	S_{30}				Third row sums
P_7	P_6	P_5	P_4	P_3	P_2	P_1	P_0	Final product

Figure 27: Parallel-implemented multiplier (after [16])

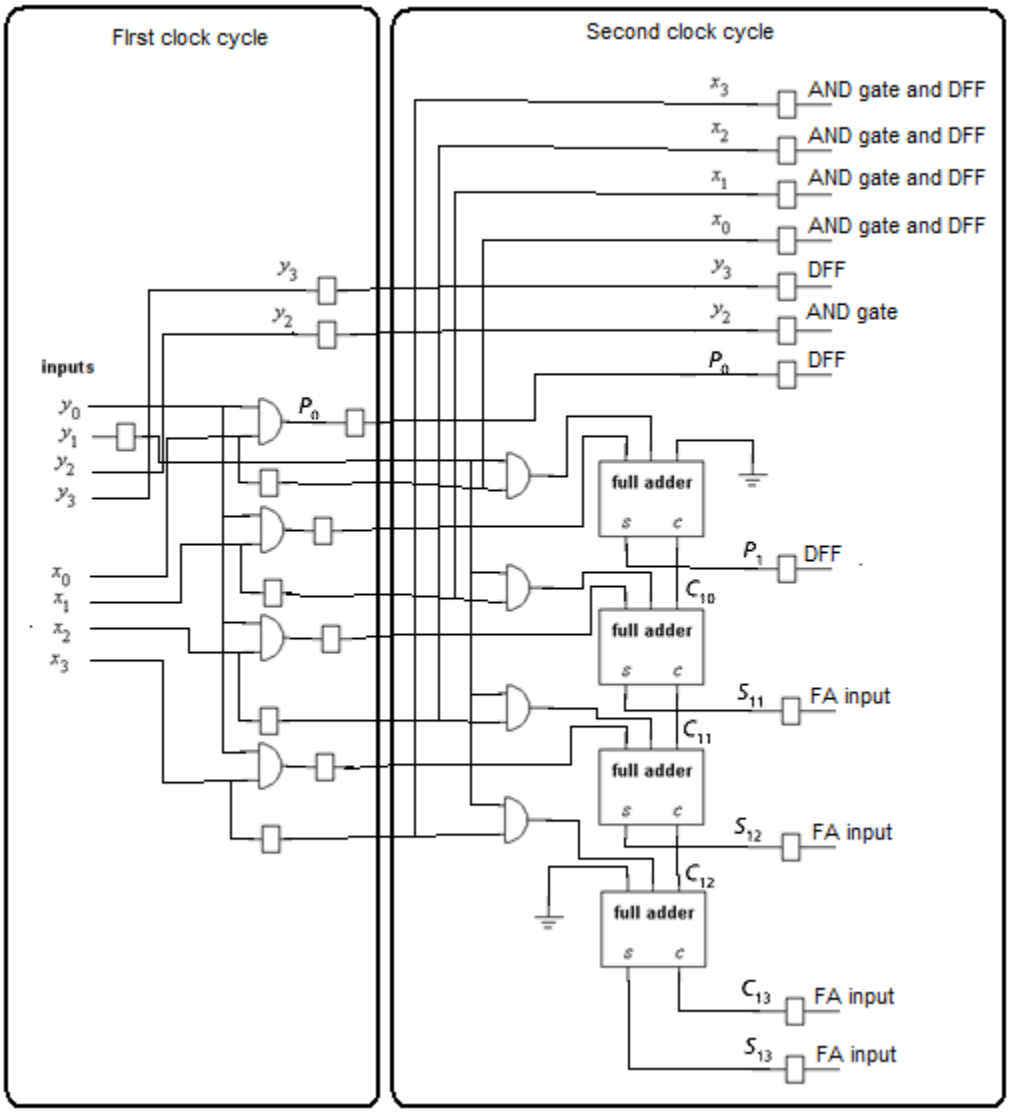


Figure 28: Schematic for the first two clock cycles of the 4x4 parallel-implemented multiplier

Each multiplier will have 16x16 bit inputs resulting in a 32-bit output. To simulate the mixer on a software-defined radio, one input will be sinusoidal, and the other will be constant. There will be two parallel-implemented multipliers and two serial-implemented multipliers. One of each will be a golden multiplier, and the other will be

the one which an SET is injected on a given node. The results from the SET multiplier and the golden multiplier will be compared to find out which output bits in the SET multiplier were errors.

3.2 Parallel-Implemented Full Adder Multiplier

The golden parallel implemented multiplier has two input signals, **A** and **B**, and an output signal, ***y***. The full adders are implemented in a parallel fashion. A normal working 4x4 parallel-implemented multiplier is shown below (Figure 29).

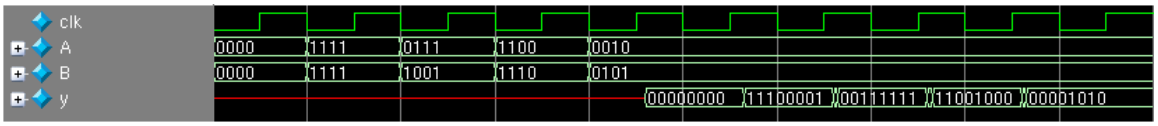


Figure 29: Golden parallel-implemented full adder multiplier

There is a four-cycle delay between the inputs and the output. The red line in the signal ***y*** is an unknown state, where the output has not been calculated. From Table 3 it looks like there is a five cycle delay, but this is because the DFFs will output the results on a positive edge clock and the passing of values at each clock cycle is done on the negative edge of the clock. The input signals, **A** and **B**, change with each clock cycle until it reaches the end and then it stays at that value for the remaining clock cycles. After the first output signal, ***y***, has been calculated, the output changes every clock cycle.

Table 3: Results for golden parallel-implemented full adder multiplier

CC	A	B	y
1	0	0	x
2	15	15	x
3	7	9	x
4	12	14	x
5	2	5	x
6	2	5	0
7	2	5	225
8	2	5	63
9	2	5	168
10	2	5	10

The parallel-implemented full adder is now set up to apply an SET to a node and see how it affects the output. There are two additional inputs to the multiplier. There is an input signal of **SETL**, which will drive a node to the value of 0 for the length of time that it is active. There is an input signal of **SETH**, which will drive a node to the value of 1 for the length of time that it is active. The SET works by applying the SET to the node on the clock cycle that it is initiated. The multipliers are set up so that the inputs will be shifted through the multipliers in clock cycles. This means that if an SET is applied to the last node in the multiplier on the tenth clock cycle, then the SET will not affect the input that is loaded on the tenth clock cycle, but it will affect a previous input. Figure 30 shows the result of Node 1 being driven low for a length of 3 clock cycles.

Figure 30: Parallel-implemented full adder multiplier with SETL

The same inputs are used as in the golden simulation so that it will be easy to see if there is a difference in the output (Table 4). Node 1 is the output of the first AND gate in the first clock cycle. Since Node 1 is in the first clock cycle, the SET is applied to the input that is loaded during the same clock cycle. This calculates the first bit in the output. This means that the least-significant bit in the output will be the bit that can be affected by Node 1 being held low. The second and third results are different from the results of the golden multiplier. The multiplication of 15 and 15 in the golden multiplier gave a result of 225, but when Node 1 is held low, the result is 224, which means that the LSB in the output is 0 instead of a 1. The same is true for the third result. The multiplication of 7 and 9 in the golden multiplier gave a result of 63, but when Node 1 is held low, the result is 62, because the LSB in the output is 0 instead of 1. Node 1 is also held low for the fourth clock cycle but it does not change the result of 168 because the value at node 1 is already 0.

Table 4: Results for parallel-implemented full adder with SETL

CC	A	B	SETH	SETL	p
1	0	0	0	0	x
2	15	15	0	1	x
3	7	9	0	1	x
4	12	14	0	1	x
5	2	5	0	0	x
6	2	5	0	0	0
7	2	5	0	0	224
8	2	5	0	0	62
9	2	5	0	0	168
10	2	5	0	0	10

The input signal **SETH** is now active for Node 1 (Figure 31). This will result in holding Node 1 high. The input signal **SETH** is held high from the third clock cycle on.

The inputs are the same as in the golden parallel-implemented full adder multiplier so that it will be easy to tell if there is a difference in the output.

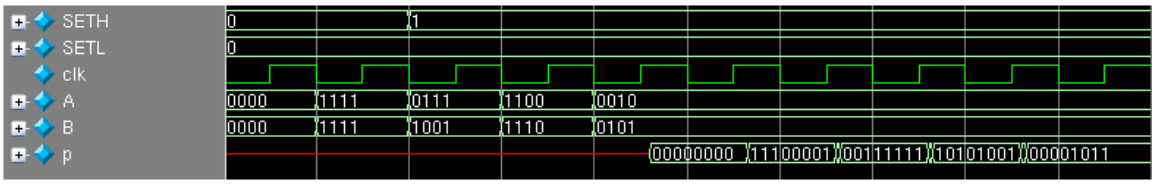


Figure 31: Parallel-implemented full adder multiplier with SETH

The effect of Node 1 being held high changes the fourth and fifth output (Table 5). The multiplication of 12 and 14 from the golden multiplier gave a result of 168. When Node 1 is held high, it gives a result of 169, because the LSB in the output is a 1 now instead of a 0. The multiplication of 2 and 5 from the golden multiplier gave a result of 10 but now gives a result of 11 when Node 1 is held high.

Table 5: Results for parallel-implemented full adder with SETH

CC	A	B	SETH	SETL	p
1	0	0	0	0	x
2	15	15	0	0	x
3	7	9	1	0	x
4	12	14	1	0	x
5	2	5	1	0	x
6	2	5	1	0	0
7	2	5	1	0	225
8	2	5	1	0	63
9	2	5	1	0	169
10	2	5	1	0	11

The results from the SET parallel-implemented full adder multiplier and the golden parallel-implemented full adder multiplier are now compared. Figure 32 shows the results of the multipliers and the number of errors in each output. Both multipliers have the same input signals, **A** and **B**, but different outputs. The output signal for the golden parallel-implemented full adder multiplier is ***y*** and the output signal for the SET parallel-implemented full adder multiplier is ***p***. When the input signals **SETH** or **SETL** is driven high, then the output for the SET parallel-implemented full adder multiplier can be different from the output for the golden parallel-implemented full adder multiplier. An SET does not always affect the output of the multiplier because the value for the node that the SET is applied to could already be in the logic state. A bit-by-bit comparison is made between the results from the SET parallel-implemented full adder multiplier and the golden parallel-implemented full adder multiplier. There is a two-cycle delay between the outputs and the comparison because the outputs are compared on the next positive edge of the clock cycle, and the results of the comparison are run through DFFs which delays them until the next positive clock edge. The **count1** signal adds up the number of errors or number of 1's in the **compare** signal. The **count1** signal is delayed a clock cycle from the **compare** signal because the number of errors is added together on a positive edge of the clock cycle, and then the DFF's send the result on the next positive-edge clock cycle. This time Node 5 is held low. Node 5 is the first AND gate in the second stage. Since the node is in the second stage, the SET will affect the input that is loaded on the previous clock cycle because it takes one clock cycle for the inputs to get to Node 5.

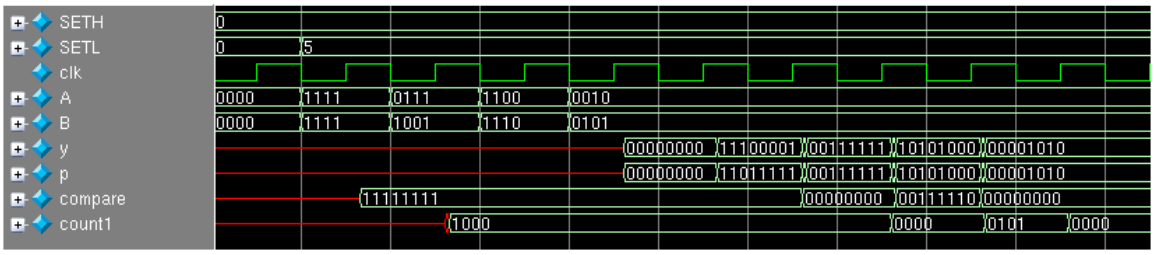


Figure 32: Comparing two parallel-implemented full adder multipliers

The result of the Node 5 being held high, on the second clock cycle, is applied to all of the inputs including the initial inputs, but it only causes a change in the second input (Table 6). Node 5 does not just affect one output like Node 1 but can affect the result of multiple output bits. In this case it changes five bits in the second output. However, holding Node 5 low does not affect the other outputs because the result of the first AND gate in the second clock cycle is already at 0 for the other cases. The first AND gate in the second clock cycles has the inputs of the LSB in input signal **A** and the second bit in input signal **B**, which is 0 for every case but the second set of inputs.

Table 6: Results for comparing two parallel-implemented full adder multipliers

CC	A	B	SETH	SETL	y	p	compare	count1
1	0	0	0	0	x	x	x	x
2	15	15	0	5	x	x	x	x
3	7	9	0	5	x	x	255	x
4	12	14	0	5	x	x	255	8
5	2	5	0	5	x	x	255	8
6	2	5	0	5	0	0	255	8
7	2	5	0	5	225	223	255	8
8	2	5	0	5	63	63	0	8
9	2	5	0	5	168	168	62	0
10	2	5	0	5	10	10	0	5
11	2	5	0	5	10	10	0	0
12	2	5	0	5	10	10	0	0
13	2	5	0	5	10	10	0	0

3.3 Serial-Implemented Full Adder Multiplier

The golden serial-implemented multiplier has two input signals, **A** and **B**, and an output signal, ***y***. The full adders are implemented in a serial fashion. A normal working 4x4 serial implemented multiplier is shown below with the outputs shown in the simulation block for the first three outputs (Figure 33).

Figure 33: Golden serial-implemented full adder multiplier with outputs

There is a longer clock cycle delay between the inputs and the outputs when compared to the parallel-implemented multiplier. The inputs for the serial-implemented multiplier go through two inverters before going to the multipliers. This is to show that an SET applied to the inverters will generate errors over multiple results, while an SET applied inside the multipliers will generate errors that happen within the same result. The **clear** signal of the DFF is activated before the inputs are sent through so that everything

is initialized to zero. The input signal **A** for the serial-implemented multiplier is held constant at one while the input signal **B** starts at one and is increased by one every clock cycle. Therefore, the outputs also start at one and then increase by one every clock cycle.

Table 7: Results for golden serial-implemented full adder multiplier

CC	A	B	y
1	0	0	x
2	0	0	x
3	0	0	x
4	0	0	x
5	0	0	x
6	0	0	x
7	0	0	x
8	0	0	x
9	0	0	x
10	0	0	x
11	0	0	x
12	0	0	x
13	0	0	x
14	1	1	x
15	1	2	x
16	1	3	X
17	1	4	X
18	1	5	X
19	1	6	X
20	1	7	0
21	1	8	0
22	1	9	0
23	1	10	0
24	1	11	0
25	1	11	0
26	1	11	0
27	1	11	0
28	1	11	0
29	1	11	0
30	1	11	1
31	1	11	2
32	1	11	3
33	1	11	4
34	1	11	5
35	1	11	6
36	1	11	7
37	1	11	8
38	1	11	9
39	1	11	10
40	1	11	11

There is a different result when the SET is applied to the inverters and when it is applied inside the multipliers. When the SET is applied to the inverters, the SET causes errors that affect the result over multiple clock cycles. When the SET is applied inside the multipliers, the SET will only affect one clock cycle on the output. Node 10 in the inverter chain is held low (Figure 34).

Figure 34: Serial-implemented full adder multiplier with SETL

Node 10 is the second inverter for the first bit, in input signal **B** and can affect the LSB in the input because it goes into the multipliers. The odd-numbered inputs are affected by the input signal **SETL** (Table 8). The even-numbered inputs are not affected by the input signal **SETL** because the LSB is already 0.

Table 8: Results for serial-implemented full adder multiplier with SETL

CC	A	B	p	SETH	SETL
1	0	0	x	0	10
2	0	0	x	0	10
3	0	0	x	0	10
4	0	0	x	0	10
5	0	0	x	0	10
6	0	0	x	0	10
7	0	0	x	0	10
8	0	0	x	0	10
9	0	0	x	0	10
10	0	0	x	0	10
11	0	0	x	0	10
12	0	0	x	0	10
13	0	0	x	0	10
14	1	1	x	0	10
15	1	2	x	0	10
16	1	3	X	0	10
17	1	4	X	0	10
18	1	5	X	0	10
19	1	6	X	0	10
20	1	7	0	0	10
21	1	8	0	0	10
22	1	9	0	0	10
23	1	10	0	0	10
24	1	11	0	0	10
25	1	11	0	0	10
26	1	11	0	0	10
27	1	11	0	0	10
28	1	11	0	0	10
29	1	11	0	0	10
30	1	11	0	0	10
31	1	11	2	0	10
32	1	11	2	0	10
33	1	11	4	0	10
34	1	11	4	0	10
35	1	11	6	0	10
36	1	11	6	0	10
37	1	11	8	0	10
38	1	11	8	0	10
39	1	11	10	0	10
40	1	11	10	0	10
41	1	11	10	0	10
42	1	11	10	0	10
43	1	11	10	0	10
44	1	11	10	0	10

Now the input signal **SETH** is active, and Node 10 is held high (Figure 35) to observe the effect on the output. The same input values are used for **A** and **B** as in the golden serial-implemented multiplier so that is easy to see the difference in the results.

Figure 35: Serial-implement full adder multiplier SETH

The even outputs are affected by the input signal **SETH** because Node 10 is the inverter for the LSB in the input signal **B** (Table 9). This is because Node 10 affects the LSB in the output.

Table 9: Results for serial-implemented full adder multiplier with SETH

CC	A	B	p	SETH	SETL
1	0	0	x	10	0
2	0	0	x	10	0
3	0	0	x	10	0
4	0	0	x	10	0
5	0	0	x	10	0
6	0	0	x	10	0
7	0	0	x	10	0
8	0	0	x	10	0
9	0	0	x	10	0
10	0	0	x	10	0
11	0	0	x	10	0
12	0	0	x	10	0
13	0	0	x	10	0
14	1	1	x	10	0
15	1	2	x	10	0
16	1	3	X	10	0
17	1	4	X	10	0
18	1	5	X	10	0
19	1	6	X	10	0
20	1	7	0	10	0
21	1	8	0	10	0
22	1	9	0	10	0
23	1	10	0	10	0
24	1	11	0	10	0
25	1	11	0	10	0
26	1	11	0	10	0
27	1	11	0	10	0
28	1	11	0	10	0
29	1	11	0	10	0
30	1	11	1	10	0
31	1	11	3	10	0
32	1	11	3	10	0
33	1	11	5	10	0
34	1	11	5	10	0
35	1	11	7	10	0
36	1	11	7	10	0
37	1	11	9	10	0
38	1	11	9	10	0
39	1	11	11	10	0
40	1	11	11	10	0
41	1	11	11	10	0
42	1	11	11	10	0
43	1	11	11	10	0
44	1	11	11	10	0

The input signal **SETL** is then applied to the second multiplier. The SET is held low and applied to Node 10 on the second multiplier (Figure 36). Node 10 is the output on the third full adder.

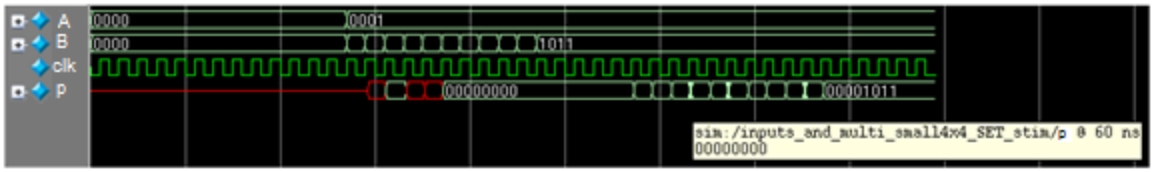


Figure 36: Serial-implemented full adder multiplier SETL2

This time it only affects the result for the output on the second multiplier. The outputs for the other multipliers are not affected by the input signal **SETL2**. The inputs for the second multiplier are 1 and 2, and the output is 0 (Table 10).

Table 10: Results for serial-implemented full adder multiplier with SETL2

CC	A	B	p	SETH2	SETL2
1	0	0	x	0	10
2	0	0	x	0	10
3	0	0	x	0	10
4	0	0	x	0	10
5	0	0	x	0	10
6	0	0	x	0	10
7	0	0	x	0	10
8	0	0	x	0	10
9	0	0	x	0	10
10	0	0	x	0	10
11	0	0	x	0	10
12	0	0	x	0	10
13	0	0	x	0	10
14	1	1	x	0	10
15	1	2	x	0	10
16	1	3	X	0	10
17	1	4	0	0	10
18	1	5	X	0	10
19	1	6	X	0	10
20	1	7	0	0	10
21	1	8	0	0	10
22	1	9	0	0	10
23	1	10	0	0	10
24	1	11	0	0	10
25	1	11	0	0	10
26	1	11	0	0	10
27	1	11	0	0	10
28	1	11	0	0	10
29	1	11	0	0	10
30	1	11	1	0	10
31	1	11	0	0	10
32	1	11	3	0	10
33	1	11	4	0	10
34	1	11	5	0	10
35	1	11	6	0	10
36	1	11	7	0	10
37	1	11	8	0	10
38	1	11	9	0	10
39	1	11	10	0	10
40	1	11	11	0	10
41	1	11	11	0	10
42	1	11	11	0	10
43	1	11	11	0	10
44	1	11	11	0	10

The input signal **SETH** is then applied to the second multiplier. The SET is applied to Node 10 and held high on the second multiplier (Figure 37). Node 10 is the output on the third full adder.

Figure 37: Serial-implemented full adder multiplier SETH2

When the input signal **SETH2** is set for Node 10, it holds Node 10 high. It only affects the result for the output on the second multiplier. The inputs for the second multiplier are 1 and 2, and the output is 255 (Table 11).

Table 11: Results for serial-implemented full adder multiplier with SETH2

CC	A	B	p	SETH2	SETL2
1	0	0	x	0	0
2	0	0	x	0	0
3	0	0	x	0	0
4	0	0	x	0	0
5	0	0	x	0	0
6	0	0	x	0	0
7	0	0	x	0	0
8	0	0	x	0	0
9	0	0	x	0	0
10	0	0	x	0	0
11	0	0	x	0	0
12	0	0	x	0	0
13	0	0	x	0	0
14	1	1	x	10	0
15	1	2	x	10	0
16	1	3	X	10	0
17	1	4	X	10	0
18	1	5	X	10	0
19	1	6	X	10	0
20	1	7	0	10	0
21	1	8	0	10	0
22	1	9	0	10	0
23	1	10	0	10	0
24	1	11	0	10	0
25	1	11	0	10	0
26	1	11	0	10	0
27	1	11	0	10	0
28	1	11	0	10	0
29	1	11	0	10	0
30	1	11	1	10	0
31	1	11	255	10	0
32	1	11	3	10	0
33	1	11	4	10	0
34	1	11	5	10	0
35	1	11	6	10	0
36	1	11	7	10	0
37	1	11	8	10	0
38	1	11	9	10	0
39	1	11	10	10	0
40	1	11	11	10	0
41	1	11	11	10	0
42	1	11	11	10	0
43	1	11	11	10	0
44	1	11	11	10	0

The results from the golden serial-implemented full adder multiplier and the SET serial-implemented full adder multiplier are compared (Figure 38). The comparator operates the same as in the parallel-implemented full adder multiplier. The output signal **count11** counts the number of bits that are different between the golden serial-

implemented full adder multiplier and the SET serial-implemented full adder multiplier. Node 10 is held low from the first multiplier. Node 10 is the output of the first full adder. This node only affects the output on the first multiplier.

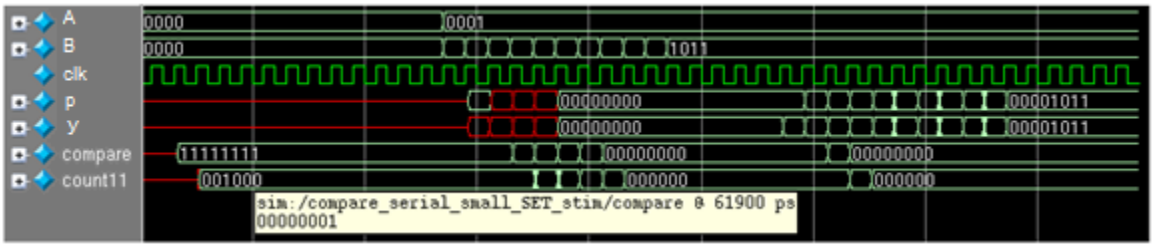


Figure 38: Comparing two serial-implemented full adder multipliers

The input signal **SETL** affects Node 10, and there is a difference in the first output signal between the output signal of ***y***, the golden serial-implemented full adder multiplier, and the output signal ***p***, the SET serial-implemented full adder multiplier (Table 12). The output signal for ***y*** is 1, and the output signal for ***p*** is 0. The output signal **compare** is shown two clock cycles later, and it is 1 or in binary 00000001.

Table 12: Results for comparing two serial-implemented full adder multipliers

cc	A	B	p	y	compare	count11
1	0	0	x	x	x	x
2	0	0	x	x	x	x
3	0	0	x	x	255	x
4	0	0	x	x	255	8
5	0	0	x	x	255	8
6	0	0	x	x	255	8
7	0	0	x	x	255	8
8	0	0	x	x	255	8
9	0	0	x	x	255	8
10	0	0	x	x	255	8
11	0	0	x	x	255	8
12	0	0	x	x	255	8
13	0	0	x	x	255	8
14	1	1	x	x	255	8
15	1	2	x	x	255	8
16	1	3	0	X	255	8
17	1	4	X	X	255	8
18	1	5	X	X	15	8
19	1	6	X	X	7	4
20	1	7	0	0	3	3
21	1	8	0	0	1	2
22	1	9	0	0	0	1
23	1	10	0	0	0	0
24	1	11	0	0	0	0
25	1	11	0	0	0	0
26	1	11	0	0	0	0
27	1	11	0	0	0	0
28	1	11	0	0	0	0
29	1	11	0	0	0	0
30	1	11	0	1	0	0
31	1	11	2	2	0	0
32	1	11	3	3	1	0
33	1	11	4	4	0	1
34	1	11	5	5	0	0
35	1	11	6	6	0	0
36	1	11	7	7	0	0
37	1	11	8	8	0	0
38	1	11	9	9	0	0
39	1	11	10	10	0	0
40	1	11	11	11	0	0
41	1	11	11	11	0	0
42	1	11	11	11	0	0
43	1	11	11	11	0	0
44	1	11	11	11	0	0

If an SET lasts long enough, it can affect multiple outputs. To find the shortest SET length that can affect multiple outputs, the clock cycles in which each node affects the output for two consecutive inputs are shown in Figure 39. For the 4x4 serial-implemented full adder multiplier, the shortest SET that can have an effect on multiple outputs is one that last four clock cycles. This is found by finding the least number of

clock cycles for a node that will affect the calculation of the first input values and the calculation of the second input values. The **CC** column is the clock cycles. **And1** is the output node for the first AND gate. **FAS1** is the sum node for the first full adder. **FAC1** is the carry node for the first full adder. **DS1** is the DFF for the sum of the first full adder. **DC1** is the DFF for the carry of the first full adder. **D11** is the first DFF for the input that is being shifted from the first AND gate to the second AND gate. **D12** is the second DFF for the input that is being shifted from the first AND gate to the second AND gate.

CC	And1	And2	And3	And4	FAS1	FAC1	FAS2	FAC2	FAS3	FAC3	FAS4	FAC4	DS1	DC1	DS2	DC2	DS3	DC3	DS4	DC4	D11	D12	D21	D22	D31	D32
1	x				x	x							x	x							x					
2	x	x			x	x	x	x					x	x	x	x					x	x				
3	x	x	x		x	x	x	x	x	x			x	x	x	x	x	x			x	x	x			
4	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x		
5	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	
6	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x		x	x	x	x	x
7	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x			x	x	x	x
8		x	x	x	x		x	x	x	x	x	x	x		x	x	x	x	x	x				x	x	x
9			x	x			x		x	x	x	x			x		x	x	x	x					x	x
10				x					x		x	x					x		x	x						x
11				x							x								x							
12	x				x	x							x	x							x					
13	x	x			x	x	x	x					x	x	x	x					x	x				
14	x	x	x		x	x	x	x	x	x			x	x	x	x	x	x			x	x	x			
15	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x		
16	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	
17	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x		x	x	x	x	x
18	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x			x	x	x	x
19		x	x	x	x		x	x	x	x	x	x	x		x	x	x	x	x	x				x	x	x
20			x	x			x		x	x	x	x			x		x	x	x	x					x	x
21				x					x		x	x					x		x	x						x
22				x							x								x							

Figure 39: Nodes and the clock cycles that they affect the output

CHAPTER IV

RESULTS AND SIGNAL ANAYLSIS FOR 16x16 MULTIPLIERS

4.1 Results for 16x16 Multipliers

For the simulations, 16x16 input multipliers are used. They function the same way as the 4x4 multipliers. One of the inputs is a sinusoidal wave with 32 values. These values are retrieved from a table. The second input is a constant, 43,690. There are a total of 128 inputs for each simulation. This means that the sinusoidal wave will make four complete cycles. The phase that the sinusoidal wave starts at changes each time a simulation is run. The SETs are configured to last for three different duration lengths (1, 6, or 36 cycles). The SETs are also set up to either hold the node low or high for the duration. The simulation selects a node and applies the SET to that node. The simulations run for all of the possible combination of nodes, durations, phases and the value of the SET (low or high). The simulations print the results for each run. In Appendix L contains an example for the parallel multiplier.

The parallel-implemented multiplier is made up of a total of 1,279 logic gates. It has 256 AND gates, 240 full adders, and 783 DFFs. The speed of the parallel-implemented multiplier is set by the full adders and the carry chain (Figure 40). The serial-implemented multiplier is made up of a total of 4,465 logic gates. It has 752 AND

gates, 752 full adders, and 2,961 DFFs. The speed of the serial-implemented multiplier is set by the full adder and the latency of the input to the output (Figure 41).

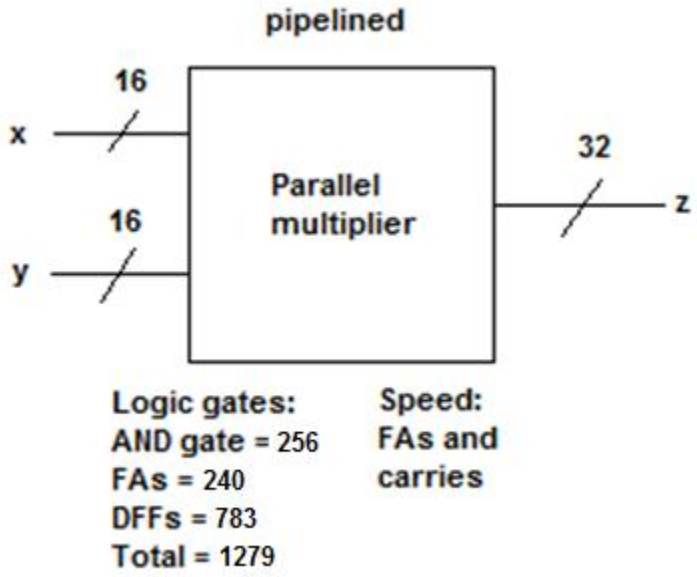


Figure 40: 16x16 parallel-implemented multiplier

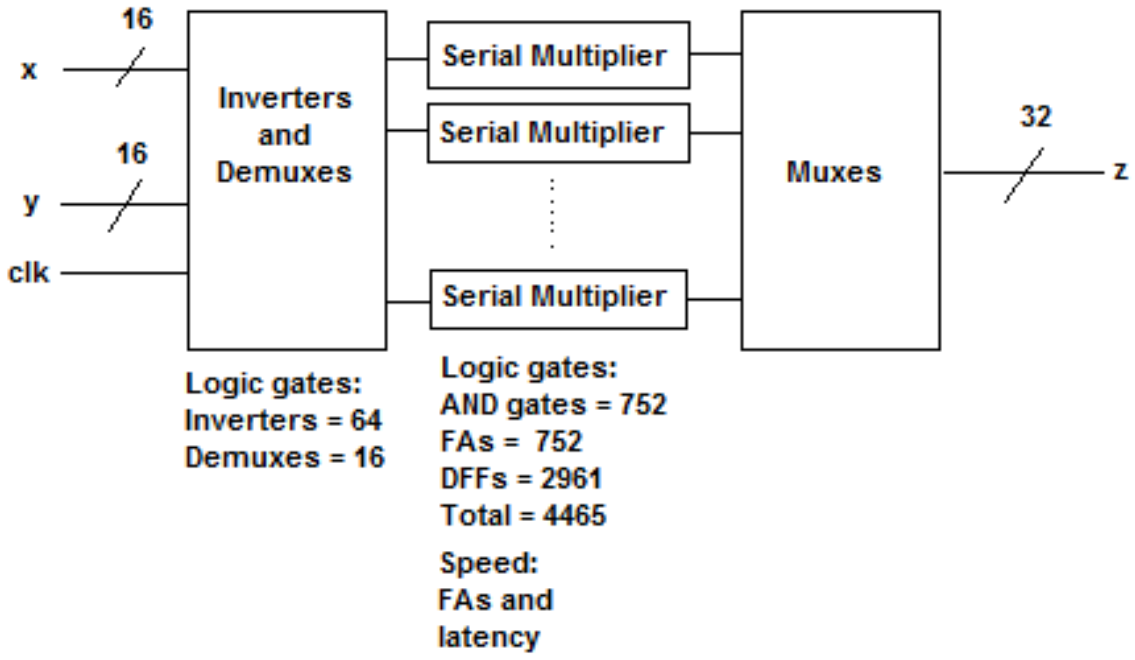


Figure 41: 16x16 serial-implemented multiplier

The results of the simulations for the parallel-implemented multiplier are organized by duration and SETH/SETL. For each duration and SETH/SETL, the SET was applied to all of the nodes (736) and every phase (0 to 31) for each node. As the SET length increases, the total number of errors increases. However, the number of clock cycles in which those errors occur also increases at the same rate. Therefore, the number of errors per clock cycle does not change. The maximum number of errors for one SET is the maximum number of errors for all of the combinations of nodes and phases for the given duration and SETH/SETL value. The maximum number of errors for one SET and the maximum number of clock cycles that the SET affects also increases. This means that as the length of the SET increases, the errors are spread out over a greater number of clock cycles.

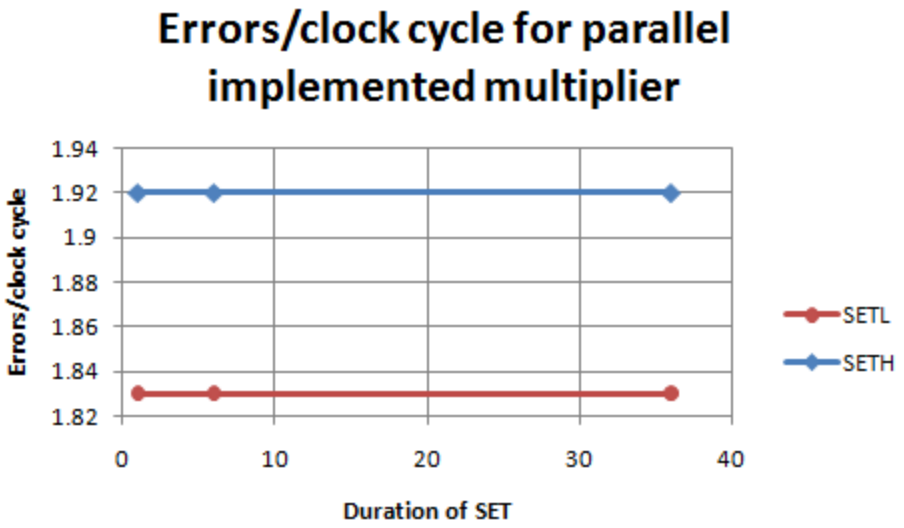


Figure 42: Errors per clock cycle in parallel-implemented full adder multiplier

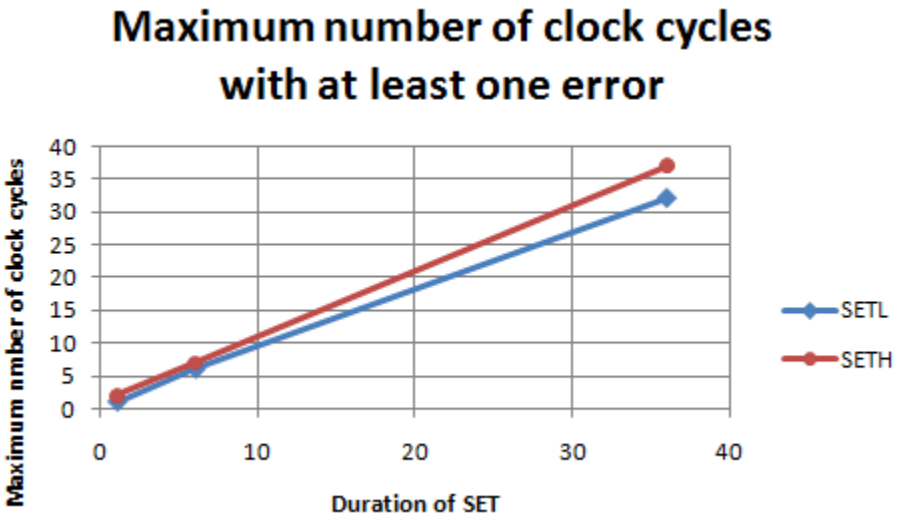


Figure 43: Maximum number of clock cycles with at least one error in parallel-implemented full adder multiplier

The results for the serial-implemented multiplier have been split up into two parts. The first set of simulations is for the SET being applied to the inverters. The first set of

simulations is indicated by a select value of 1. The second set of simulations is for the SET being applied one of the 47 copies of the serial multiplier. The second set of simulations is indicated by a select value of 2 through 48. For a select value of 2, the SET is applied to the serial multiplier that first receives input values. For a select value of 48, the SET is applied to the serial multiplier that is last in receiving input values. The results for each have also been divided into duration and SETH/SETL values. For the inverters within each duration and SETH/SETL value, the SET was applied to all of the nodes for the inverters (64) and all of the phases for each node. The results are the same for SETH/SETL being 0 or 1. This is because each of the input bits passes through two inverters. The number of errors increases for the increasing duration of the SET. This is expected since a new input is added every clock cycle. So, the longer the duration the more input bits it will affect. The maximum number of errors for an SET also increases. The total number of clock cycles that are affected increases at about the same rate as the total number of errors. This is because each bit that the SET affects is a part of a different input, so it will be spread out over more clock cycles. This is also why the number of maximum number of clock cycles affected by the SET increases and is the same number as the duration length. Since the total number of errors and the total number of clock cycles increases at about the same rate, the errors per clock cycle does not change much but remains about 9.4.

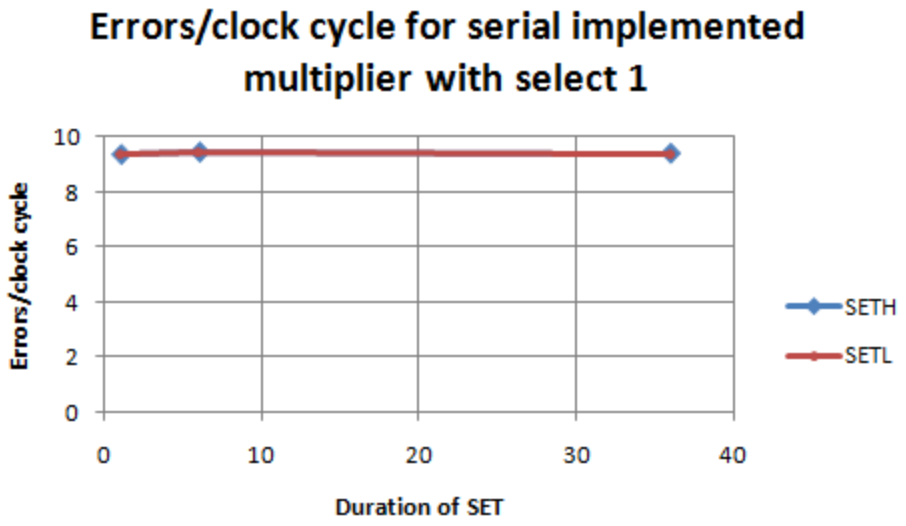


Figure 44: Errors per clock cycle in serial-implemented full adder multiplier results for select value of 1

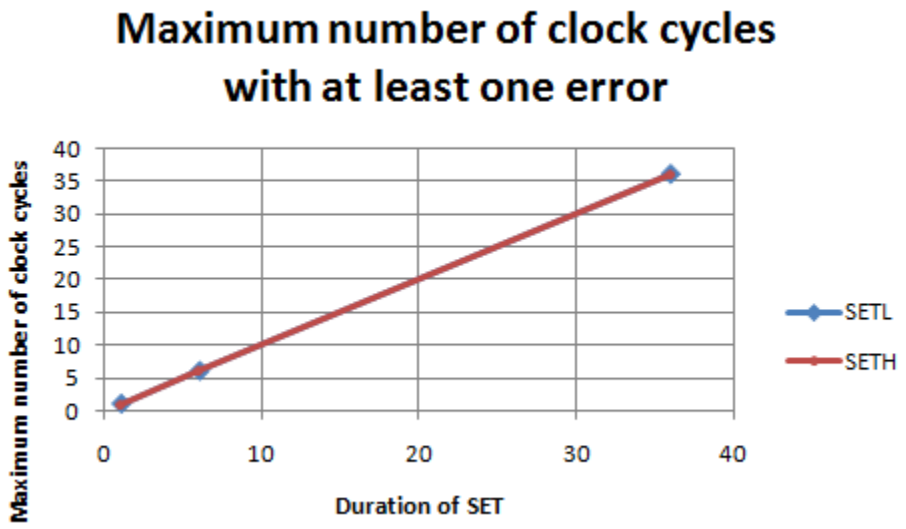


Figure 45: Maximum number of clock cycles with at least one error in serial-implemented full adder multiplier for select value of 1

The results for the nodes inside each of the multipliers are different. The total number of errors still increases but does so at the same rate as the total number of clock

cycles. For an SET with duration of one, there is not much difference for the errors per clock cycle for the serial-implemented multiplier or the parallel-implemented multiplier. This is expected because the SET lasts only one clock cycle. The errors per clock cycle for the parallel-implemented multiplier stayed the same when increasing the duration of the SET. The errors per clock cycle for the serial-implemented multiplier increased as the SET duration increased. This means that the errors for both multipliers are increasing, but for the serial-implemented multiplier, the errors are not being spread out over multiple clock cycles unless the SET lasts at least 47 clock cycles. This can best be seen in the maximum number of clock cycles. For the parallel-implemented multiplier, the maximum number of clock cycles in which an error occurred increased from duration of 1, 6, and 36. For the serial-implemented multiplier, the maximum number of clock cycles in which an error occurred mostly stays the same, because there are 47 different multiplies, and each input is put into a different multiplier. This way, if an SET is applied to a node, it will only affect one result instead of multiple results. With duration of 36, the maximum number of clock cycles in which an error occurs is 2. When it has finished loading the inputs into the last multiplier, it goes back to the first one and starts over. So, if an SET lasts longer than 47 clock cycles, it could affect multiple results, but this is assumes that the SET is applied to the node when it first starts computing each result. For the simulation, the SET was not always applied when the inputs were loaded but sometimes during the middle or end of a computation. This is why for duration of 36, the maximum number of clock cycles in which an error occurs is 2. If the SET is applied on a node at the end of a calculation then it will also last into the beginning of and have an effect on the next calculation. For a 16x16 bit serial-implemented full adder multiplier

set up in this fashion, the minimum length of an SET to possibly have an effect on two different outputs is 16 clock cycles.

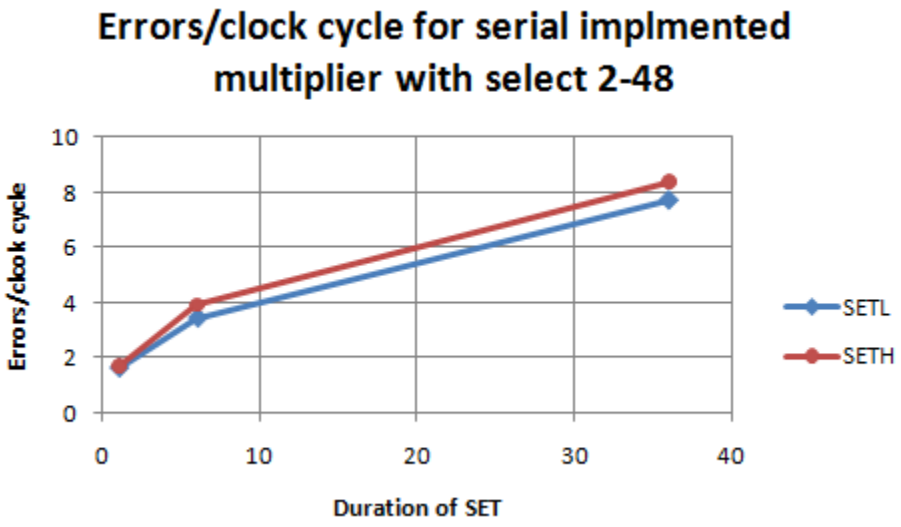


Figure 46: Errors per clock cycle in serial-implemented full adder multiplier results for select value of 2-48

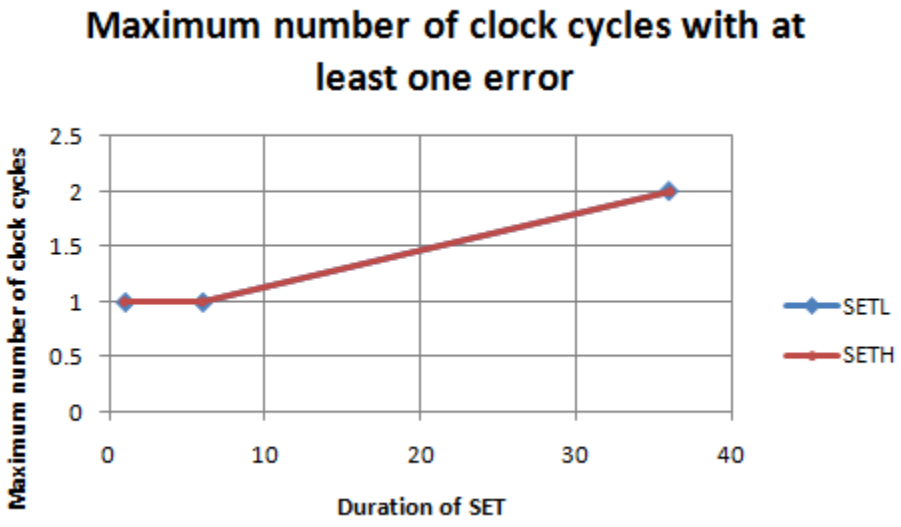


Figure 47: Maximum number of clock cycles with at least one error in serial-implemented full adder multiplier for select value of 2

4.2 Signal Analysis

The multiplier has two input signals. One is a constant 43,690 (Figure 48) and the other is a sinusoidal wave (Figure 49). The sinusoidal wave has amplitude of 32,767 and a non-zero center which is 32,767. This allows the signal to span over the entire range of numbers for a 16-bit value input (0 to 65,535). The function for the input signals is:

$$x_1 = 43{,}690, \tag{5}$$

$$x_2 = 32{,}767 + 32{,}767 * \sin\left(\frac{2 * \pi * n}{32}\right), \tag{6}$$

where *n* represents the points from 0 to 127.

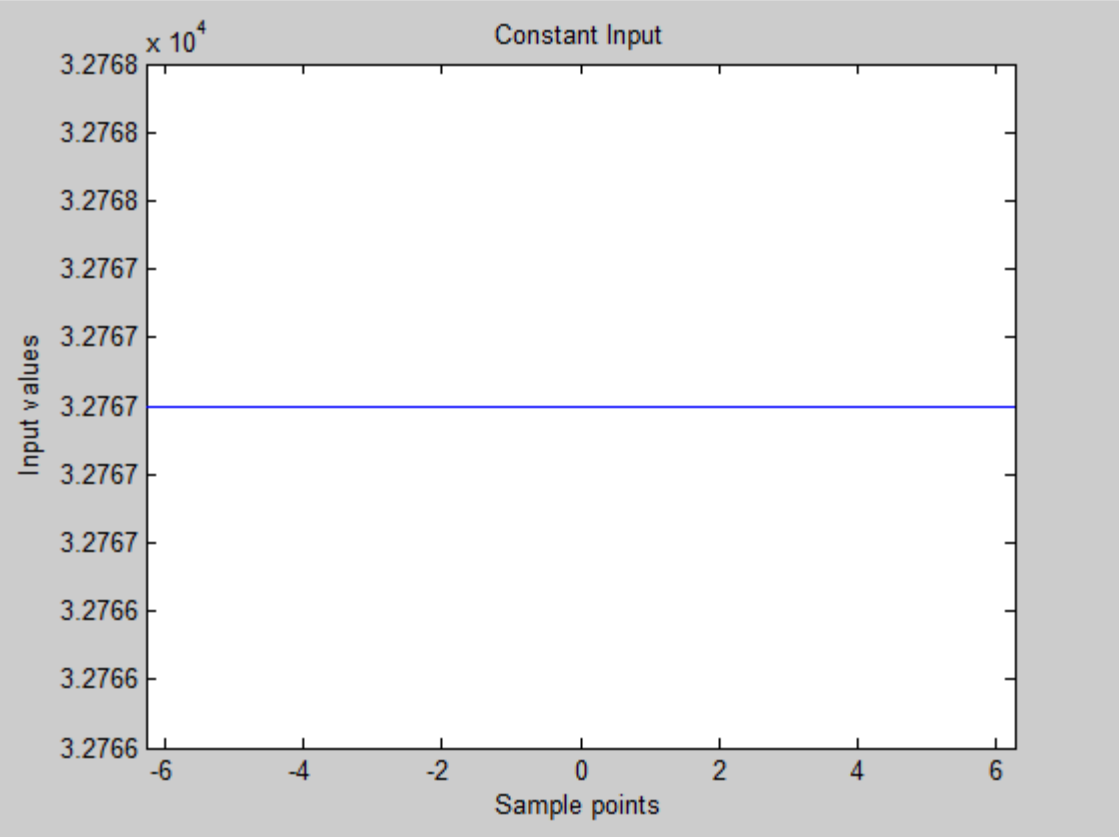


Figure 48: Constant input

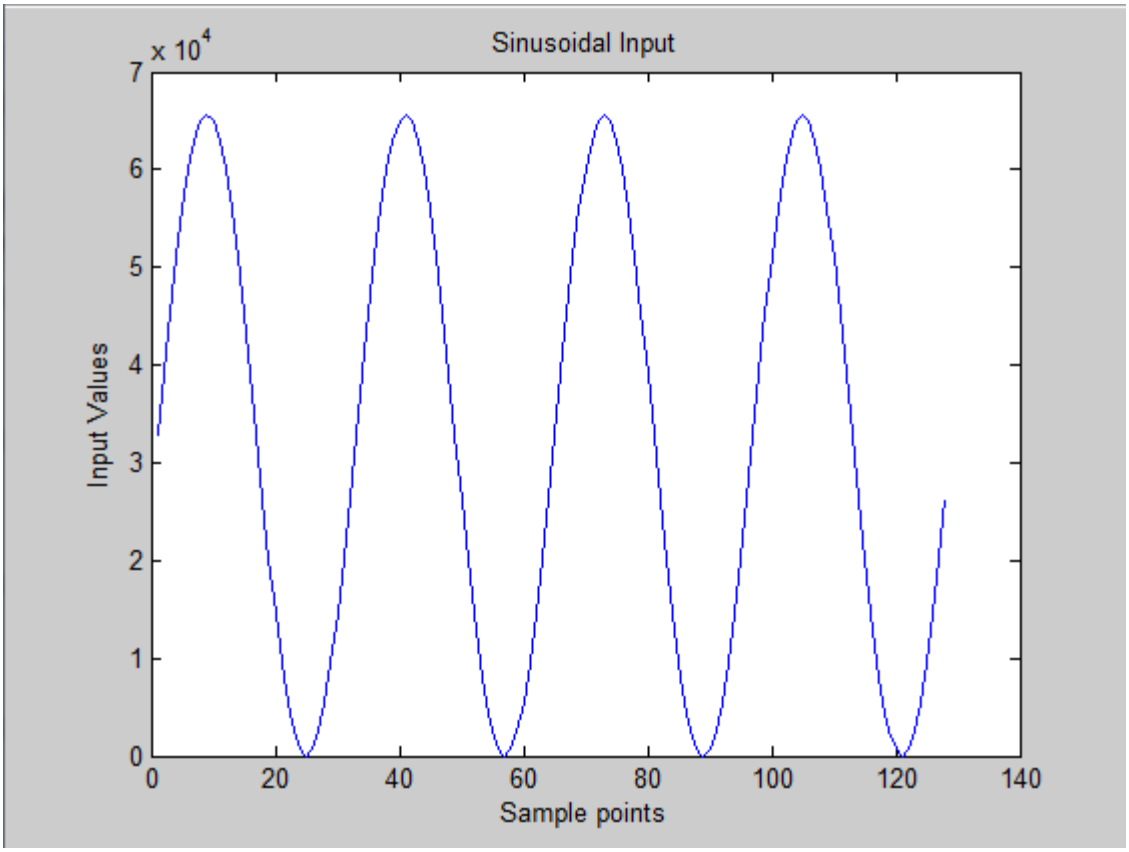


Figure 49: Sinusoidal input

When these two signals are multiplied together, they result in a sinusoidal signal that has been shifted up from the input sinusoid (Figure 50). The function for the output signal is:

$$y = 1431590230 + 1431590230 * \sin\left(\frac{2 * \pi * n}{32}\right), \quad (7)$$

where *n* represents the points from 0 to 127.

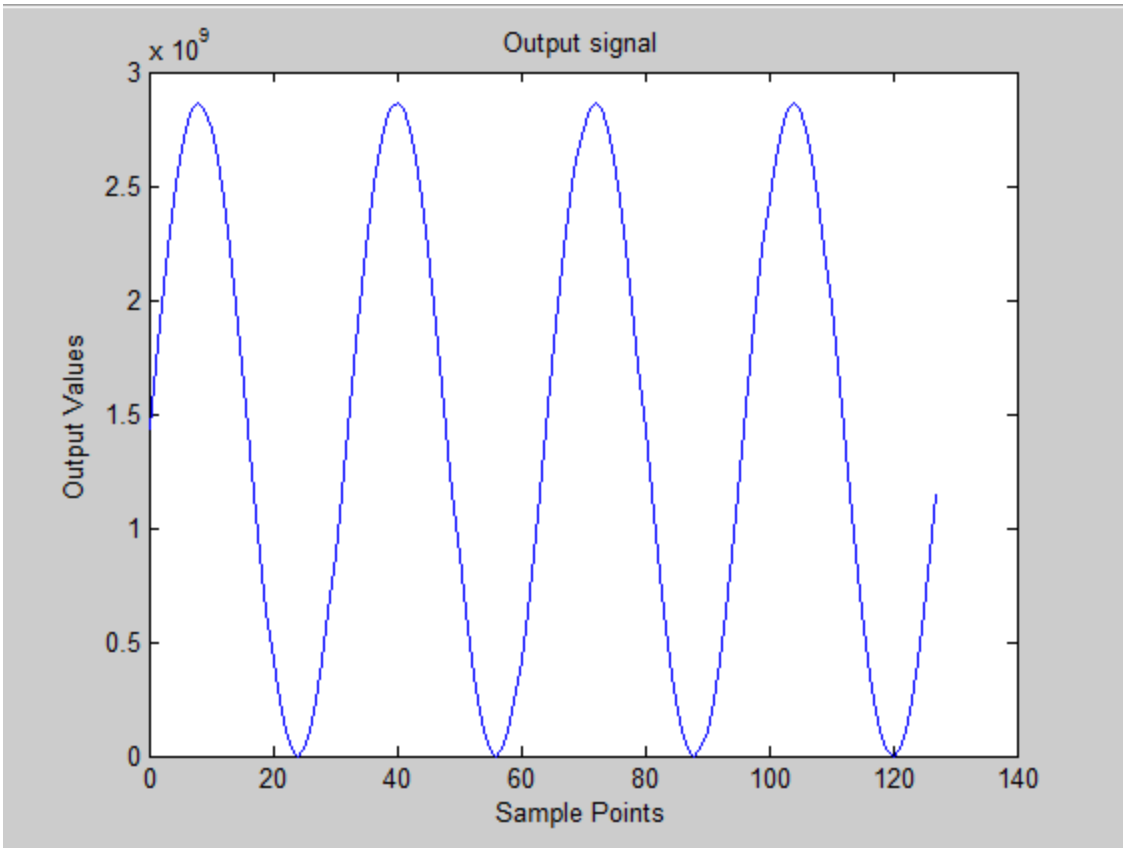


Figure 50: Output signal

A Fast Fourier Transform (FFT) is a fast algorithm for finding the Discrete Fourier Transform (DFT) [17]. The FFT takes the discrete signal that is in the time domain and transforms it to its discrete frequency domain representation.

The FFT is computed in MATLAB using the function FFT(x,N), where *x* is the signal and *N* is the number of points in the FFT. *N* has to be at least as large as the number of sample points. The FFT is performed on the output signal. The number of points for the signal is 32 for one period, the value of 128 is used for *N*, and the signal is the output signal of the multiplier. A plot of the FFT for the sinusoidal inputs signals and the output signal is shown in Figure 51, Figure 52, and Figure 53.

The Spurious-Free dynamic range (SFDR) can also be calculated from the FFT. The SFDR measures the difference between the maximum signal component of the carrier frequency and the next largest noise or distortion component. The SFDR range of the output signal without an error is 98 dBc (decibels relative to the carrier). The dBc unit is the power ratio of a signal to a carrier signal.

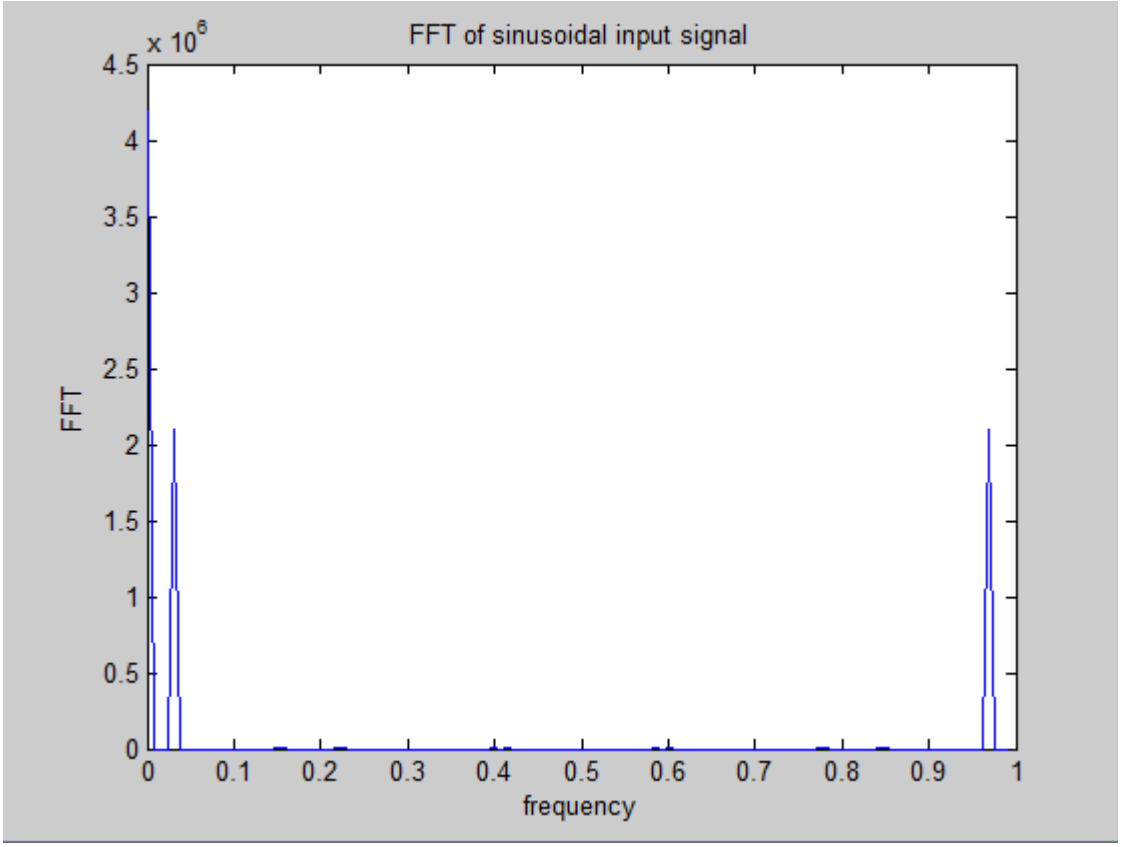


Figure 51: FFT of sinusoidal input

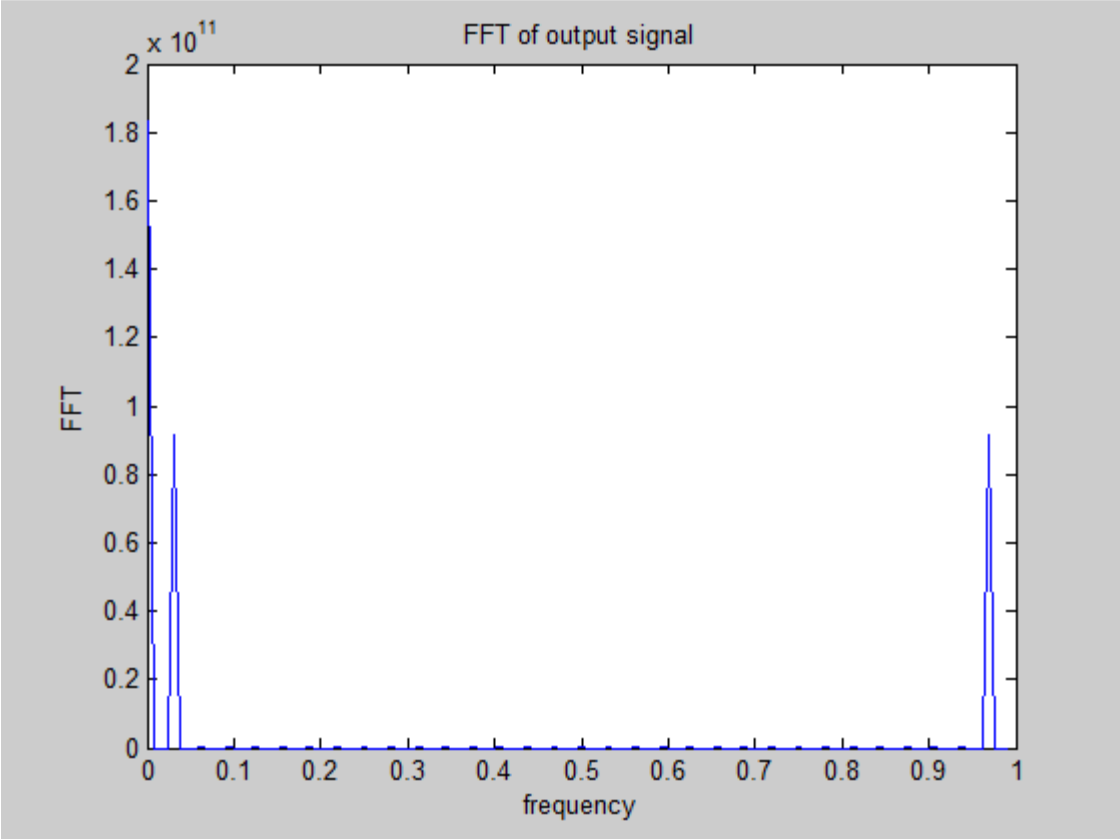


Figure 52: FFT of the output signal

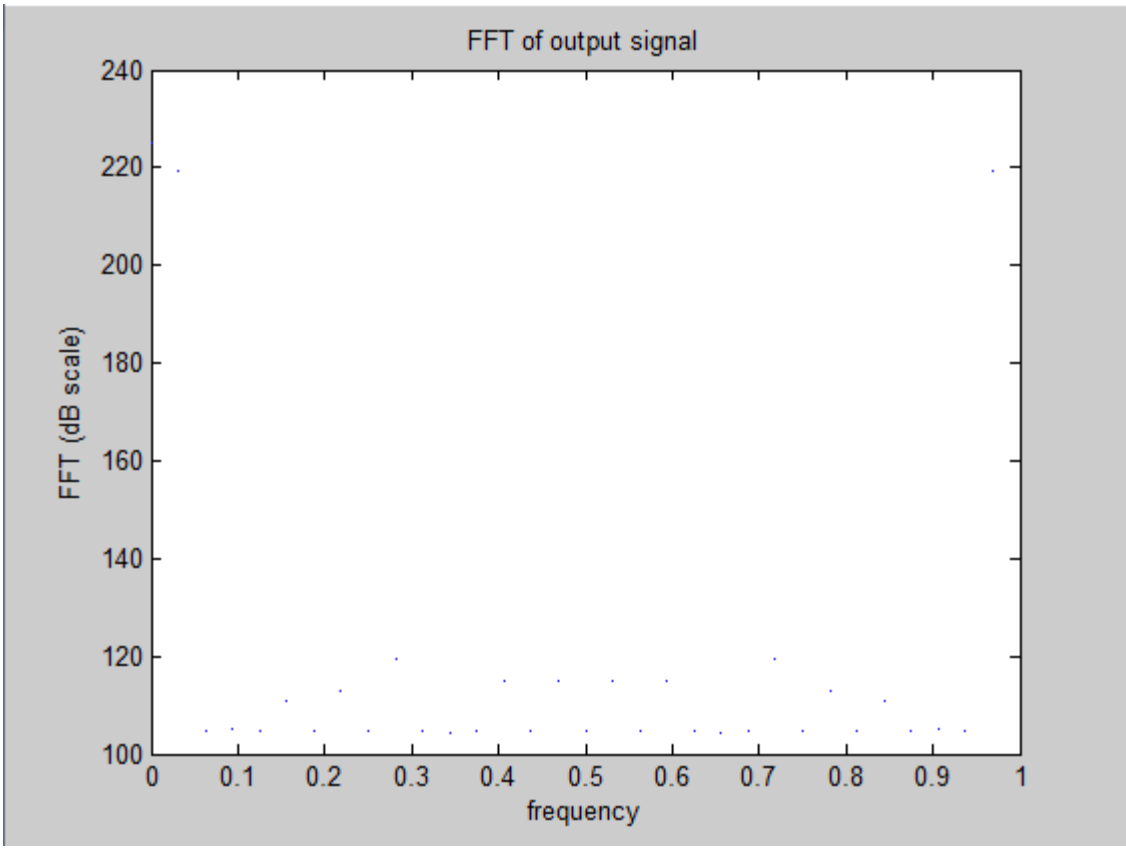


Figure 53: FFT for output signal with dB scale

Some of runs were made with both the serial-implemented and parallel-implemented multipliers and applied SETs of length 6 and 36. The graphs of the output signals are graphed and their FFTs (Figure 54, Figure 55, Figure 56, Figure 57, Figure 58, Figure 59, Figure 60, Figure 61, Figure 62, Figure 63, Figure 64, and Figure 65).

The SFDR for the parallel multiplier with an SET of duration 6 is 23 dBc (Figure 63). The SFDR for the serial multiplier with an SET of duration 6 and select value of 2-48 is 39 dBc (Figure 55). The SFDR for the parallel multiplier with an SET of duration of 36 is 27 dBc (Figure 65). The SFDR for the serial multiplier with an SET of duration of 36 and select value 2-48 is 49 dBc (Figure 59). Since the values for the serial

multiplier are higher than the parallel multiplier that means that the FFT for the serial multiplier is smoother than for the parallel multiplier.

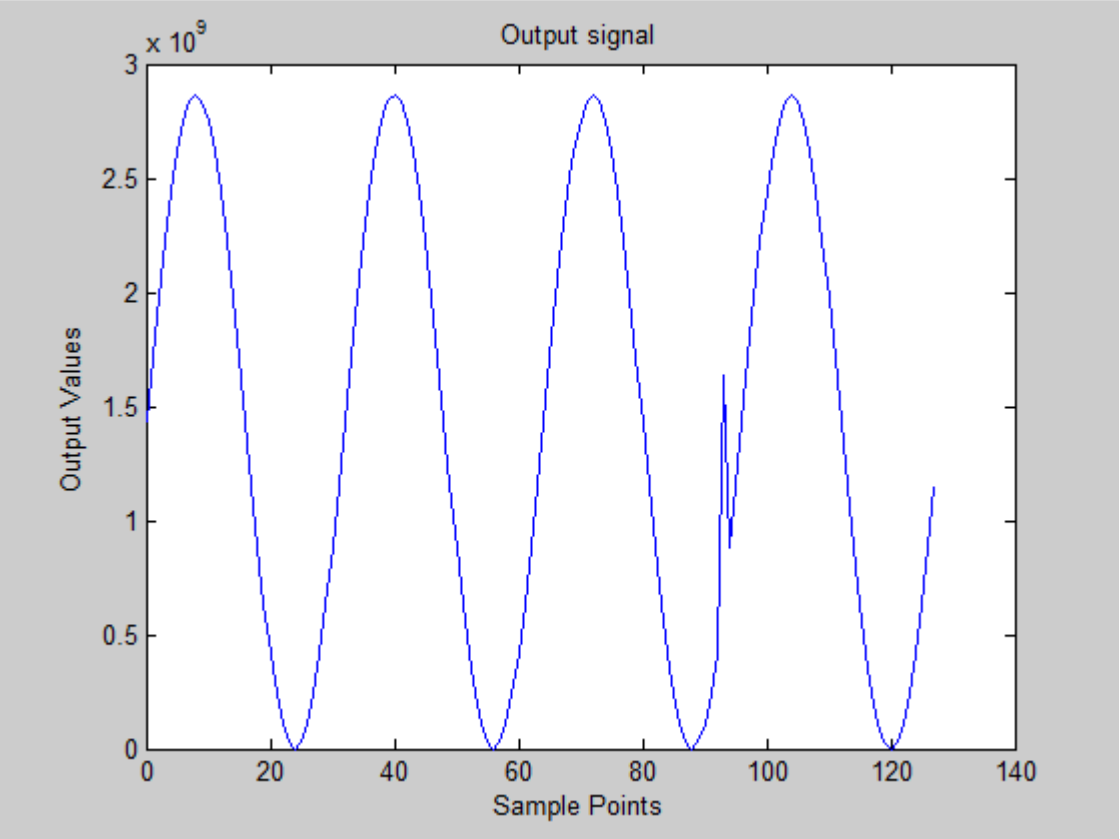


Figure 54: Output signal for serial-implemented multiplier with SET length of 6 and select value of 2-48

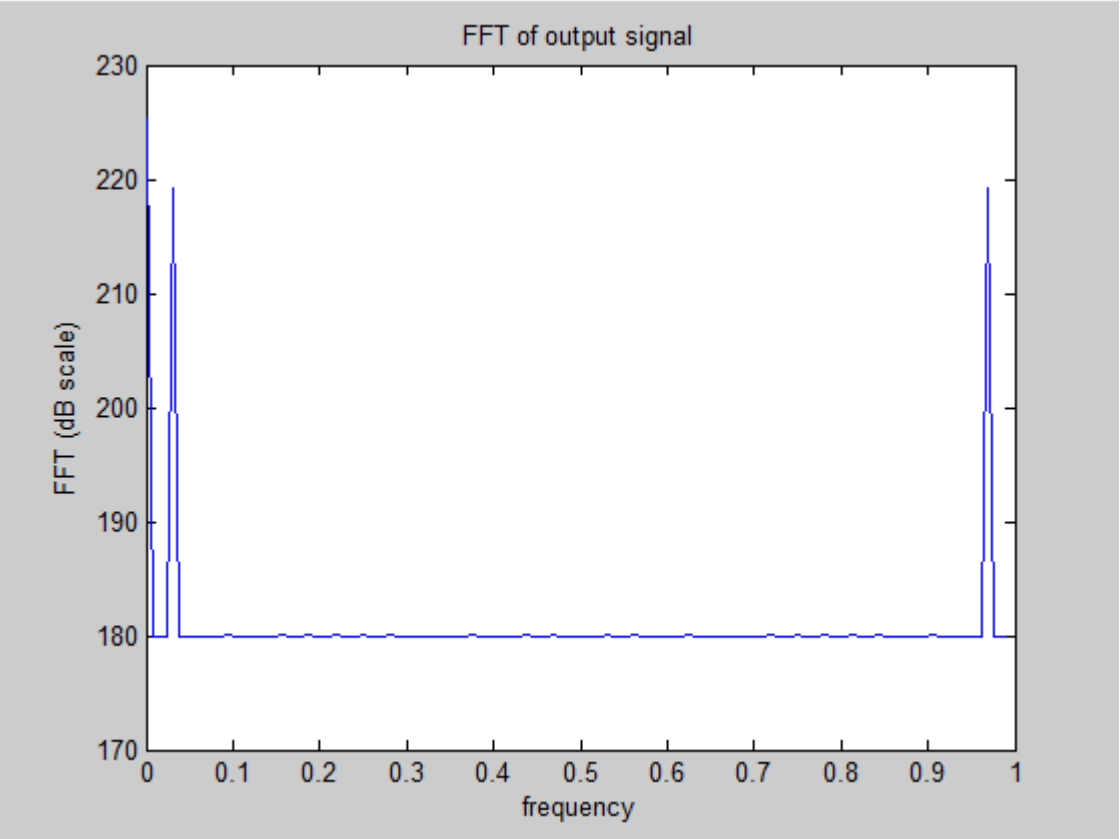


Figure 55: FFT for serial-implemented multiplier with SET length of 6 and select value of 2-48

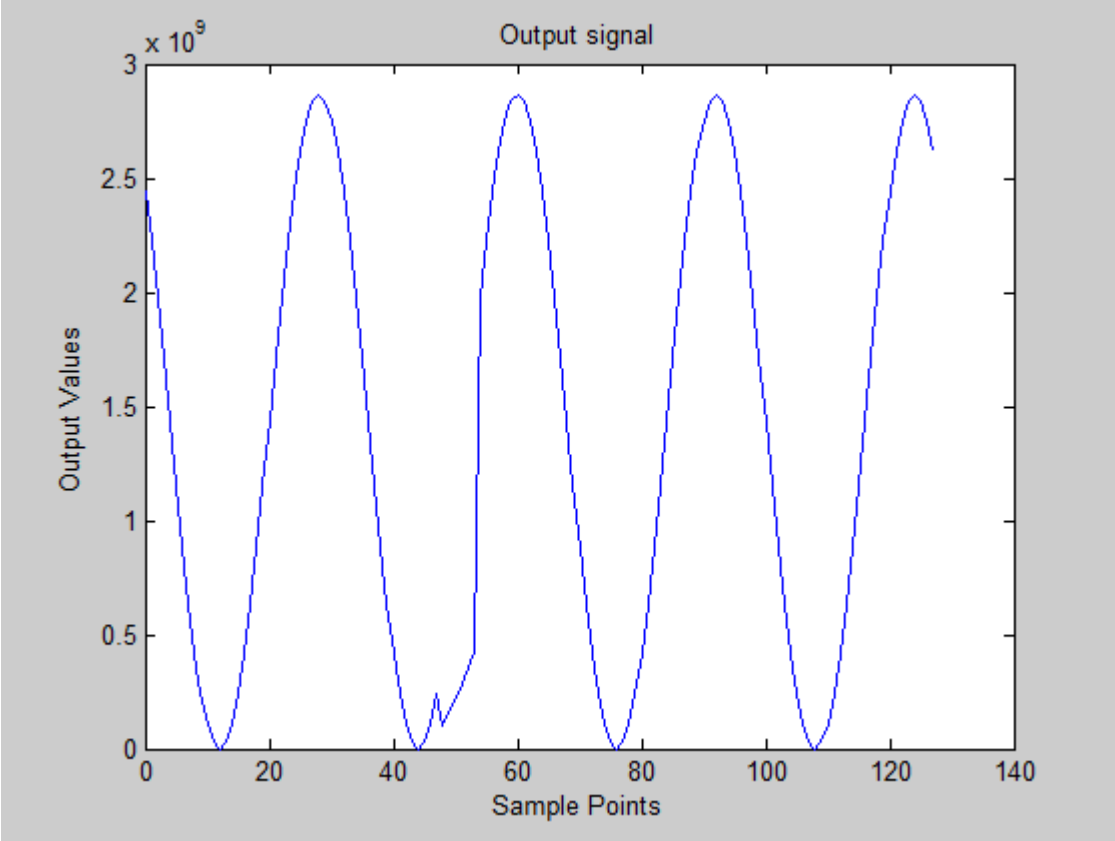


Figure 56: Output signal of serial-implemented multiplier with SET length of 6 and select value of 1

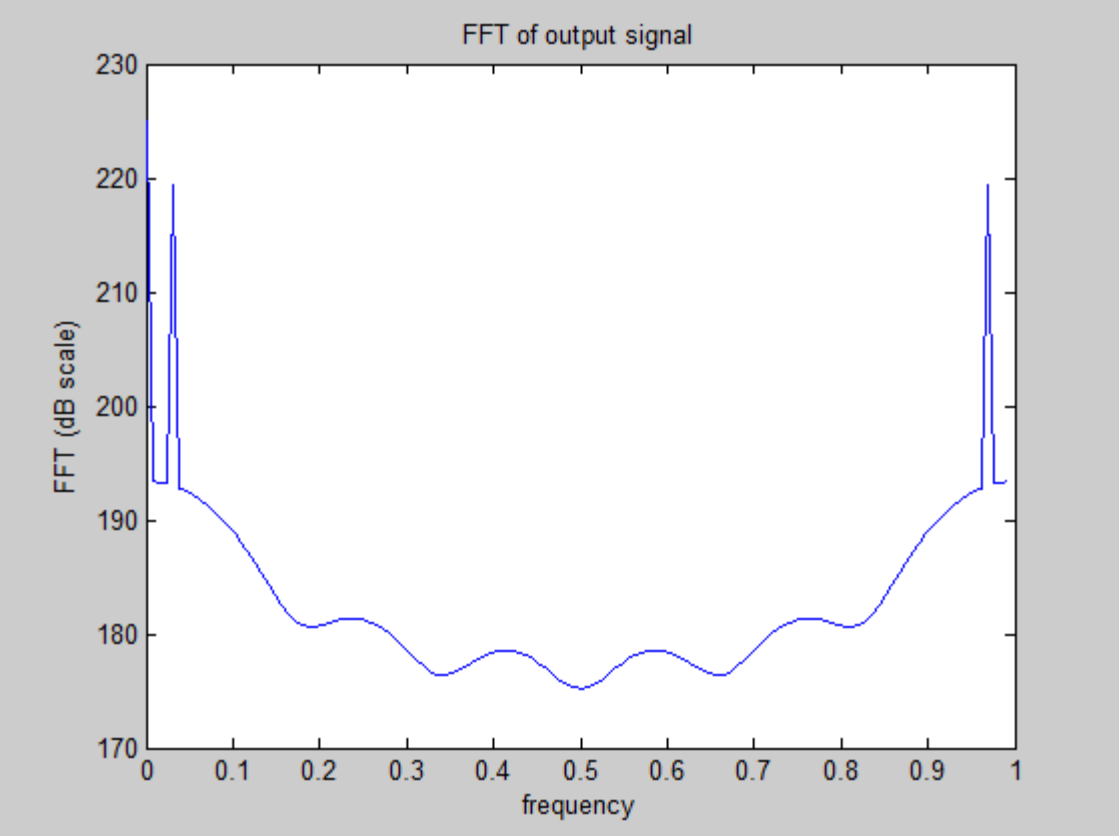


Figure 57: FFT of serial-implemented multiplier with SET length of 6 and select value of 1

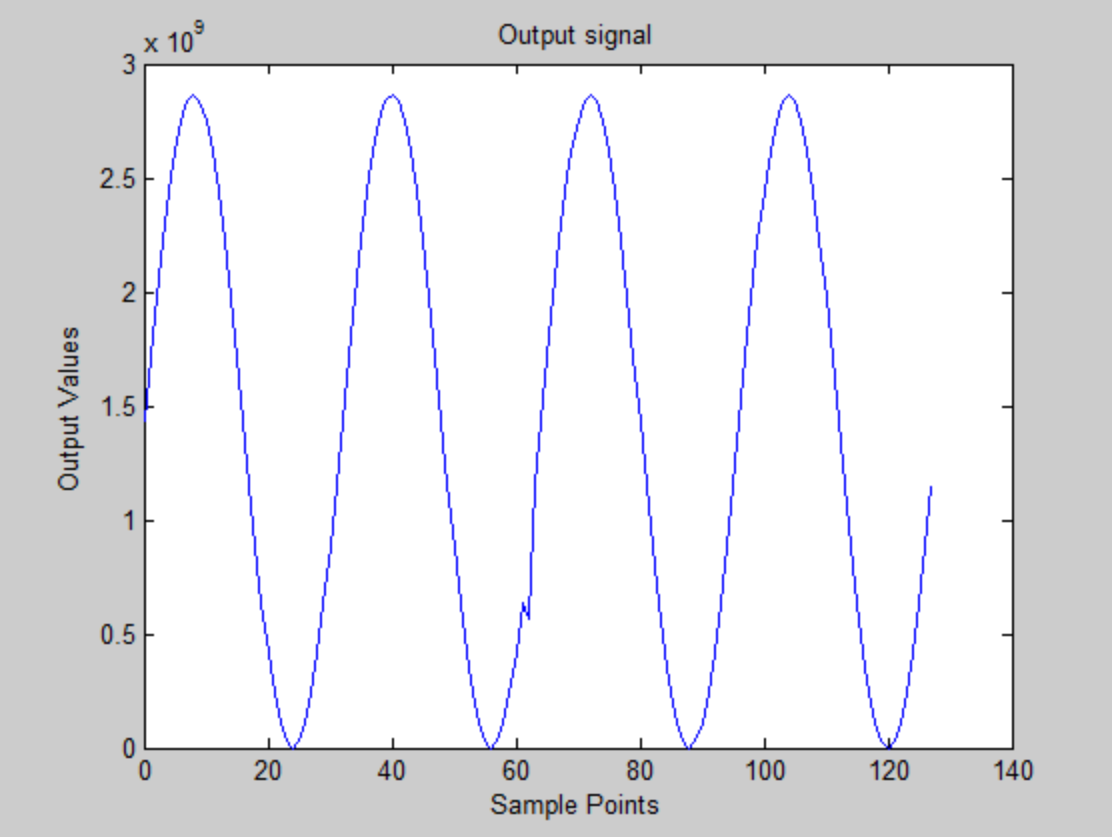


Figure 58: Output signal for serial-implemented multiplier with SET length of 36 and select value of 2-48

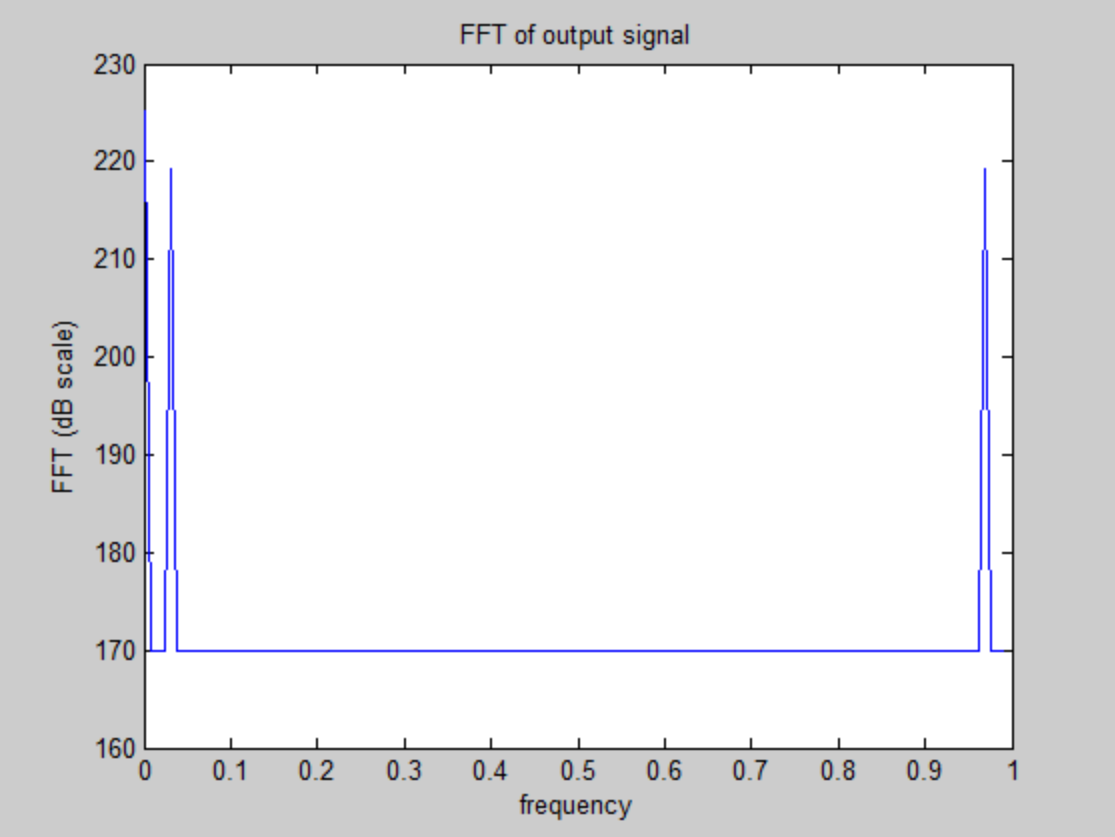


Figure 59: FFT for serial-implemented multiplier with SET length of 36 and select value of 2-48

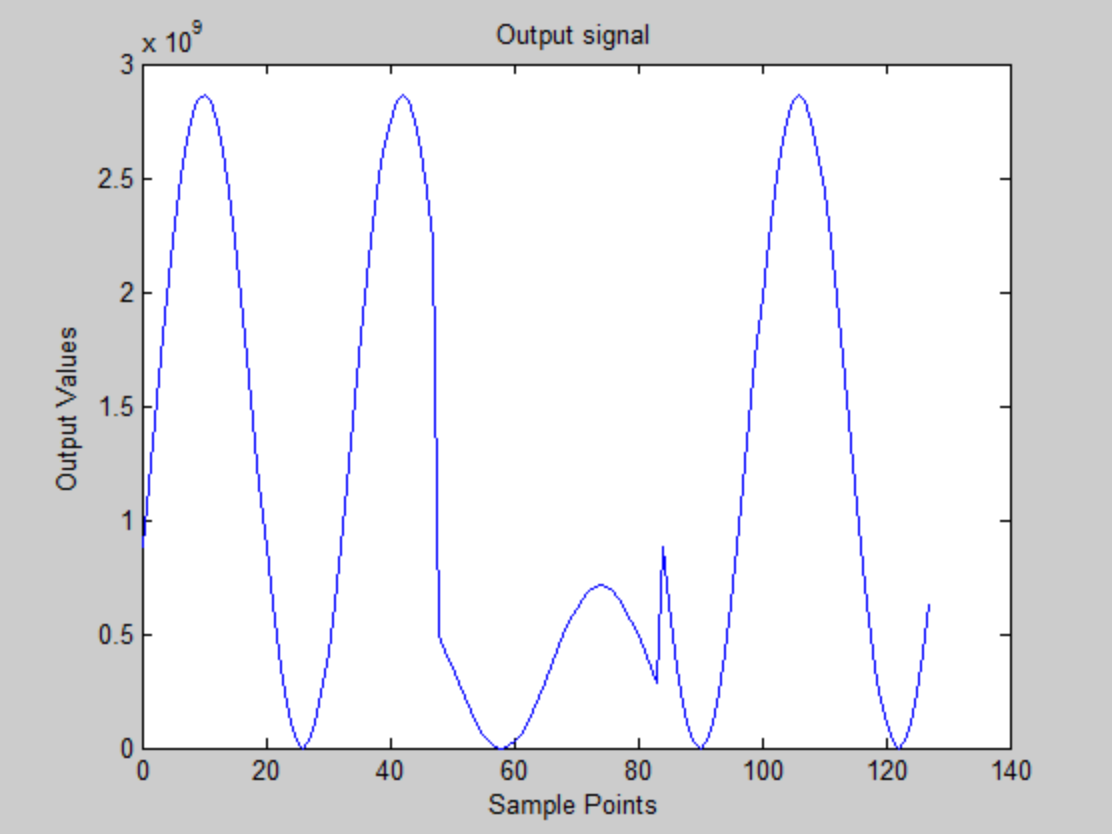


Figure 60: Output signal for serial-implemented multiplier with SET length of 36 and select value of 1

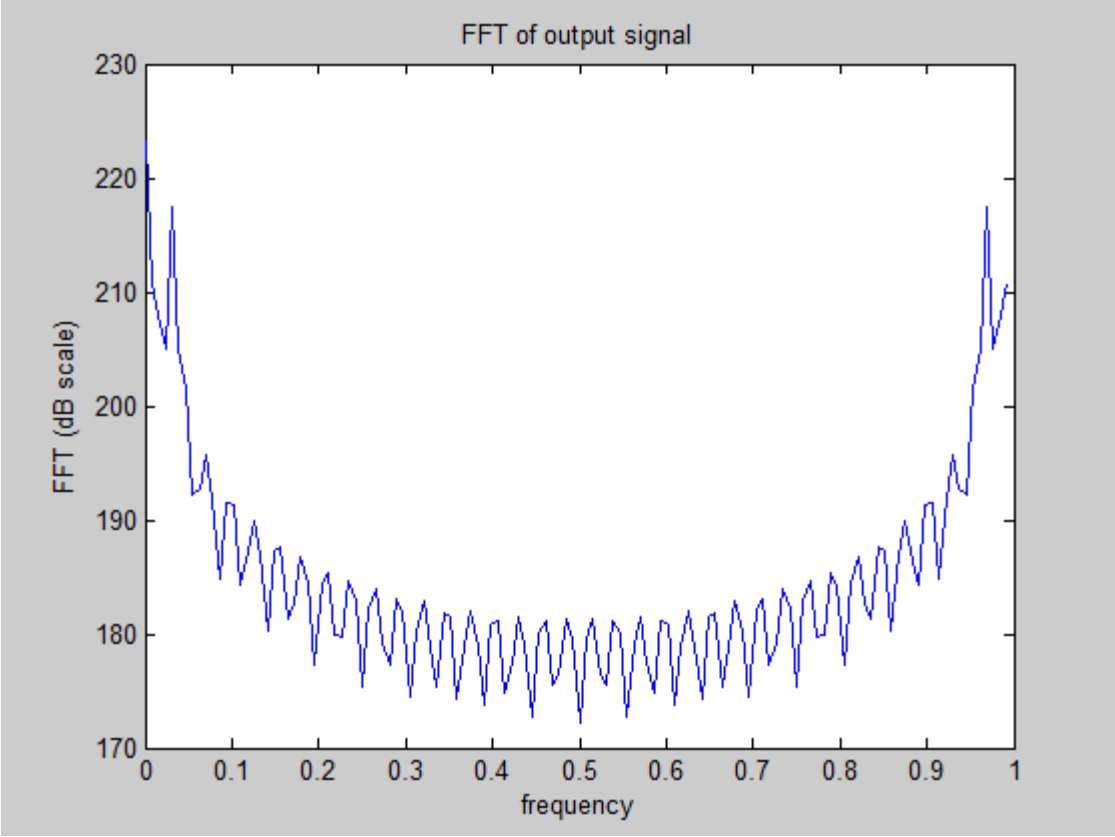


Figure 61: FFT for serial-implemented multiplier with SET length of 36 and select value of 1

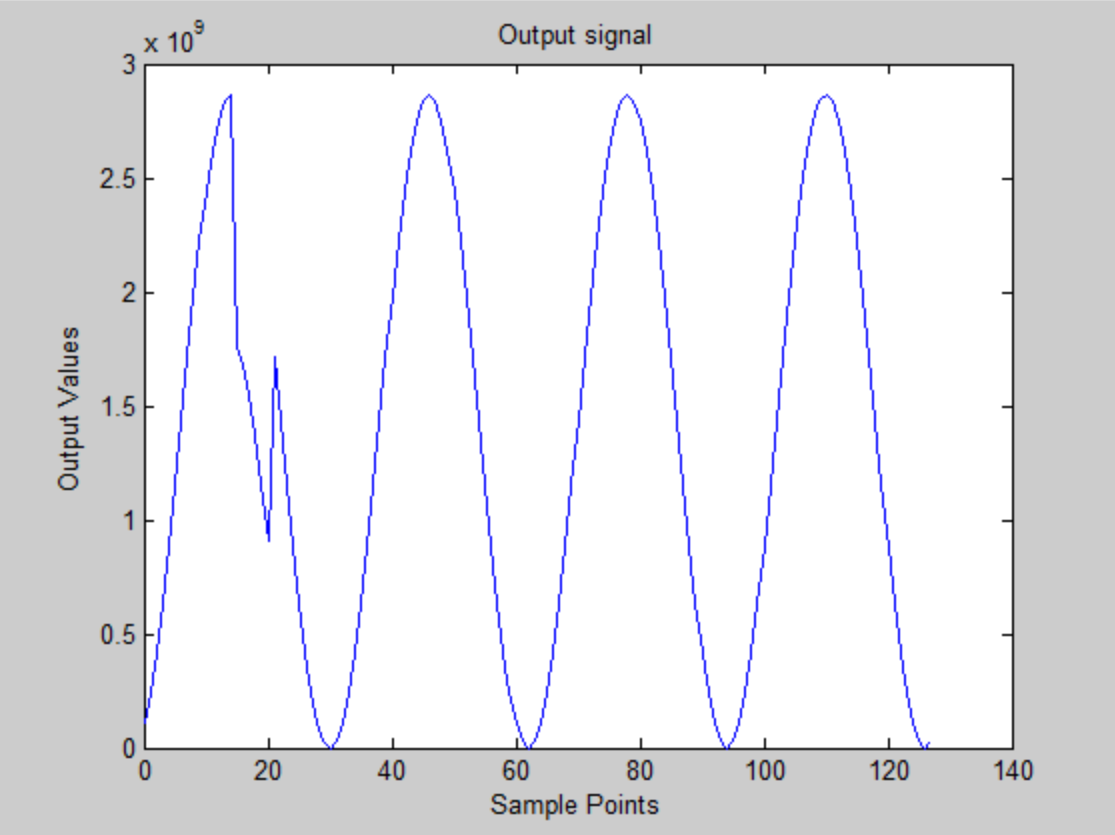


Figure 62: Output signal of parallel-implemented multiplier with SET length of 6

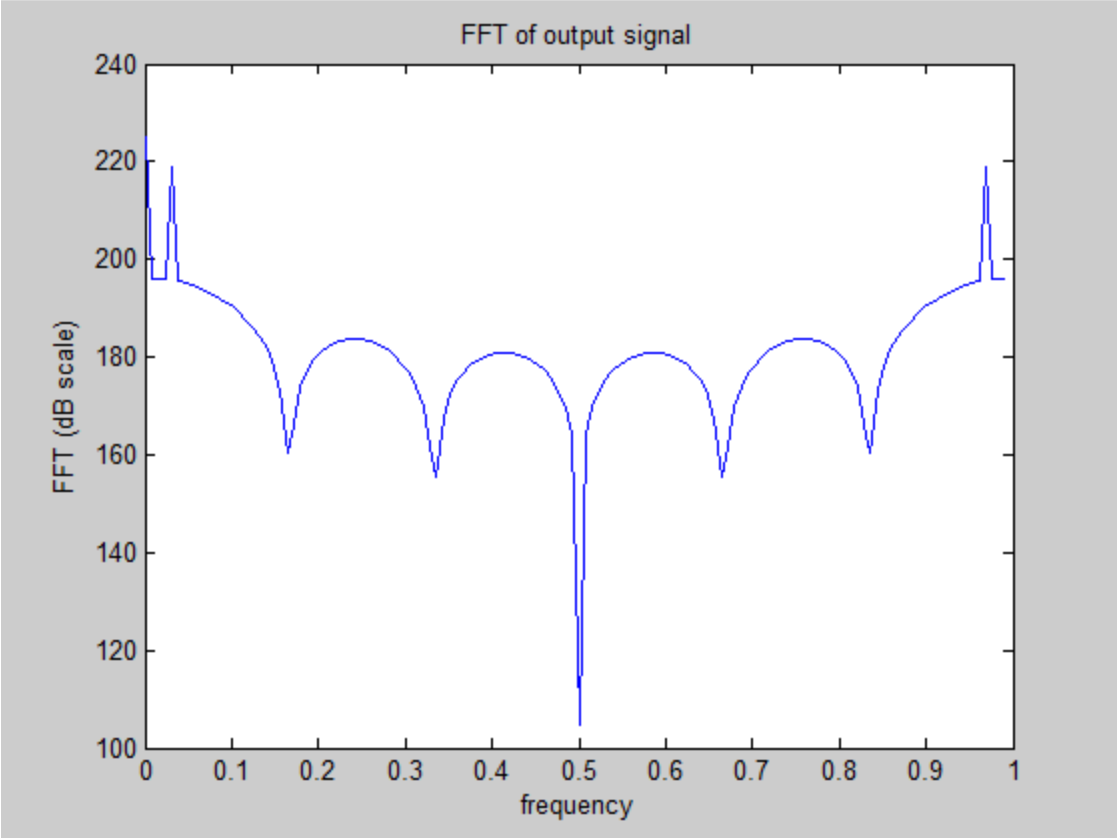


Figure 63: FFT of parallel-implemented multiplier with SET length of 6

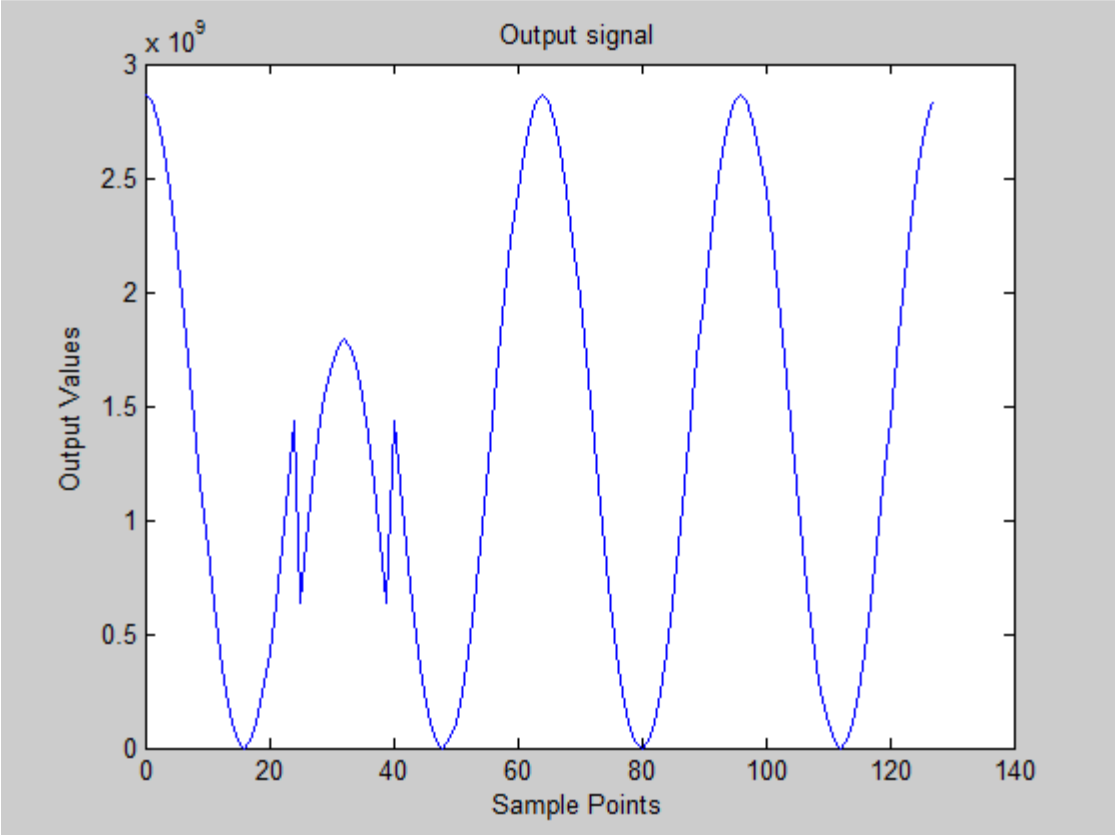


Figure 64: Output signal for parallel-implemented multiplier with SET length of 36

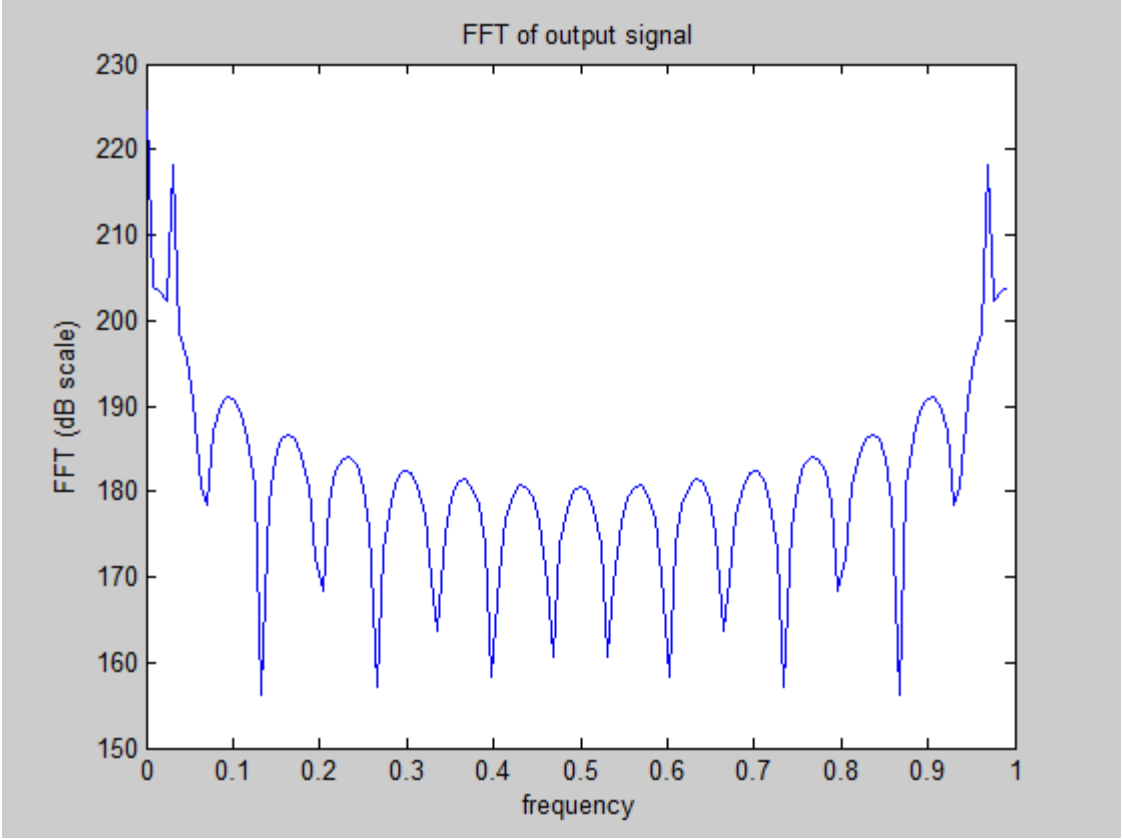


Figure 65: FFT of parallel-implemented multiplier with SET length of 36

CHAPTER V

CONCLUSION

This thesis examined the effects that a burst of errors has on a communication system and the method of de-interleaving to spread out the errors. The encoders can handle errors in the communication system as long as they do not occur in a burst. When a burst of errors occurs, all of the errors are bunched up together, and the number of errors is much greater. The method of de-interleaving spreads the errors out so that it is easier to reduce the effects of the errors.

A parallel-implemented full adder multiplier and a serial-implemented full adder multiplier was simulated with SET lengths of 1, 6, and 36 clock cycles applied to each of them. The errors produced were mostly identical for the SET lengths of 1 clock cycle. For the SET lengths of 6 and 36 clock cycles, the errors for the parallel-implemented full adder multiplier often extended over multiple clock cycles, but for the serial-implemented full adder multiplier, most of the errors were confined to one clock cycle. When the SET was applied on a node at the end of the calculation and lasted into the next one, the errors in the output values were extended to two clock cycles, but separated by 47 clock cycles. This means that if an SET last longer than 47 clock cycles, then it can affect multiplier outputs but errors will be separated by 47 clock cycles.

When bursts occurred in the communication system and when the SET was applied to the serial-implemented full adder multiplier, the results of the errors were similar. The errors for the communication system were bunched together, and the errors in the serial-implemented full adder multiplier were bunched together usually in one clock cycle. When the de-interleaving process was applied to the communication system and the SET was applied to the parallel-implemented full adder multiplier, the results of the errors were similar. The errors for the communication system were spread out instead of being bunched together, and the errors for the parallel-implemented full adder multiplier were spread out over many clock cycles instead of all being in one.

The size and speed were examined for the parallel-implemented multiplier and the serial-implemented multiplier. The parallel-implemented multiplier is slower because the speed of the circuit is set by the full adders and the carry chain. For the serial-implemented multiplier, the speed is set by the full adders and the latency of the input to the output. The parallel-implemented multiplier has fewer logic gates than the serial-implemented multiplier. The serial-implemented multiplier has a total of 4,465 logic gates, whereas the parallel-implemented multiplier has a total of 1,279 logic gates. Using triple modular redundancy and voting on the parallel-implemented multiplier would still end up with less logic gates than the serial multiplier. However, reducing the latency for the serial multiplier would reduce the number of copies needed, which will reduce the number of logic gates.

REFERENCES

[1] G. Niu, R. Krithivasan, J. D. Cressler, P. A. Riggs, B. A. Randall, P. W. Marshall, R. A. Reed, and B. Gilbert, "A Comparison of SEU Tolerance in High-Speed SiGe HBT Digital Logic Designed With Multiple Circuit Architectures," *IEEE Transactions on Nuclear Science,* vol. 49, pp. 3107-3114, Dec. 2002.

[2] W. H. W. Tuttlebee, *Software Defined Radio: Baseband Technology for 3G Handsets and Basestations,* 2004.

[3] E. Chapman, J. Jackson, D. Robison, R. E. Choueiry, M. Musisi-Nkambwe, B. Zamito, "Software Defined Radio," *Proceeding of KGCOE-MD2004: Multi-Disciplinary Engineering Design Conference,* May 2004.

[4] W. Kogler, "Software defined radio: Digital mixer and numerical controlled oscillators," Presentation given at *Institute for Communication Networks and Satellite Communications.*

[5] P. E. Dodd and L. W. Massengill, "Basic mechanisms and modeling of single-event upset in digital microelectronics," *IEEE Transactions on Nuclear Science,* vol. 50, pp. 583-602, June 2003.

[6] R. C. Baumann, "Soft errors in advanced semiconductor devices-part I: the three radiation sources," *IEEE Transactions on Device and Materials Reliability,* vol. 1, pp. 17-22, Mar. 2001.

[7] J. Black and W. T. Holman, "Circuit Simulation of Single Event Effects," *IEEE NSREC Short Course*, 2006.

[8] S. E. Diehl, J. E. Vinson, B. D. Shafer, and T. M. Mnich, "Considerations for single event immune VLSI logic," *IEEE Transactions on Nuclear Science,* vol. 30, pp. 4501-4507, Jan. 1983.

[9] L. W. Massengill, A. E. Baranski, D. O. Van Nort, J. Meng and B. L. Bhuva, "Analysis of Single-Event Effects in Combinational Logic - Simulation of the AM2901 Bitslice Processor," *IEEE Trans. Nuclear Science*, vol. 47, pp.2609-2615, Dec. 2000.

[10] A. Leuciuc, B. Zhao, Y. Tian, and J. Sun, "Analysis of single-event effects in continuous-time delta-sigma modulators," *IEEE Trans. Nuclear Science,* vol. 51, pp. 3519–3524, Dec. 2004.

[11] S. Rezgui, G. M. Swift, R. Velazco, and F. F. Farmanesh, "Validation of an SEU Simulation Technique for a Complex Processor: PowerPC7400," *IEEE Trans. on Nuclear Science*, vol. 49, pp. 3156-3162, Dec. 2002.

[12] B. Sklar, “Rayleigh Fading Channels in Mobile Digital Communication Systems Part II: Mitigation,” *IEEE Communications Magazine,* vol. 35, pp. 102-109, Jul. 1997.

[13] R. L. Bogusch, *Digital Communications in Fading Channels: Modulation and Coding*, AFWL-TR-87-52, MRC-R-1043, April 1989.

[14] G. D. Forney Jr., “The Viterbi Algorithm,” *Proceedings of the IEEE,* vol. 61, pp. 268-278, Mar. 1973.

[15] P. T. Balsara and D. T. Harper III, “Understanding VLSI Bit Serial Multipliers,” *IEEE Trans. on Education*, vol. 39, no. 1, Feb. 1996.

[16] C. H. Roth, Jr. and L. K. John, *Digital Systems Design Using VHDL*, Second Edition, Thomson Learning, 2008.

[17] E. O. Brigham, R. E. Morrow, “The Fast Fourier Transform,” *IEEE Spectrum*, Dec. 1967.

[18] P. E. Dodd, M. R. Shaneyfelt, J. A. Felix, and J. R. Schwank, "Production and propagation of single-event transients in high-speed digital logic ICs," *IEEE Transactions on Nuclear Science,* vol. 51, pp. 3278-3284, Dec. 2004.

[19] J.S. Kauppila, L.W. Massengill, W.T. Holman, A.V. Kauppila, and S. Sanathanamurthy, “Single event simulation methodology for analog/mixed signal design hardening,” *IEEE Trans. Nuclear Science*, vol. 51, pp. 3603-3608, Dec. 2004.

[20] Y. Boulghassoul, J.D. Rowe, and L.W. Massengill, “Applicability of circuit macromodeling to analog single-event transient analysis,” *IEEE Trans. Nuclear Science*, vol. 50, pp. 2119-2125, Dec. 2003.

[21] N. Kaul, “Computer-Aided Estimation of Vulnerability of CMOS VLSI Circuits to Single-Event Upsets,” PhD Dissertation, Dept. of Electrical Engineering, Vanderbilt University, 1992.

[22] N. Kaul, B. L. Bhuva, S. E. Kerns, "Simulation of SEU Transients in CMOS ICs," *IEEE Trans. Nuclear Science*, vol. 38, pp. 1514-1520, Dec. 1991.

[23] C. Grimm and K. Waldschmidt, “KIR-a-graph-based model for description of mixed analog/digital systems”, *Design Automation Conference,* pp.568-573, Sept. 16-20 1996.

[24] G. Niu, R. Krithivasan, J. D. Cressler, P. A. Riggs, B. A. Randall, P. W. Marshall, R. A. Reed and B. Gilbert, “A Comparison of SEU Tolerance in High-Speed SiGe HBT Digital Logic Designed With Multiple Circuit Architectures,” *IEEE Trans. Nuclear Science*, vol. 49, pp.3107-3114, Dec. 2002.

[25] L. W. Massengill, "SEU Modeling and Prediction Techniques," *IEEE NSREC Short Course*, 1993.

APPENDIX A

Code for a NAND module

This is the code for a NAND module. This modulus takes the inputs from the testbench and applies them to the NAND gate. This NAND gate is set up so that an SET can be applied to it as well. The modulus has four input signals **A**, **B**, **SETH** and **SETL**. The input signals **A** and **B** are the input signals for the NAND gate and the input signals **SETH** and **SETL** are the SETs that can be applied to the NAND gate. If the **SETH** signal is applied, logic high, then the output on the NAND gate will be held high for as long as **SETH** signal is applied. If the **SETL** signal is applied, logic high, then the output on the NAND gate will be held low for as long as **SETL** signal is applied. The input signals **SETH** or **SETL** are applied when they are set high in the stimulus and they are not effective when they are set low in the stimulus.

CODE:

```
//timescale for simulation
`timescale 1ns/100p
//the library needed to run the NAND gate in the simulation
`include "setlibraryfile.v"
//beginning of the NAND modulus
module nand2_1x_SETH (A, B, y, SETH, SETL);
//inputs
        input A, B, SETH, SETL;
//output
```

```
        output y;
//wires
        wire a0_buf,a1_buf, a2_buf, a3_buf;
//NAND gate for Boeing library and primitive NOR gates
        nand2_1x a1( .in0(A), .in1(B),.y(a0_buf));
        nor nor0 (a1_buf, a0_buf, SETH);
        nor nor1 (y,a1_buf,SETL);

//end of modulus
endmodule
```

APPENDIX B

Code for the NAND testbench

This is the code for the NAND testbench. The testbench sets up the inputs that are to be applied to the modulus and can print out the results for the given inputs. The testbench first set up the initial values. The number of clock iterations is listed when the inputs should change along with the value that the inputs will be change to. This testbench changes the inputs for the NAND gate every clock cycle. The #2 is one clock iteration. The inputs go through all the different combinations for input signals **A** and **B** three times. The input signals **SETH** and **SETL** are initially set to low but the **SETL** signal is applied for the second combination of inputs. The stimulus will graph the values for the inputs and the outputs.

CODE:

```
//timescale for the simulations
`timescale 1ns / 100ps
//beginning of the stimulus
module nand2_1x_SETH_stim;
//regs
  reg A, B, clk, SETH, SETL;
//wires
  wire y;
//instantiating the NAND modulus
```

```
  nand2_1x_SETH  nand2_1x_SETH_instance1 ( A, B, y, SETH, SETL );


  //stimulus information
//the initial value for the clock
  initial
    clk = 1'b0;
//repeating the clock cycle
  always
    #1 clk = ~clk;
//initializing the values for the inputs
  initial
    begin
       A = 1'b0; B = 1'b0; SETL = 1'b0; SETH = 1'b0;
 //changing the inputs every clock cycle
//the first combination of inputs with SETH and SETL set to low
    #2  A = 1'b0; B = 1'b0;


    #2  A = 1'b0; B = 1'b1;


    #2  A = 1'b1; B = 1'b0;


    #2  A = 1'b1; B = 1'b1;
//the second combination of inputs with SETL set to high and SETH still low
    #2  A = 1'b0; B = 1'b0; SETL =1'b1;


    #2  A = 1'b0; B = 1'b1;
```

```
    #2  A = 1'b1; B = 1'b0;

    #2  A = 1'b1; B = 1'b1;

    #2  A = 1'b0; B = 1'b0;
//the third combination of inputs with SETL and SETH set to low
    #2  A = 1'b0; B = 1'b1; SETL=1'b0;

    #2  A = 1'b1; B = 1'b0;

    #2  A = 1'b1; B = 1'b1;
//wait one clock cycle and display “simulation complete” and end the simulation
    #2
      $display("\n Simulation complete\n");
      $finish;
    end

 // probe information
//display the information for the inputs A, B, SETH, SETL and the output y.
  initial
    begin
      $display("               Time        Inputs        Outputs");
      $display("                          A  B    SETH SETL     y");
      $display("               ====      =========      =======");
      $monitor($time,"       %b  %b   %b  %b   %b", A, B, SETH, SETL,  y);
    end
//end the stimulus
```

```
endmodule
```

APPENDIX C

Code for a 1-to-2 encoder

This is the code for a 1-to-2 encoder in Matlab®. It starts out by having the user enter the number of samples. From the input in creates a random sample of inputs and plots them. It then encodes the inputs by sending it through an encoder. Since it is a 1-to-2 encoder the encoded data is twice the size of the input data. For every input it produces two outputs. The random noise is generated by creating a set of random numbers from 0 to 1 that is twice as long as the input data. The encoded data is then added to the noise using modulo-2 addition. If any of the noise values are 1 then it will flip the bit in the encoded data. The data is then decoded using the decoder and it computes the bit error rate.

CODE:

```
clear %delets all variables in the working directory
clc    %clears the command window

%input data
%user inputs the number of samples and it creates a random sample for the input data
%plots the data using a stem plot
n=input('Enter number of samples: ')%number of samples
a=randint(n,1); %creates random samples, input data
subplot(331)
stem(a)%plots input
title('Input')

%encode the input
%sets up the trellis
%the first value is the constraint length and the second set of values is the generator
polynomials
%plots the encoded data using a stem plot
t = poly2trellis(3,[6,7]); % Define trellis
```

```
code = convenc(a,t); % Encode the data
subplot(332)
stem(code)%plots encoded data
title('Encoded data')

%random noise
%creates a random noise of values between 0 and 1 that is twice the size of the input data
x=rand(2*n,1)%noise
y=x
i=1;
%sets up a loop that will loop through each value in the random noise
while i<=length(x)
%if the value for the noise is greater than 0.028 set it equal to 0
%a value of 0 will not flip the bit in the encoded data
if (x(i)>0.028)
   y(i)=0
   i=i+1;
%if the value for the noise is greater than or equal to 0.01 then set it equal to 1
%a value of 1 will flip the bit in the encoded data
elseif (x(i)>=0.01)
   y(i)=1
   i=i+1;
%if the value for the noise is less than 0.01 then create a random noise sample that is has
a value of 1 half of the time
%this will create a burst of errors
elseif (x(i)<0.01)
   s=16;
   z=rand(s,1)
   b=z
   for u=1:length(z)
      if (z(u)>0.5)
         b(u)=0
      else b(u)=1
      end
      if i+u-1<=length(x)
       y(i+u-1)=b(u)
      end
   end
   i=i+16;
end
end
subplot(333)
stem(y)%plots noise
title('noise')

%adds noise and encoded data
```

```
%performs a modulo 2 addition on the data
%if the value of the result is 2 then it becomes 0
c=code+y
subplot(334)
stem(c)
title('noise+encoded data')
p=c
for i=1:length(c)
   if c(i)==0
      p(i)=0;
   elseif c(i)==1
      p(i)=1;
   elseif c(i)==2
      p(i)=0;
   end
end
subplot(335)
stem(p)
title('xor')

%decode data
%uses a Viterbi decoder to decode the data
%plots the decoded data
qcode = quantiz(p,[0.001,.1,.3,.5,.7,.9,.999]);%quantize data

tblen = 5; delay = tblen; % Traceback length
decoded = vitdec(qcode,t,tblen,'cont','soft',3); % Decode
subplot(336)
stem(decoded)
title('decoded data')

%Compute bit error rate
%calculates the number of errors and the number of total values and find the ratio of the
two numbers.
F=a(1:end-5)
D=decoded(decoded+6:end)
number=0
total=0
for i=1:length(F)
   if F(i)==D(i);
      total=total+1
   else
      total=total+1
      number=number+1
   end
   ratio=number/total
```

```
end
```

APPENDIX D

Code for a 4x4 parallel-implemented full adder multiplier module

This is the code for a 4x4 parallel-implemented full adder multiplier module. This takes the values of the inputs from the stimulus and multiplies them together using a parallel full adder design. The inputs are **A** and **B** and the output is ***y***.

CODE:

```
 //timescale for the simulation
`timescale 1ns / 100ps
//the Boeing library needed for the adders and the and gates
`include "setlibraryfile.v"


//module setup
//beginning of module
module multi_parallel_4x4 (A, B, C, cin, clk, pre, clr, y);

//inputs
        input cin, clk, clr, pre, C;
        input [3:0] A, B;
//output
        output [7:0] y;


//regs

reg aa1, aa2, aa3, aa4, aa5, aa6, aa7, aa8, aa9, aa10, aa11, aa12, aa13, aa14, aa15,
aa16, aa17, aa18, aa19, aa20, aa21, aa22, aa23, aa24, aa25, aa26, aa27, aa28,
aa29, aa30, aa31, aa32, aa39, aa41, aa42, aa43, aa44, aa45, aa46, aa47, aa48,
aa49, aa50, aa51, aa52, aa53, aa54, aa55, aa56, aa57, aa58, aa59, aa60, aa61,
aa62, aa63, aa64, aa65, aa66, aa67, aa68, aa69, aa70, aa71, aa72, aa73, aa74,
aa75, aa76;
```

```
//wires

        wire a0_buf, a1_buf, a2_buf, a3_buf, a4_buf, a5_buf, a6_buf, a7_buf,
a8_buf, a9_buf, a10_buf, a11_buf, a12_buf, a13_buf, a14_buf, a15_buf, a16_buf,
a17_buf, a18_buf, a19_buf, a20_buf, a21_buf, a22_buf, a23_buf, a24_buf,
a25_buf, a26_buf, a27_buf, a28_buf, a29_buf, a30_buf, a31_buf, a32_buf,
a33_buf, a34_buf, a35_buf, a36_buf, a37_buf, a38_buf, a39_buf, a40_buf,
a41_buf, a42_buf, a43_buf, a44_buf, a45_buf, a46_buf, a47_buf, a48_buf,
a49_buf, a50_buf, a51_buf, a52_buf, a53_buf, a54_buf, a55_buf, a56_buf,
a57_buf, a58_buf, a59_buf, a60_buf, a61_buf, a62_buf, a63_buf, a64_buf,
a65_buf, a66_buf, a67_buf, a68_buf, a69_buf, a70_buf, a71_buf, a72_buf,
a73_buf, a74_buf, a75_buf, a76_buf, a77_buf, a78_buf, a79_buf, a80_buf,
a81_buf, a82_buf, a83_buf, a84_buf, a85_buf, a86_buf, a87_buf, a88_buf,
a89_buf, a90_buf, a91_buf, a92_buf, a93_buf, a94_buf, a95_buf, a96_buf;



//Multiplier


//***First stage***


//input A for DFF:  cells (a1-a4) , wires (a1-a4)

        dff_cp_1x al( .clr(clr), .d(A[0]), .clk(clk), .pre(pre), .q(a1_buf));
        dff_cp_1x a2( .clr(clr), .d(A[1]), .clk(clk), .pre(pre), .q(a2_buf));
        dff_cp_1x a3( .clr(clr), .d(A[2]), .clk(clk), .pre(pre), .q(a3_buf));
        dff_cp_1x a4( .clr(clr), .d(A[3]), .clk(clk), .pre(pre), .q(a4_buf));




//input B for DFF:  cells (a5-a8) , wires (a5-a8)

        dff_cp_1x a5( .clr(clr), .d(B[0]), .clk(clk), .pre(pre), .q(a5_buf));
        dff_cp_1x a6( .clr(clr), .d(B[1]), .clk(clk), .pre(pre), .q(a6_buf));
        dff_cp_1x a7( .clr(clr), .d(B[2]), .clk(clk), .pre(pre), .q(a7_buf));
        dff_cp_1x a8( .clr(clr), .d(B[3]), .clk(clk), .pre(pre), .q(a8_buf));




//and first stage: cells (a9-a12), wires (a9-a12)

        and2_1x a9( .in1(a5_buf), .in0(a1_buf), .y(a9_buf));
```

```
        and2_1x a10( .in1(a5_buf), .in0(a2_buf), .y(a10_buf));
        and2_1x a11( .in1(a5_buf), .in0(a3_buf), .y(a11_buf));
        and2_1x a12( .in1(a5_buf), .in0(a4_buf), .y(a12_buf));



//Results register:  cells(a13-a20, wires (a13-a20)

        dff_cp_1x a13( .clr(clr), .d(a9_buf), .clk(clk), .pre(pre), .q(a13_buf));
        dff_cp_1x a14( .clr(clr), .d(a10_buf), .clk(clk), .pre(pre), .q(a14_buf));
        dff_cp_1x a15( .clr(clr), .d(a11_buf), .clk(clk), .pre(pre), .q(a15_buf));
        dff_cp_1x a16( .clr(clr), .d(a12_buf), .clk(clk), .pre(pre), .q(a16_buf));
        dff_cp_1x a17( .clr(clr), .d(C), .clk(clk), .pre(pre), .q(a17_buf));
        dff_cp_1x a18( .clr(clr), .d(C), .clk(clk), .pre(pre), .q(a18_buf));
        dff_cp_1x a19( .clr(clr), .d(C), .clk(clk), .pre(pre), .q(a19_buf));
        dff_cp_1x a20( .clr(clr), .d(C), .clk(clk), .pre(pre), .q(a20_buf));


//Test register


//      dff_cp_1x a13( .clr(clr), .d(a9_buf), .clk(clk), .pre(pre), .q(y[0]));
//      dff_cp_1x a14( .clr(clr), .d(a10_buf), .clk(clk), .pre(pre), .q(y[1]));
//      dff_cp_1x a15( .clr(clr), .d(a11_buf), .clk(clk), .pre(pre), .q(y[2]));
//      dff_cp_1x a16( .clr(clr), .d(a12_buf), .clk(clk), .pre(pre), .q(y[3]));
//      dff_cp_1x a17( .clr(clr), .d(C), .clk(clk), .pre(pre), .q(y[4]));
//      dff_cp_1x a18( .clr(clr), .d(C), .clk(clk), .pre(pre), .q(y[5]));
//      dff_cp_1x a19( .clr(clr), .d(C), .clk(clk), .pre(pre), .q(y[6]));
//      dff_cp_1x a20( .clr(clr), .d(C), .clk(clk), .pre(pre), .q(y[7]));
//clock in the outputs
always @(negedge clk)
        begin
                aa13 <= a13_buf;
                aa14 <= a14_buf;
                aa15 <= a15_buf;
                aa16 <= a16_buf;
                aa17 <= a17_buf;
                aa18 <= a18_buf;
                aa19 <= a19_buf;
                aa20 <= a20_buf;

        end

//***Second stage***
```

```
//clock in the inputs
always @(negedge clk)
        begin
                aa1 <= a1_buf;
                aa2 <= a2_buf;
                aa3 <= a3_buf;
                aa4 <= a4_buf;

        end



//input A for DFF:  cells (a21-a24) , wires (a21-a24)

        dff_cp_1x a21( .clr(clr), .d(aa1), .clk(clk), .pre(pre), .q(a21_buf));
        dff_cp_1x a22( .clr(clr), .d(aa2), .clk(clk), .pre(pre), .q(a22_buf));
        dff_cp_1x a23( .clr(clr), .d(aa3), .clk(clk), .pre(pre), .q(a23_buf));
        dff_cp_1x a24( .clr(clr), .d(aa4), .clk(clk), .pre(pre), .q(a24_buf));

//clock the inputs
always @(negedge clk)
        begin
                aa5 <= a5_buf;
                aa6 <= a6_buf;
                aa7 <= a7_buf;
                aa8 <= a8_buf;

        end
//input B for DFF:  cells (a25-a28) , wires (a25-a28)

        dff_cp_1x a25( .clr(clr), .d(aa5), .clk(clk), .pre(pre), .q(a25_buf));
        dff_cp_1x a26( .clr(clr), .d(aa6), .clk(clk), .pre(pre), .q(a26_buf));
        dff_cp_1x a27( .clr(clr), .d(aa7), .clk(clk), .pre(pre), .q(a27_buf));
        dff_cp_1x a28( .clr(clr), .d(aa8), .clk(clk), .pre(pre), .q(a28_buf));



//and second stage: cells(a29-a32), wires(a29-a32)

        and2_1x a29( .in1(a26_buf), .in0(a21_buf), .y(a29_buf));
        and2_1x a30( .in1(a26_buf), .in0(a22_buf), .y(a30_buf));
        and2_1x a31( .in1(a26_buf), .in0(a23_buf), .y(a31_buf));
        and2_1x a32( .in1(a26_buf), .in0(a24_buf), .y(a32_buf));
```

```
//add to previous stage using Full Adder: cells (a33-a36), wires (a33-a40)

	add_1x1x a33( .a(aa14), .b(a29_buf), .cin(C), .cout(a33_buf),
.y(a34_buf));
	add_1x1x a34( .a(aa15), .b(a30_buf), .cin(a33_buf), .cout(a35_buf),
.y(a36_buf));
	add_1x1x a35( .a(aa16), .b(a31_buf), .cin(a35_buf), .cout(a37_buf),
.y(a38_buf));
	add_1x1x a36( .a(C), .b(a32_buf), .cin(a37_buf), .cout(a39_buf),
.y(a40_buf));

//take the last carry bit to the next stage
always @(posedge clk)
	begin
		aa39 <= a39_buf;
	end



//Results register: cells(a37-a44), wires (a41-a48)

	dff_cp_1x a37( .clr(clr), .d(aa13), .clk(clk), .pre(pre), .q(a41_buf));
	dff_cp_1x a38( .clr(clr), .d(a34_buf), .clk(clk), .pre(pre), .q(a42_buf));
	dff_cp_1x a39( .clr(clr), .d(a36_buf), .clk(clk), .pre(pre), .q(a43_buf));
	dff_cp_1x a40( .clr(clr), .d(a38_buf), .clk(clk), .pre(pre), .q(a44_buf));
	dff_cp_1x a41( .clr(clr), .d(a40_buf), .clk(clk), .pre(pre), .q(a45_buf));
	dff_cp_1x a42( .clr(clr), .d(aa18), .clk(clk), .pre(pre), .q(a46_buf));
	dff_cp_1x a43( .clr(clr), .d(aa19), .clk(clk), .pre(pre), .q(a47_buf));
	dff_cp_1x a44( .clr(clr), .d(aa20), .clk(clk), .pre(pre), .q(a48_buf));


//Test register


//	dff_cp_1x a37( .clr(clr), .d(aa13), .clk(clk), .pre(pre), .q(y[0]));
//	dff_cp_1x a38( .clr(clr), .d(a34_buf), .clk(clk), .pre(pre), .q(y[1]));
//	dff_cp_1x a39( .clr(clr), .d(a36_buf), .clk(clk), .pre(pre), .q(y[2]));
//	dff_cp_1x a40( .clr(clr), .d(a38_buf), .clk(clk), .pre(pre), .q(y[3]));
//	dff_cp_1x a41( .clr(clr), .d(a40_buf), .clk(clk), .pre(pre), .q(y[4]));
//	dff_cp_1x a42( .clr(clr), .d(aa18), .clk(clk), .pre(pre), .q(y[5]));
//	dff_cp_1x a43( .clr(clr), .d(aa19), .clk(clk), .pre(pre), .q(y[6]));
//	dff_cp_1x a44( .clr(clr), .d(aa20), .clk(clk), .pre(pre), .q(y[7]));

//clock the outputs
always @(negedge clk)
```

```
        begin
                aa41 <= a41_buf;
                aa42 <= a42_buf;
                aa43 <= a43_buf;
                aa44 <= a44_buf;
                aa45 <= a45_buf;
                aa46 <= a46_buf;
                aa47 <= a47_buf;
                aa48 <= a48_buf;

        end



//***Third Stage***

//clock the inputs
always @(negedge clk)
        begin
                aa21 <= a21_buf;
                aa22 <= a22_buf;
                aa23 <= a23_buf;
                aa24 <= a24_buf;

        end

//input A for DFF:  cells (a45-a48) , wires (a49-a52)

        dff_cp_1x a45( .clr(clr), .d(aa21), .clk(clk), .pre(pre), .q(a49_buf));
        dff_cp_1x a46( .clr(clr), .d(aa22), .clk(clk), .pre(pre), .q(a50_buf));
        dff_cp_1x a47( .clr(clr), .d(aa23), .clk(clk), .pre(pre), .q(a51_buf));
        dff_cp_1x a48( .clr(clr), .d(aa24), .clk(clk), .pre(pre), .q(a52_buf));

//clock the inputs
always @(negedge clk)
        begin
                aa25 <= a25_buf;
                aa26 <= a26_buf;
                aa27 <= a27_buf;
                aa28 <= a28_buf;
        end
//input B for DFF:  cells (a49-a52) , wires (a53-a56)

        dff_cp_1x a49( .clr(clr), .d(aa25), .clk(clk), .pre(pre), .q(a53_buf));
        dff_cp_1x a50( .clr(clr), .d(aa26), .clk(clk), .pre(pre), .q(a54_buf));
        dff_cp_1x a51( .clr(clr), .d(aa27), .clk(clk), .pre(pre), .q(a55_buf));
```

```
        dff_cp_1x a52( .clr(clr), .d(aa28), .clk(clk), .pre(pre), .q(a56_buf));



//and third stage: cells(a53-a56), wires(a57-a60)

        and2_1x a53( .in1(a55_buf), .in0(a49_buf), .y(a57_buf));
        and2_1x a54( .in1(a55_buf), .in0(a50_buf), .y(a58_buf));
        and2_1x a55( .in1(a55_buf), .in0(a51_buf), .y(a59_buf));
        and2_1x a56( .in1(a55_buf), .in0(a52_buf), .y(a60_buf));



//add to previous stage using Full Adder: cells (a57-a60), wires (a61-a68)

        add_1x1x a57( .a(aa43), .b(a57_buf), .cin(C), .cout(a61_buf),
.y(a62_buf));
        add_1x1x a58( .a(aa44), .b(a58_buf), .cin(a61_buf), .cout(a63_buf),
.y(a64_buf));
        add_1x1x a59( .a(aa45), .b(a59_buf), .cin(a63_buf), .cout(a65_buf),
.y(a66_buf));
        add_1x1x a60( .a(aa39), .b(a60_buf), .cin(a65_buf), .cout(a67_buf),
.y(a68_buf));
//take the last carry to the next stage
always @(posedge clk)
        begin
                aa67 <= a67_buf;
        end

//Results register: cells(a61-a68), wires (a69-a76)

        dff_cp_1x a61( .clr(clr), .d(aa41), .clk(clk), .pre(pre), .q(a69_buf));
        dff_cp_1x a62( .clr(clr), .d(aa42), .clk(clk), .pre(pre), .q(a70_buf));
        dff_cp_1x a63( .clr(clr), .d(a62_buf), .clk(clk), .pre(pre), .q(a71_buf));
        dff_cp_1x a64( .clr(clr), .d(a64_buf), .clk(clk), .pre(pre), .q(a72_buf));
        dff_cp_1x a65( .clr(clr), .d(a66_buf), .clk(clk), .pre(pre), .q(a73_buf));
        dff_cp_1x a66( .clr(clr), .d(a68_buf), .clk(clk), .pre(pre), .q(a74_buf));
        dff_cp_1x a67( .clr(clr), .d(aa47), .clk(clk), .pre(pre), .q(a75_buf));
        dff_cp_1x a68( .clr(clr), .d(aa48), .clk(clk), .pre(pre), .q(a76_buf));


//Test register
```

```
//        dff_cp_1x a61( .clr(clr), .d(aa41), .clk(clk), .pre(pre), .q(y[0]));
//        dff_cp_1x a62( .clr(clr), .d(aa42), .clk(clk), .pre(pre), .q(y[1]));
//        dff_cp_1x a63( .clr(clr), .d(a62_buf), .clk(clk), .pre(pre), .q(y[2]));
//        dff_cp_1x a64( .clr(clr), .d(a64_buf), .clk(clk), .pre(pre), .q(y[3]));
//        dff_cp_1x a65( .clr(clr), .d(a66_buf), .clk(clk), .pre(pre), .q(y[4]));
//        dff_cp_1x a66( .clr(clr), .d(a68_buf), .clk(clk), .pre(pre), .q(y[5]));
//        dff_cp_1x a67( .clr(clr), .d(aa47), .clk(clk), .pre(pre), .q(y[6]));
//        dff_cp_1x a68( .clr(clr), .d(aa48), .clk(clk), .pre(pre), .q(y[7]));
//clock the outputs
always @(negedge clk)
        begin
                aa69 <= a69_buf;
                aa70 <= a70_buf;
                aa71 <= a71_buf;
                aa72 <= a72_buf;
                aa73 <= a73_buf;
                aa74 <= a74_buf;
                aa75 <= a75_buf;
                aa76 <= a76_buf;

        end



//***Fourth Stage***

//clock the inputs
always @(negedge clk)
        begin
                aa49 <= a49_buf;
                aa50 <= a50_buf;
                aa51 <= a51_buf;
                aa52 <= a52_buf;

        end
//input A for DFF:  cells (a69-a72) , wires (a77-a80)
        dff_cp_1x a69( .clr(clr), .d(aa49), .clk(clk), .pre(pre), .q(a77_buf));
        dff_cp_1x a70( .clr(clr), .d(aa50), .clk(clk), .pre(pre), .q(a78_buf));
        dff_cp_1x a71( .clr(clr), .d(aa51), .clk(clk), .pre(pre), .q(a79_buf));
        dff_cp_1x a72( .clr(clr), .d(aa52), .clk(clk), .pre(pre), .q(a80_buf));

//clock the inputs

always @(negedge clk)
        begin
                aa53 <= a53_buf;
```

```
                aa54 <= a54_buf;
                aa55 <= a55_buf;
                aa56 <= a56_buf;

        end
//input B for DFF:  cells (a73-a76) , wires (a81-a84)
        dff_cp_1x a73( .clr(clr), .d(aa53), .clk(clk), .pre(pre), .q(a81_buf));
        dff_cp_1x a74( .clr(clr), .d(aa54), .clk(clk), .pre(pre), .q(a82_buf));
        dff_cp_1x a75( .clr(clr), .d(aa55), .clk(clk), .pre(pre), .q(a83_buf));
        dff_cp_1x a76( .clr(clr), .d(aa56), .clk(clk), .pre(pre), .q(a84_buf));



//and fourth stage: cells(a77-a80), wires(a85-a88)

        and2_1x a77( .in1(a84_buf), .in0(a77_buf), .y(a85_buf));
        and2_1x a78( .in1(a84_buf), .in0(a78_buf), .y(a86_buf));
        and2_1x a79( .in1(a84_buf), .in0(a79_buf), .y(a87_buf));
        and2_1x a80( .in1(a84_buf), .in0(a80_buf), .y(a88_buf));



//add to previous stage using Full Adder: cells (a81-a84), wires (a89-a96)

        add_1x1x a81( .a(aa72), .b(a85_buf), .cin(C), .cout(a89_buf),
.y(a90_buf));
        add_1x1x a82( .a(aa73), .b(a86_buf), .cin(a89_buf), .cout(a91_buf),
.y(a92_buf));
        add_1x1x a83( .a(aa74), .b(a87_buf), .cin(a91_buf), .cout(a93_buf),
.y(a94_buf));
        add_1x1x a84( .a(aa67), .b(a88_buf), .cin(a93_buf), .cout(a95_buf),
.y(a96_buf));



//Results register: cells(a85-a92)


        dff_cp_1x a85( .clr(clr), .d(aa69), .clk(clk), .pre(pre), .q(y[0]));
        dff_cp_1x a86( .clr(clr), .d(aa70), .clk(clk), .pre(pre), .q(y[1]));
        dff_cp_1x a87( .clr(clr), .d(aa71), .clk(clk), .pre(pre), .q(y[2]));
        dff_cp_1x a88( .clr(clr), .d(a90_buf), .clk(clk), .pre(pre), .q(y[3]));
        dff_cp_1x a89( .clr(clr), .d(a92_buf), .clk(clk), .pre(pre), .q(y[4]));
        dff_cp_1x a90( .clr(clr), .d(a94_buf), .clk(clk), .pre(pre), .q(y[5]));
        dff_cp_1x a91( .clr(clr), .d(a96_buf), .clk(clk), .pre(pre), .q(y[6]));
        dff_cp_1x a92( .clr(clr), .d(a95_buf), .clk(clk), .pre(pre), .q(y[7]));
```

```
endmodule
```

APPENDIX E

Code for a 4x4 SET parallel-implemented full adder multiplier module

This is the code for a 4x4 SET parallel-implemented full adder multiplier module. This module takes the inputs, **A** and **B**, from the stimulus and multiplies them through parallel full adders. This also has two more inputs, **SETH** and **SETL**. The SET inputs are used to drive the output node on the AND gates and the carry and output nodes on the full adders to either low or high. The multiplier has one output, ***p***.

CODE:

```
//timescale for the simulations
`timescale 1ns / 100ps
//the Boeing libraries needed for the and gates, the dffs and the full adders
`include "setlibraryfile.v"
`include "setlibrary_SETH.v"

//module setup
//The beginning of the module

module multi_parallel_4x4_SET (A, B, C, cin, clk, pre, clr, p, SETH, SETL);
//inputs
        input cin, clk, clr, pre, C;
        input [31:0] SETH, SETL;
        input [3:0] A, B;
//output
        output [7:0] p;
```

//regs

reg aa1, aa2, aa3, aa4, aa5, aa6, aa7, aa8, aa9, aa10, aa11, aa12, aa13, aa14, aa15, aa16, aa17, aa18, aa19, aa20, aa21, aa22, aa23, aa24, aa25, aa26, aa27, aa28, aa29, aa30, aa31, aa32, aa39, aa41, aa42, aa43, aa44, aa45, aa46, aa47, aa48, aa49, aa50, aa51, aa52, aa53, aa54, aa55, aa56, aa57, aa58, aa59, aa60, aa61, aa62, aa63, aa64, aa65, aa66, aa67, aa68, aa69, aa70, aa71, aa72, aa73, aa74, aa75, aa76;

//wires

wire a0_buf, a1_buf, a2_buf, a3_buf, a4_buf, a5_buf, a6_buf, a7_buf, a8_buf, a9_buf, a10_buf, a11_buf, a12_buf, a13_buf, a14_buf, a15_buf, a16_buf, a17_buf, a18_buf, a19_buf, a20_buf, a21_buf, a22_buf, a23_buf, a24_buf, a25_buf, a26_buf, a27_buf, a28_buf, a29_buf, a30_buf, a31_buf, a32_buf, a33_buf, a34_buf, a35_buf, a36_buf, a37_buf, a38_buf, a39_buf, a40_buf, a41_buf, a42_buf, a43_buf, a44_buf, a45_buf, a46_buf, a47_buf, a48_buf, a49_buf, a50_buf, a51_buf, a52_buf, a53_buf, a54_buf, a55_buf, a56_buf, a57_buf, a58_buf, a59_buf, a60_buf, a61_buf, a62_buf, a63_buf, a64_buf, a65_buf, a66_buf, a67_buf, a68_buf, a69_buf, a70_buf, a71_buf, a72_buf, a73_buf, a74_buf, a75_buf, a76_buf, a77_buf, a78_buf, a79_buf, a80_buf, a81_buf, a82_buf, a83_buf, a84_buf, a85_buf, a86_buf, a87_buf, a88_buf, a89_buf, a90_buf, a91_buf, a92_buf, a93_buf, a94_buf, a95_buf, a96_buf;

//Multiplier

//***First stage***

```
//input A for DFF:  cells (a1-a4) , wires (a1-a4)

        dff_cp_1x al( .clr(clr), .d(A[0]), .clk(clk), .pre(pre), .q(a1_buf));
        dff_cp_1x a2( .clr(clr), .d(A[1]), .clk(clk), .pre(pre), .q(a2_buf));
        dff_cp_1x a3( .clr(clr), .d(A[2]), .clk(clk), .pre(pre), .q(a3_buf));
        dff_cp_1x a4( .clr(clr), .d(A[3]), .clk(clk), .pre(pre), .q(a4_buf));



//input B for DFF:  cells (a5-a8) , wires (a5-a8)

        dff_cp_1x a5( .clr(clr), .d(B[0]), .clk(clk), .pre(pre), .q(a5_buf));
        dff_cp_1x a6( .clr(clr), .d(B[1]), .clk(clk), .pre(pre), .q(a6_buf));
        dff_cp_1x a7( .clr(clr), .d(B[2]), .clk(clk), .pre(pre), .q(a7_buf));
        dff_cp_1x a8( .clr(clr), .d(B[3]), .clk(clk), .pre(pre), .q(a8_buf));

//and first stage: cells (a9-a12), wires (a9-a12)

        and2_1x_SETH a9( .in1(a5_buf), .in0(a1_buf), .clk(clk), .y(a9_buf),
.SETH(SETH), .SETL(SETL), .ADDR(1));
        and2_1x_SETH a10( .in1(a5_buf), .in0(a2_buf), .clk(clk), .y(a10_buf),
.SETH(SETH), .SETL(SETL), .ADDR(2));
        and2_1x_SETH a11( .in1(a5_buf), .in0(a3_buf), .clk(clk), .y(a11_buf),
.SETH(SETH), .SETL(SETL), .ADDR(3));
        and2_1x_SETH a12( .in1(a5_buf), .in0(a4_buf), .clk(clk), .y(a12_buf),
.SETH(SETH), .SETL(SETL), .ADDR(4));
```

```
//Results register:  cells(a13-a20), wires (a13-a20)

        dff_cp_1x a13( .clr(clr), .d(a9_buf), .clk(clk), .pre(pre), .q(a13_buf));
        dff_cp_1x a14( .clr(clr), .d(a10_buf), .clk(clk), .pre(pre), .q(a14_buf));
        dff_cp_1x a15( .clr(clr), .d(a11_buf), .clk(clk), .pre(pre), .q(a15_buf));
        dff_cp_1x a16( .clr(clr), .d(a12_buf), .clk(clk), .pre(pre), .q(a16_buf));
        dff_cp_1x a17( .clr(clr), .d(C), .clk(clk), .pre(pre), .q(a17_buf));
        dff_cp_1x a18( .clr(clr), .d(C), .clk(clk), .pre(pre), .q(a18_buf));
        dff_cp_1x a19( .clr(clr), .d(C), .clk(clk), .pre(pre), .q(a19_buf));
        dff_cp_1x a20( .clr(clr), .d(C), .clk(clk), .pre(pre), .q(a20_buf));


//Test register


//      dff_cp_1x a13( .clr(clr), .d(a9_buf), .clk(clk), .pre(pre), .q(y[0]));
//      dff_cp_1x a14( .clr(clr), .d(a10_buf), .clk(clk), .pre(pre), .q(y[1]));
//      dff_cp_1x a15( .clr(clr), .d(a11_buf), .clk(clk), .pre(pre), .q(y[2]));
//      dff_cp_1x a16( .clr(clr), .d(a12_buf), .clk(clk), .pre(pre), .q(y[3]));
//      dff_cp_1x a17( .clr(clr), .d(C), .clk(clk), .pre(pre), .q(y[4]));
//      dff_cp_1x a18( .clr(clr), .d(C), .clk(clk), .pre(pre), .q(y[5]));
//      dff_cp_1x a19( .clr(clr), .d(C), .clk(clk), .pre(pre), .q(y[6]));
//      dff_cp_1x a20( .clr(clr), .d(C), .clk(clk), .pre(pre), .q(y[7]));
//clock the outputs
```

```
always @(negedge clk)
        begin
                aa13 <= a13_buf;
                aa14 <= a14_buf;
                aa15 <= a15_buf;
                aa16 <= a16_buf;
                aa17 <= a17_buf;
                aa18 <= a18_buf;
                aa19 <= a19_buf;
                aa20 <= a20_buf;

        end

//***Second stage***
//clock the inputs
always @(negedge clk)
        begin
                aa1 <= a1_buf;
                aa2 <= a2_buf;
                aa3 <= a3_buf;
                aa4 <= a4_buf;

        end



//input A for DFF:  cells (a21-a24) , wires (a21-a24)
```

```
        dff_cp_1x a21( .clr(clr), .d(aa1), .clk(clk), .pre(pre), .q(a21_buf));
        dff_cp_1x a22( .clr(clr), .d(aa2), .clk(clk), .pre(pre), .q(a22_buf));
        dff_cp_1x a23( .clr(clr), .d(aa3), .clk(clk), .pre(pre), .q(a23_buf));
        dff_cp_1x a24( .clr(clr), .d(aa4), .clk(clk), .pre(pre), .q(a24_buf));

//clock the inputs
always @(negedge clk)
        begin
                aa5 <= a5_buf;
                aa6 <= a6_buf;
                aa7 <= a7_buf;
                aa8 <= a8_buf;

        end
//input B for DFF:  cells (a25-a28) , wires (a25-a28)

        dff_cp_1x a25( .clr(clr), .d(aa5), .clk(clk), .pre(pre), .q(a25_buf));
        dff_cp_1x a26( .clr(clr), .d(aa6), .clk(clk), .pre(pre), .q(a26_buf));
        dff_cp_1x a27( .clr(clr), .d(aa7), .clk(clk), .pre(pre), .q(a27_buf));
        dff_cp_1x a28( .clr(clr), .d(aa8), .clk(clk), .pre(pre), .q(a28_buf));



//and second stage: cells(a29-a32), wires(a29-a32)
```

```
        and2_1x_SETH a29( .in1(a26_buf), .in0(a21_buf), .clk(clk), .y(a29_buf),
.SETH(SETH), .SETL(SETL), .ADDR(5));

        and2_1x_SETH a30( .in1(a26_buf), .in0(a22_buf), .clk(clk), .y(a30_buf),
.SETH(SETH), .SETL(SETL), .ADDR(6));

        and2_1x_SETH a31( .in1(a26_buf), .in0(a23_buf), .clk(clk), .y(a31_buf),
.SETH(SETH), .SETL(SETL), .ADDR(7));

        and2_1x_SETH a32( .in1(a26_buf), .in0(a24_buf), .clk(clk), .y(a32_buf),
.SETH(SETH), .SETL(SETL), .ADDR(8));




//add to previous stage using Full Adder: cells (a33-a36), wires (a33-a40)



        add_1x1x_SETH a33( .a(aa14), .b(a29_buf), .cin(C), .clk(clk), .cout(a33_buf),
.y(a34_buf), .SETH(SETH), .SETL(SETL), .ADDR_COUT(9), .ADDR_Y(10));

        add_1x1x_SETH a34( .a(aa15), .b(a30_buf), .cin(a33_buf), .clk(clk),
.cout(a35_buf), .y(a36_buf), .SETH(SETH), .SETL(SETL), .ADDR_COUT(11),
.ADDR_Y(12));

        add_1x1x_SETH a35( .a(aa16), .b(a31_buf), .cin(a35_buf), .clk(clk),
.cout(a37_buf), .y(a38_buf), .SETH(SETH), .SETL(SETL), .ADDR_COUT(13),
.ADDR_Y(14));

        add_1x1x_SETH a36( .a(C), .b(a32_buf), .cin(a37_buf), .clk(clk), .cout(a39_buf),
.y(a40_buf), .SETH(SETH), .SETL(SETL), .ADDR_COUT(15), .ADDR_Y(16));



//take the last carry out to the next stage

always @(posedge clk)

        begin

                aa39 <= a39_buf;

        end
```

```
//Results register: cells(a37-a44), wires (a41-a48)

        dff_cp_1x a37( .clr(clr), .d(aa13), .clk(clk), .pre(pre), .q(a41_buf));
        dff_cp_1x a38( .clr(clr), .d(a34_buf), .clk(clk), .pre(pre), .q(a42_buf));
        dff_cp_1x a39( .clr(clr), .d(a36_buf), .clk(clk), .pre(pre), .q(a43_buf));
        dff_cp_1x a40( .clr(clr), .d(a38_buf), .clk(clk), .pre(pre), .q(a44_buf));
        dff_cp_1x a41( .clr(clr), .d(a40_buf), .clk(clk), .pre(pre), .q(a45_buf));
        dff_cp_1x a42( .clr(clr), .d(aa18), .clk(clk), .pre(pre), .q(a46_buf));
        dff_cp_1x a43( .clr(clr), .d(aa19), .clk(clk), .pre(pre), .q(a47_buf));
        dff_cp_1x a44( .clr(clr), .d(aa20), .clk(clk), .pre(pre), .q(a48_buf));


//Test register


//      dff_cp_1x a37( .clr(clr), .d(aa13), .clk(clk), .pre(pre), .q(y[0]));
//      dff_cp_1x a38( .clr(clr), .d(a34_buf), .clk(clk), .pre(pre), .q(y[1]));
//      dff_cp_1x a39( .clr(clr), .d(a36_buf), .clk(clk), .pre(pre), .q(y[2]));
//      dff_cp_1x a40( .clr(clr), .d(a38_buf), .clk(clk), .pre(pre), .q(y[3]));
//      dff_cp_1x a41( .clr(clr), .d(a40_buf), .clk(clk), .pre(pre), .q(y[4]));
//      dff_cp_1x a42( .clr(clr), .d(aa18), .clk(clk), .pre(pre), .q(y[5]));
//      dff_cp_1x a43( .clr(clr), .d(aa19), .clk(clk), .pre(pre), .q(y[6]));
//      dff_cp_1x a44( .clr(clr), .d(aa20), .clk(clk), .pre(pre), .q(y[7]));

//clock the outputs
always @(negedge clk)
```

```
        begin
                aa41 <= a41_buf;
                aa42 <= a42_buf;
                aa43 <= a43_buf;
                aa44 <= a44_buf;
                aa45 <= a45_buf;
                aa46 <= a46_buf;
                aa47 <= a47_buf;
                aa48 <= a48_buf;

        end



//***Third Stage***

//clock the inputs
always @(negedge clk)
        begin
                aa21 <= a21_buf;
                aa22 <= a22_buf;
                aa23 <= a23_buf;
                aa24 <= a24_buf;

        end

//input A for DFF:  cells (a45-a48) , wires (a49-a52)
```

```
        dff_cp_1x a45( .clr(clr), .d(aa21), .clk(clk), .pre(pre), .q(a49_buf));
        dff_cp_1x a46( .clr(clr), .d(aa22), .clk(clk), .pre(pre), .q(a50_buf));
        dff_cp_1x a47( .clr(clr), .d(aa23), .clk(clk), .pre(pre), .q(a51_buf));
        dff_cp_1x a48( .clr(clr), .d(aa24), .clk(clk), .pre(pre), .q(a52_buf));


//clock the inputs
always @(negedge clk)
        begin
                aa25 <= a25_buf;
                aa26 <= a26_buf;
                aa27 <= a27_buf;
                aa28 <= a28_buf;
        end
//input B for DFF:  cells (a49-a52) , wires (a53-a56)
        dff_cp_1x a49( .clr(clr), .d(aa25), .clk(clk), .pre(pre), .q(a53_buf));
        dff_cp_1x a50( .clr(clr), .d(aa26), .clk(clk), .pre(pre), .q(a54_buf));
        dff_cp_1x a51( .clr(clr), .d(aa27), .clk(clk), .pre(pre), .q(a55_buf));
        dff_cp_1x a52( .clr(clr), .d(aa28), .clk(clk), .pre(pre), .q(a56_buf));


//and third stage: cells(a53-a56), wires(a57-a60)


        and2_1x_SETH a53( .in1(a55_buf), .in0(a49_buf), .clk(clk), .y(a57_buf),
.SETH(SETH), .SETL(SETL), .ADDR(17));
        and2_1x_SETH a54( .in1(a55_buf), .in0(a50_buf), .clk(clk), .y(a58_buf),
.SETH(SETH), .SETL(SETL), .ADDR(18));
        and2_1x_SETH a55( .in1(a55_buf), .in0(a51_buf), .clk(clk), .y(a59_buf),
.SETH(SETH), .SETL(SETL), .ADDR(19));
```

```
        and2_1x_SETH a56( .in1(a55_buf), .in0(a52_buf), .clk(clk), .y(a60_buf),
.SETH(SETH), .SETL(SETL), .ADDR(20));


//add to previous stage using Full Adder: cells (a57-a60), wires (a61-a68)


        add_1x1x_SETH a57( .a(aa43), .b(a57_buf), .cin(C), .clk(clk), .cout(a61_buf),
.y(a62_buf), .SETH(SETH), .SETL(SETL), .ADDR_COUT(21), .ADDR_Y(22));

        add_1x1x_SETH a58( .a(aa44), .b(a58_buf), .cin(a61_buf), .clk(clk),
.cout(a63_buf), .y(a64_buf), .SETH(SETH), .SETL(SETL), .ADDR_COUT(23),
.ADDR_Y(24));

        add_1x1x_SETH a59( .a(aa45), .b(a59_buf), .cin(a63_buf), .clk(clk),
.cout(a65_buf), .y(a66_buf), .SETH(SETH), .SETL(SETL), .ADDR_COUT(25),
.ADDR_Y(26));

        add_1x1x_SETH a60( .a(aa39), .b(a60_buf), .cin(a65_buf), .clk(clk),
.cout(a67_buf), .y(a68_buf), .SETH(SETH), .SETL(SETL), .ADDR_COUT(27),
.ADDR_Y(28));

//take the last carry out to the next stage

always @(posedge clk)

        begin

                aa67 <= a67_buf;

        end


//Results register: cells(a61-a68), wires (a69-a76)


        dff_cp_1x a61( .clr(clr), .d(aa41), .clk(clk), .pre(pre), .q(a69_buf));

        dff_cp_1x a62( .clr(clr), .d(aa42), .clk(clk), .pre(pre), .q(a70_buf));

        dff_cp_1x a63( .clr(clr), .d(a62_buf), .clk(clk), .pre(pre), .q(a71_buf));

        dff_cp_1x a64( .clr(clr), .d(a64_buf), .clk(clk), .pre(pre), .q(a72_buf));

        dff_cp_1x a65( .clr(clr), .d(a66_buf), .clk(clk), .pre(pre), .q(a73_buf));

        dff_cp_1x a66( .clr(clr), .d(a68_buf), .clk(clk), .pre(pre), .q(a74_buf));

        dff_cp_1x a67( .clr(clr), .d(aa47), .clk(clk), .pre(pre), .q(a75_buf));
```

```
        dff_cp_1x a68( .clr(clr), .d(aa48), .clk(clk), .pre(pre), .q(a76_buf));

//Test register

//      dff_cp_1x a61( .clr(clr), .d(aa41), .clk(clk), .pre(pre), .q(y[0]));
//      dff_cp_1x a62( .clr(clr), .d(aa42), .clk(clk), .pre(pre), .q(y[1]));
//      dff_cp_1x a63( .clr(clr), .d(a62_buf), .clk(clk), .pre(pre), .q(y[2]));
//      dff_cp_1x a64( .clr(clr), .d(a64_buf), .clk(clk), .pre(pre), .q(y[3]));
//      dff_cp_1x a65( .clr(clr), .d(a66_buf), .clk(clk), .pre(pre), .q(y[4]));
//      dff_cp_1x a66( .clr(clr), .d(a68_buf), .clk(clk), .pre(pre), .q(y[5]));
//      dff_cp_1x a67( .clr(clr), .d(aa47), .clk(clk), .pre(pre), .q(y[6]));
//      dff_cp_1x a68( .clr(clr), .d(aa48), .clk(clk), .pre(pre), .q(y[7]));
//clock the outputs
always @(negedge clk)
        begin
                aa69 <= a69_buf;
                aa70 <= a70_buf;
                aa71 <= a71_buf;
                aa72 <= a72_buf;
                aa73 <= a73_buf;
                aa74 <= a74_buf;
                aa75 <= a75_buf;
                aa76 <= a76_buf;

        end

//***Fourth Stage***
```

```
//clock the inputs
always @(negedge clk)
	begin
		aa49 <= a49_buf;
		aa50 <= a50_buf;
		aa51 <= a51_buf;
		aa52 <= a52_buf;

	end
//input A for DFF:  cells (a69-a72) , wires (a77-a80)
	dff_cp_1x a69( .clr(clr), .d(aa49), .clk(clk), .pre(pre), .q(a77_buf));
	dff_cp_1x a70( .clr(clr), .d(aa50), .clk(clk), .pre(pre), .q(a78_buf));
	dff_cp_1x a71( .clr(clr), .d(aa51), .clk(clk), .pre(pre), .q(a79_buf));
	dff_cp_1x a72( .clr(clr), .d(aa52), .clk(clk), .pre(pre), .q(a80_buf));


//clock the inputs
always @(negedge clk)
	begin
		aa53 <= a53_buf;
		aa54 <= a54_buf;
		aa55 <= a55_buf;
		aa56 <= a56_buf;

	end
//input B for DFF:  cells (a73-a76) , wires (a81-a84)
```

```
        dff_cp_1x a73( .clr(clr), .d(aa53), .clk(clk), .pre(pre), .q(a81_buf));

        dff_cp_1x a74( .clr(clr), .d(aa54), .clk(clk), .pre(pre), .q(a82_buf));

        dff_cp_1x a75( .clr(clr), .d(aa55), .clk(clk), .pre(pre), .q(a83_buf));

        dff_cp_1x a76( .clr(clr), .d(aa56), .clk(clk), .pre(pre), .q(a84_buf));


//and second stage: cells(a77-a80), wires(a85-a88)

        and2_1x_SETH a77( .in1(a84_buf), .in0(a77_buf), .clk(clk), .y(a85_buf),
.SETH(SETH), .SETL(SETL), .ADDR(29));

        and2_1x_SETH a78( .in1(a84_buf), .in0(a78_buf), .clk(clk), .y(a86_buf),
.SETH(SETH), .SETL(SETL), .ADDR(30));

        and2_1x_SETH a79( .in1(a84_buf), .in0(a79_buf), .clk(clk), .y(a87_buf),
.SETH(SETH), .SETL(SETL), .ADDR(31));

        and2_1x_SETH a80( .in1(a84_buf), .in0(a80_buf), .clk(clk), .y(a88_buf),
.SETH(SETH), .SETL(SETL), .ADDR(32));


//add to previous stage using Full Adder: cells (a81-a84), wires (a89-a96)


        add_1x1x_SETH a81( .a(aa72), .b(a85_buf), .cin(C), .clk(clk), .cout(a89_buf),
.y(a90_buf), .SETH(SETH), .SETL(SETL), .ADDR_COUT(33), .ADDR_Y(34));

        add_1x1x_SETH a82( .a(aa73), .b(a86_buf), .cin(a89_buf), .clk(clk),
.cout(a91_buf), .y(a92_buf), .SETH(SETH), .SETL(SETL), .ADDR_COUT(35),
.ADDR_Y(36));

        add_1x1x_SETH a83( .a(aa74), .b(a87_buf), .cin(a91_buf), .clk(clk),
.cout(a93_buf), .y(a94_buf), .SETH(SETH), .SETL(SETL), .ADDR_COUT(37),
.ADDR_Y(38));

        add_1x1x_SETH a84( .a(aa67), .b(a88_buf), .cin(a93_buf), .clk(clk),
.cout(a95_buf), .y(a96_buf), .SETH(SETH), .SETL(SETL), .ADDR_COUT(39),
.ADDR_Y(40));


//Results register: cells(a85-a92)


        dff_cp_1x a85( .clr(clr), .d(aa69), .clk(clk), .pre(pre), .q(p[0]));
```

```
        dff_cp_1x a86( .clr(clr), .d(aa70), .clk(clk), .pre(pre), .q(p[1]));
        dff_cp_1x a87( .clr(clr), .d(aa71), .clk(clk), .pre(pre), .q(p[2]));
        dff_cp_1x a88( .clr(clr), .d(a90_buf), .clk(clk), .pre(pre), .q(p[3]));
        dff_cp_1x a89( .clr(clr), .d(a92_buf), .clk(clk), .pre(pre), .q(p[4]));
        dff_cp_1x a90( .clr(clr), .d(a94_buf), .clk(clk), .pre(pre), .q(p[5]));
        dff_cp_1x a91( .clr(clr), .d(a96_buf), .clk(clk), .pre(pre), .q(p[6]));
        dff_cp_1x a92( .clr(clr), .d(a95_buf), .clk(clk), .pre(pre), .q(p[7]));
endmodule
```

APPENDIX F

Code to compare two 4x4 multipliers

This is the code to compare two 4x4 multipliers. This takes the results, ***y*** and ***p***, of two 4x4 multipliers and performs a bit by bit comparison between them. If the bits are equal it will give a 0 and if the bits are different then it will give a 1. It also adds up the result of each bit by bit comparison to find the total number non-equal bits. The output for the bit by bit comparison is **compare** signal and the output for the total number of non-equal bits is **count1** signal.

CODE:

```
//timescale for the simluations
`timescale 1ns / 100ps
//the multiplier modules that are needed to give the inputs for the comparator
`include "multi_parallel_4x4.v"
`include "multi_parallel_4x4_SET.v"

//module setup
//the beginning of the module
module compare_parallel_4x4 (A, B, C, cin, clk, pre, clr, SETH, SETL, compare,
count1);

//inputs
        input [3:0] A, B;
        input [31:0] SETH, SETL;
```

```
        input clk, pre, cin, C, clr;
//wires
        wire [7:0] y;
        wire [7:0] p;
//outputs
        output [7:0] compare;
        output [3:0] count1;

//regs
reg aa, bb, cc, dd, ee, ff, gg, hh;
reg [3:0] z;
//instantiate the two multipliers
multi_parallel_4x4 multi_parallel_4x4_instance1( .A(A), .B(B), .C(C), .cin(cin), .clk(clk), .pre(pre), .clr(clr), .y(y));
multi_parallel_4x4_SET multi_parallel_4x4_SET_instance1( .A(A), .B(B), .C(C), .cin(cin), .clk(clk), .pre(pre), .clr(clr), .p(p), .SETH(SETH), .SETL(SETL));

//perform a bit by bit comparison of the outputs on the positive edge clock cycle
always@(posedge clk)
if (y[0]==p[0])
        aa<=0;
        else
        aa<=1;

always@(posedge clk)
if (y[1]==p[1])
        bb<=0;
        else
```

```
        bb<=1;

always@(posedge clk)
if (y[2]==p[2])
        cc<=0;
        else
        cc<=1;

always@(posedge clk)
if (y[3]==p[3])
        dd<=0;
        else
        dd<=1;

always@(posedge clk)
if (y[4]==p[4])
        ee<=0;
        else
        ee<=1;

always@(posedge clk)
if (y[5]==p[5])
        ff<=0;
        else
        ff<=1;

always@(posedge clk)
```

```
if (y[6]==p[6])
        gg<=0;
        else
        gg<=1;

always@(posedge clk)
if (y[7]==p[7])
        hh<=0;
        else
        hh<=1;
//add the result of the comparison together to get the total number of non-equal bits
always@(posedge clk)
       z<=aa+bb+cc+dd+ee+ff+gg+hh;
 //output the total number of non-equal bits
        dff_cp_1x a5273 ( .clr(clr), .d(z[0]), .clk(clk), .pre(pre), .q(count1[0]));
        dff_cp_1x a5274 ( .clr(clr), .d(z[1]), .clk(clk), .pre(pre), .q(count1[1]));
        dff_cp_1x a5275 ( .clr(clr), .d(z[2]), .clk(clk), .pre(pre), .q(count1[2]));
        dff_cp_1x a5276 ( .clr(clr), .d(z[3]), .clk(clk), .pre(pre), .q(count1[3]));

//output comparator results cells (a5242-a5248)

        dff_cp_1x a5241 ( .clr(clr), .d(aa), .clk(clk), .pre(pre), .q(compare[0]));
        dff_cp_1x a5242 ( .clr(clr), .d(bb), .clk(clk), .pre(pre), .q(compare[1]));
        dff_cp_1x a5243 ( .clr(clr), .d(cc), .clk(clk), .pre(pre), .q(compare[2]));
        dff_cp_1x a5244 ( .clr(clr), .d(dd), .clk(clk), .pre(pre), .q(compare[3]));
        dff_cp_1x a5245 ( .clr(clr), .d(ee), .clk(clk), .pre(pre), .q(compare[4]));
        dff_cp_1x a5246 ( .clr(clr), .d(ff), .clk(clk), .pre(pre), .q(compare[5]));
```

```
    dff_cp_1x a5247 ( .clr(clr), .d(gg), .clk(clk), .pre(pre), .q(compare[6]));
    dff_cp_1x a5248 ( .clr(clr), .d(hh), .clk(clk), .pre(pre), .q(compare[7]));

endmodule
```

APPENDIX G

Code for a 4x4 parallel-implemented full adder multiplier testbench

This is the code for a 4x4 parallel-implemented full adder multiplier testbench. This supplies the inputs, **A**, **B**, **SETH** and **SETL**, for the multipliers. The inputs can be changed every clock cycle. The testbench prints the inputs for the multipliers and the outputs from the comparator.

CODE:

```
//timescale for the simulations
`timescale 1ns / 100ps
//the beginning of the module
module compare_parallel_4x4_stim;
 //integers
  integer SETH, SETL;
//regs
  reg  cin, clk, clr, pre, C;
  reg [3:0] A, B;
//wires
  wire [7:0] compare;
  wire [3:0] count1;
  wire [7:0] y, p;
//instantiate the multipliers and the comparator modules
  multi_parallel_4x4 multi_parallel_4x4_instance1(A, B, C, cin, clk, pre, clr, y);
  multi_parallel_4x4_SET multi_parallel_4x4_SET_instance1(A, B, C, cin, clk, pre, clr,
p, SETH, SETL);
```

```
  compare_parallel_4x4  compare_parallel_4x4_instance1( A, B, C, cin, clk, pre, clr, SETH, SETL, compare, count1);


  // stimulus information
//initial the clock
  initial
   clk = 1'b0;
//set the clock to repeat
  always
   #1 clk = ~clk;
//initial the inputs
  initial
begin
    A = 4'd0; B=4'd0; cin=1'd0; pre=1'd1; clr=1'd1; C=1'd0; SETH=0; SETL=0;
//change the inputs
    #2 A = 4'd15; B = 4'd15;  SETL=5;


    #2 A = 4'd7; B = 4'd9;


    #2 A = 4'd12; B = 4'd14;


    #2 A = 4'd2; B = 4'd5;


    #18
//display simulations complete and finish
      $display("\n Simulation complete\n");
      $finish;
    end
```

```
  // probe information

//print the inputs and outputs on the positive edge clock

   always@(posedge clk)

      $display($time,"   %d %d %d %d %d %d  %d %d", A, B, SETH, SETL, y, p,
compare, count1);


endmodule
```

APPENDIX H

Code for a 4x4 serial-implemented full adder multiplier module

This is the code for a 4x4 serial-implemented full adder multiplier module. It takes the inputs from the testbench and multiplies them in a serial full adder style. It gives an output, ***rr***.

CODE:

```
`timescale 1ns / 100ps
`include "multi_4x4_dff_small.v"
`include "setlibraryfile.v"

//module setup

module inputs_and_multi_small4x4 (B, C1, cin, clk, pre, clr, clr1, c, rr, s44, s45, s46,
s47, s144, s145, s146, s147, s244, s245, s246, s247, s344, s345, s346, s347, s444, s445,
s446, s447, s544, s545, s546, s547, s644, s645, s646, s647, s744, s745, s746, s747, s844,
s845, s846, s847, s944, s945, s946, s947, s1044, s1045, s1046, s1047, clk2, clk3, clk4,
clk5, clk6, clk7, clk8, clk9, clk10, clk11, clk12, count, count1);

//inputs and outputs

    input [3:0] B;
    input [3:0] C1;
  input clk, pre, cin, clr, clr1, c;
    input [31:0] count, count1;
    input clk2, clk3, clk4, clk5, clk6, clk7, clk8, clk9, clk10, clk11, clk12;
```

```
    input s44, s45, s46, s47;
    input s144, s145, s146, s147;
    input s244, s245, s246, s247;
    input s344, s345, s346, s347;
    input s444, s445, s446, s447;
    input s544, s545, s546, s547;
    input s644, s645, s646, s647;
    input s744, s745, s746, s747;
    input s844, s845, s846, s847;
    input s944, s945, s946, s947;
    input s1044, s1045, s1046, s1047;


    wire [7:0] y1, y2, y3, y4, y5, y6, y7, y8, y9, y10, y11;

    wire [7:0] b1, b2, b3, b4, b5, b6, b7, b8, b9, b10, b11, b12, b13, b14, b15, b16,
b17, b18, b19, b20, b21, b22, b23, b24, b25, b26, b27, b28, b29, b30, b31, b32, b33, b34,
b35, b36, b37, b38, b39, b40, b41, b42, b43, b44, b45, b46, b47, b48, b49, b50;
   wire [3:0] S;

    reg r1, r2, r3, r4, r5, r6, r7, r8;
    reg [3:0] e1, e2, e3, e4, e5, e6, e7, e8, e9, e10, e11;
   wire [7:0] c1;

    output [7:0] rr;

//input B
```

```
        inv_4x a61(.in0(B[0]), .y(a0_buf));
        inv_16x a62(.in0(a0_buf), .y(S[0]));

        inv_4x a63(.in0(B[1]), .y(a2_buf));
        inv_16x a64(.in0(a2_buf), .y(S[1]));

        inv_4x a65(.in0(B[2]), .y(a4_buf));
        inv_16x a66(.in0(a4_buf), .y(S[2]));

        inv_4x a67(.in0(B[3]), .y(a6_buf));
        inv_16x a68(.in0(a6_buf), .y(S[3]));



//input C

        inv_4x a93(.in0(C1[0]), .y(a32_buf));
        inv_16x a94(.in0(a32_buf), .y(b1[0]));

        inv_4x a95(.in0(C1[1]), .y(a34_buf));
        inv_16x a96(.in0(a34_buf), .y(b1[1]));

        inv_4x a97(.in0(C1[2]), .y(a36_buf));
        inv_16x a98(.in0(a36_buf), .y(b1[2]));

        inv_4x a99(.in0(C1[3]), .y(a38_buf));
        inv_16x a100(.in0(a38_buf), .y(b1[3]));
```

```
always@(negedge clk)
      begin
     if (count1==1)
       e1<=b1;
     else
       e1<=0;

      end

always@(negedge clk)
     begin
     if (count1==2)
       e2<=b1;
      else
       e2<=0;
     end

always@(negedge clk)
   begin
     if (count1==3)
       e3<=b1;
    else
       e3<=0;
```

```
   end

always@(negedge clk)
    begin
     if (count1==4)
       e4<=b1;
     else
       e4<=0;
    end

always@(negedge clk)
    begin
     if (count1==5)
       e5<=b1;
     else
       e5<=0;
    end

always@(negedge clk)
    begin
     if (count1==6)
       e6<=b1;
     else
       e6<=0;
     end

always@(negedge clk)
```

```
    begin
     if (count1==7)
       e7<=b1;
     else
       e7<=0;
     end

always@(negedge clk)
    begin
     if (count1==8)
       e8<=b1;
     else
       e8<=0;
     end

always@(negedge clk)
    begin
     if (count1==9)
       e9<=b1;
     else
       e9<=0;
     end

always@(negedge clk)
    begin
     if (count1==10)
       e10<=b1;
```

```
    else
      e10<=0;
    end

always@(negedge clk)
   begin
    if (count1==11)
      e11<=b1;
    else
      e11<=0;
    end
```

```
        multi_4x4_dff_small a2(.A(S), .B(e1), .C(c), .cin(cin), .clk(clk), .pre(pre),
.clr(clr), .y(y1), .s44(s44), .s45(s45), .s46(s46), .s47(s47), .clk2(clk2));
```

```
        multi_4x4_dff_small a3(.A(S), .B(e2), .C(c), .cin(cin), .clk(clk), .pre(pre),
.clr(clr), .y(y2), .s44(s144), .s45(s145), .s46(s146), .s47(s147), .clk2(clk3));
```

```
        multi_4x4_dff_small a4(.A(S), .B(e3), .C(c), .cin(cin), .clk(clk), .pre(pre),
.clr(clr), .y(y3), .s44(s244), .s45(s245), .s46(s246), .s47(s247), .clk2(clk4));
```

multi_4x4_dff_small a5(.A(S), .B(e4), .C(c), .cin(cin), .clk(clk), .pre(pre), .clr(clr), .y(y4), .s44(s344), .s45(s345), .s46(s346), .s47(s347), .clk2(clk5));

multi_4x4_dff_small a6(.A(S), .B(e5), .C(c), .cin(cin), .clk(clk), .pre(pre), .clr(clr), .y(y5), .s44(s444), .s45(s445), .s46(s446), .s47(s447), .clk2(clk6));

multi_4x4_dff_small a7(.A(S), .B(e6), .C(c), .cin(cin), .clk(clk), .pre(pre), .clr(clr), .y(y6), .s44(s544), .s45(s545), .s46(s546), .s47(s547), .clk2(clk7));

multi_4x4_dff_small a8(.A(S), .B(e7), .C(c), .cin(cin), .clk(clk), .pre(pre), .clr(clr), .y(y7), .s44(s644), .s45(s645), .s46(s646), .s47(s647), .clk2(clk8));

multi_4x4_dff_small a9(.A(S), .B(e8), .C(c), .cin(cin), .clk(clk), .pre(pre), .clr(clr), .y(y8), .s44(s744), .s45(s745), .s46(s746), .s47(s747), .clk2(clk9));

multi_4x4_dff_small a10(.A(S), .B(e9), .C(c), .cin(cin), .clk(clk), .pre(pre), .clr(clr), .y(y9), .s44(s844), .s45(s845), .s46(s846), .s47(s847), .clk2(clk10));

```
        multi_4x4_dff_small a11(.A(S), .B(e10), .C(c), .cin(cin), .clk(clk), .pre(pre),
.clr(clr), .y(y10), .s44(s944), .s45(s945), .s46(s946), .s47(s947), .clk2(clk11));




        multi_4x4_dff_small a12(.A(S), .B(e11), .C(c), .cin(cin), .clk(clk), .pre(pre),
.clr(clr), .y(y11), .s44(s1044), .s45(s1045), .s46(s1046), .s47(s1047), .clk2(clk12));




always @(posedge clk)
  begin
    if (count==1)
    begin
      r1 <= y1[0];
      r2 <= y1[1];
      r3 <= y1[2];
      r4 <= y1[3];
      r5 <= y1[4];
      r6 <= y1[5];
      r7 <= y1[6];
      r8 <= y1[7];

   end
 end
```

```
always @(posedge clk)
  begin
    if (count==2)
    begin
      r1 <= y2[0];
      r2 <= y2[1];
      r3 <= y2[2];
      r4 <= y2[3];
      r5 <= y2[4];
      r6 <= y2[5];
      r7 <= y2[6];
      r8 <= y2[7];

   end
 end

always @(posedge clk)
  begin
    if (count==3)
    begin
      r1 <= y3[0];
      r2 <= y3[1];
      r3 <= y3[2];
      r4 <= y3[3];
      r5 <= y3[4];
      r6 <= y3[5];
```

```
      r7 <= y3[6];
      r8 <= y3[7];

    end
  end

always @(posedge clk)
   begin
     if (count==4)
     begin
       r1 <= y4[0];
       r2 <= y4[1];
       r3 <= y4[2];
       r4 <= y4[3];
       r5 <= y4[4];
       r6 <= y4[5];
       r7 <= y4[6];
       r8 <= y4[7];

    end
  end

always @(posedge clk)
   begin
     if (count==5)
     begin
       r1 <= y5[0];
```

```
      r2 <= y5[1];
      r3 <= y5[2];
      r4 <= y5[3];
      r5 <= y5[4];
      r6 <= y5[5];
      r7 <= y5[6];
      r8 <= y5[7];

    end
  end


always @(posedge clk)
   begin
     if (count==6)
     begin
       r1 <= y6[0];
       r2 <= y6[1];
       r3 <= y6[2];
       r4 <= y6[3];
       r5 <= y6[4];
       r6 <= y6[5];
       r7 <= y6[6];
       r8 <= y6[7];

    end
  end
```

```
always @(posedge clk)
  begin
    if (count==7)
    begin
      r1 <= y7[0];
      r2 <= y7[1];
      r3 <= y7[2];
      r4 <= y7[3];
      r5 <= y7[4];
      r6 <= y7[5];
      r7 <= y7[6];
      r8 <= y7[7];

   end
 end


always @(posedge clk)
  begin
    if (count==8)
    begin
      r1 <= y8[0];
      r2 <= y8[1];
      r3 <= y8[2];
      r4 <= y8[3];
      r5 <= y8[4];
```

```
      r6 <= y8[5];
      r7 <= y8[6];
      r8 <= y8[7];

   end
  end

always @(posedge clk)
   begin
     if (count==9)
     begin
       r1 <= y9[0];
       r2 <= y9[1];
       r3 <= y9[2];
       r4 <= y9[3];
       r5 <= y9[4];
       r6 <= y9[5];
       r7 <= y9[6];

   end
  end

always @(posedge clk)
   begin
     if (count==10)
     begin
       r1 <= y10[0];
```

```
      r2 <= y10[1];
      r3 <= y10[2];
      r4 <= y10[3];
      r5 <= y10[4];
      r6 <= y10[5];
      r7 <= y10[6];
      r8 <= y10[7];

   end
  end


always @(posedge clk)
   begin
     if (count==11)
     begin
       r1 <= y11[0];
       r2 <= y11[1];
       r3 <= y11[2];
       r4 <= y11[3];
       r5 <= y11[4];
       r6 <= y11[5];
       r7 <= y11[6];
       r8 <= y11[7];

    end
  end
```

```
//output comparator results cells (a1561-a1592)

        dff_cp_1x a5241 ( .clr(clr), .d(r1), .clk(clk), .pre(pre), .q(rr[0]));
        dff_cp_1x a5242 ( .clr(clr), .d(r2), .clk(clk), .pre(pre), .q(rr[1]));
        dff_cp_1x a5243 ( .clr(clr), .d(r3), .clk(clk), .pre(pre), .q(rr[2]));
        dff_cp_1x a5244 ( .clr(clr), .d(r4), .clk(clk), .pre(pre), .q(rr[3]));
        dff_cp_1x a5245 ( .clr(clr), .d(r5), .clk(clk), .pre(pre), .q(rr[4]));
        dff_cp_1x a5246 ( .clr(clr), .d(r6), .clk(clk), .pre(pre), .q(rr[5]));
        dff_cp_1x a5247 ( .clr(clr), .d(r7), .clk(clk), .pre(pre), .q(rr[6]));
        dff_cp_1x a5248 ( .clr(clr), .d(r8), .clk(clk), .pre(pre), .q(rr[7]));



endmodule
```

APPENDIX I

Code for a 4x4 SET serial-implemented full adder multiplier module

This is the code for a 4x4 SET serial-implemented full adder multiplier module. It takes the inputs from the testbench and multiplies the inputs using serial full adders. This also has input signals for **SETH**, **SETL**, **SETH1** through **SETH11**, and **SETL1** through **SETL11**. This will drive the output node on the AND gates and the carry and output nodes on the full adders to high or low. It gives the output, ***r***, of the multiplication.

CODE:

```
`timescale 1ns / 100ps
`include "multi_4x4_dff_small_SET.v"
`include "setlibraryfile.v"
`include "setlibrary_SETH.v"

//module setup

module inputs_and_multi_small4x4_SET (B, C1, cin, clk, pre, clr, clr1, c, r, s44, s45,
s46, s47, s144, s145, s146, s147, s244, s245, s246, s247, s344, s345, s346, s347, s444,
s445, s446, s447, s544, s545, s546, s547, s644, s645, s646, s647, s744, s745, s746, s747,
s844, s845, s846, s847, s944, s945, s946, s947, s1044, s1045, s1046, s1047, clk2, clk3,
clk4, clk5, clk6, clk7, clk8, clk9, clk10, clk11, clk12, count, count1, SETH, SETL,
SETH1, SETL1, SETH2, SETL2, SETH3, SETL3, SETH4, SETL4, SETH5, SETL5,
SETH6, SETL6, SETH7, SETL7, SETH8, SETL8, SETH9, SETL9, SETH10, SETL10,
SETH11, SETL11);

//inputs and outputs
```

```
    input [3:0] B;

    input [3:0] C1;

    input [31:0] SETH, SETL, SETH1, SETL1, SETH2, SETL2, SETH3, SETL3,
SETH4, SETL4, SETH5, SETL5, SETH6, SETL6, SETH7, SETL7, SETH8, SETL8,
SETH9, SETL9, SETH10, SETL10, SETH11, SETL11;

  input clk, pre, cin, clr, clr1, c;

    input [31:0] count, count1;

    input clk2, clk3, clk4, clk5, clk6, clk7, clk8, clk9, clk10, clk11, clk12;

    input s44, s45, s46, s47;

    input s144, s145, s146, s147;

    input s244, s245, s246, s247;

    input s344, s345, s346, s347;

    input s444, s445, s446, s447;

    input s544, s545, s546, s547;

    input s644, s645, s646, s647;

    input s744, s745, s746, s747;

    input s844, s845, s846, s847;

    input s944, s945, s946, s947;

    input s1044, s1045, s1046, s1047;



    wire [7:0] y1, y2, y3, y4, y5, y6, y7, y8, y9, y10, y11;


    wire [7:0] b1, b2, b3, b4, b5, b6, b7, b8, b9, b10, b11, b12, b13, b14, b15, b16,
b17, b18, b19, b20, b21, b22, b23, b24, b25, b26, b27, b28, b29, b30, b31, b32, b33, b34,
b35, b36, b37, b38, b39, b40, b41, b42, b43, b44, b45, b46, b47, b48, b49, b50;

  wire [3:0] S;


    reg r1, r2, r3, r4, r5, r6, r7, r8;
```

```
        reg [3:0] e1, e2, e3, e4, e5, e6, e7, e8, e9, e10, e11;
      wire [7:0] c1;

        output [7:0] r;

//input B

        inv_4x_SETH a61(.in0(B[0]), .clk(clk), .y(a0_buf), .SETH(SETH),
.SETL(SETL), .ADDR(1));
        inv_16x_SETH a62(.in0(a0_buf), .clk(clk), .y(S[0]), .SETH(SETH),
.SETL(SETL), .ADDR(2));

        inv_4x_SETH a63(.in0(B[1]), .clk(clk), .y(a2_buf), .SETH(SETH),
.SETL(SETL), .ADDR(3));
        inv_16x_SETH a64(.in0(a2_buf), .clk(clk), .y(S[1]), .SETH(SETH),
.SETL(SETL), .ADDR(4));

        inv_4x_SETH a65(.in0(B[2]), .clk(clk), .y(a4_buf), .SETH(SETH),
.SETL(SETL), .ADDR(5));
        inv_16x_SETH a66(.in0(a4_buf), .clk(clk), .y(S[2]), .SETH(SETH),
.SETL(SETL), .ADDR(6));

        inv_4x_SETH a67(.in0(B[3]), .clk(clk), .y(a6_buf), .SETH(SETH),
.SETL(SETL), .ADDR(7));
        inv_16x_SETH a68(.in0(a6_buf), .clk(clk), .y(S[3]), .SETH(SETH),
.SETL(SETL), .ADDR(8));

//input C
```

```
        inv_4x_SETH a93(.in0(C1[0]), .clk(clk), .y(a32_buf), .SETH(SETH),
.SETL(SETL), .ADDR(9));

        inv_16x_SETH a94(.in0(a32_buf), .clk(clk), .y(b1[0]), .SETH(SETH),
.SETL(SETL), .ADDR(10));



        inv_4x_SETH a95(.in0(C1[1]), .clk(clk), .y(a34_buf), .SETH(SETH),
.SETL(SETL), .ADDR(11));

        inv_16x_SETH a96(.in0(a34_buf), .clk(clk), .y(b1[1]), .SETH(SETH),
.SETL(SETL), .ADDR(12));



        inv_4x_SETH a97(.in0(C1[2]), .clk(clk), .y(a36_buf), .SETH(SETH),
.SETL(SETL), .ADDR(13));

        inv_16x_SETH a98(.in0(a36_buf), .clk(clk), .y(b1[2]), .SETH(SETH),
.SETL(SETL), .ADDR(14));



        inv_4x_SETH a99(.in0(C1[3]), .clk(clk), .y(a38_buf), .SETH(SETH),
.SETL(SETL), .ADDR(15));

        inv_16x_SETH a100(.in0(a38_buf), .clk(clk), .y(b1[3]), .SETH(SETH),
.SETL(SETL), .ADDR(16));









always@(negedge clk)

    begin

   if (count1==1)

     e1<=b1;

   else

     e1<=0;


    end
```

```
always@(negedge clk)
     begin
     if (count1==2)
       e2<=b1;
      else
       e2<=0;
     end

always@(negedge clk)
   begin
     if (count1==3)
       e3<=b1;
    else
       e3<=0;
   end

always@(negedge clk)
    begin
     if (count1==4)
       e4<=b1;
     else
       e4<=0;
    end

always@(negedge clk)
    begin
```

```
     if (count1==5)
       e5<=b1;
     else
       e5<=0;
    end

always@(negedge clk)
    begin
     if (count1==6)
       e6<=b1;
     else
       e6<=0;
     end

always@(negedge clk)
    begin
     if (count1==7)
       e7<=b1;
     else
       e7<=0;
     end

always@(negedge clk)
    begin
     if (count1==8)
       e8<=b1;
     else
```

```
      e8<=0;
    end

always@(negedge clk)
   begin
    if (count1==9)
      e9<=b1;
    else
      e9<=0;
    end

always@(negedge clk)
   begin
    if (count1==10)
      e10<=b1;
    else
      e10<=0;
    end

always@(negedge clk)
   begin
    if (count1==11)
      e11<=b1;
    else
      e11<=0;
    end
```

multi_4x4_dff_small_SET a2(.A(S), .B(e1), .C(c), .cin(cin), .clk(clk), .pre(pre), .clr(clr), .y(y1), .s44(s44), .s45(s45), .s46(s46), .s47(s47), .clk2(clk2), .SETH(SETH1), .SETL(SETL1));

multi_4x4_dff_small_SET a3(.A(S), .B(e2), .C(c), .cin(cin), .clk(clk), .pre(pre), .clr(clr), .y(y2), .s44(s144), .s45(s145), .s46(s146), .s47(s147), .clk2(clk3), .SETH(SETH2), .SETL(SETL2));

multi_4x4_dff_small_SET a4(.A(S), .B(e3), .C(c), .cin(cin), .clk(clk), .pre(pre), .clr(clr), .y(y3), .s44(s244), .s45(s245), .s46(s246), .s47(s247), .clk2(clk4), .SETH(SETH3), .SETL(SETL3));

multi_4x4_dff_small_SET a5(.A(S), .B(e4), .C(c), .cin(cin), .clk(clk), .pre(pre), .clr(clr), .y(y4), .s44(s344), .s45(s345), .s46(s346), .s47(s347), .clk2(clk5), .SETH(SETH4), .SETL(SETL4));

multi_4x4_dff_small_SET a6(.A(S), .B(e5), .C(c), .cin(cin), .clk(clk), .pre(pre), .clr(clr), .y(y5), .s44(s444), .s45(s445), .s46(s446), .s47(s447), .clk2(clk6), .SETH(SETH5), .SETL(SETL5));

multi_4x4_dff_small_SET a7(.A(S), .B(e6), .C(c), .cin(cin), .clk(clk), .pre(pre), .clr(clr), .y(y6), .s44(s544), .s45(s545), .s46(s546), .s47(s547), .clk2(clk7), .SETH(SETH6), .SETL(SETL6));

multi_4x4_dff_small_SET a8(.A(S), .B(e7), .C(c), .cin(cin), .clk(clk), .pre(pre), .clr(clr), .y(y7), .s44(s644), .s45(s645), .s46(s646), .s47(s647), .clk2(clk8), .SETH(SETH7), .SETL(SETL7));

multi_4x4_dff_small_SET a9(.A(S), .B(e8), .C(c), .cin(cin), .clk(clk), .pre(pre), .clr(clr), .y(y8), .s44(s744), .s45(s745), .s46(s746), .s47(s747), .clk2(clk9), .SETH(SETH8), .SETL(SETL8));

multi_4x4_dff_small_SET a10(.A(S), .B(e9), .C(c), .cin(cin), .clk(clk), .pre(pre), .clr(clr), .y(y9), .s44(s844), .s45(s845), .s46(s846), .s47(s847), .clk2(clk10), .SETH(SETH9), .SETL(SETL9));

multi_4x4_dff_small_SET a11(.A(S), .B(e10), .C(c), .cin(cin), .clk(clk), .pre(pre), .clr(clr), .y(y10), .s44(s944), .s45(s945), .s46(s946), .s47(s947), .clk2(clk11), .SETH(SETH10), .SETL(SETL10));

```
        multi_4x4_dff_small_SET a12(.A(S), .B(e11), .C(c), .cin(cin), .clk(clk), .pre(pre),
.clr(clr), .y(y11), .s44(s1044), .s45(s1045), .s46(s1046), .s47(s1047), .clk2(clk12),
.SETH(SETH11), .SETL(SETL11));




always @(posedge clk)
  begin
    if (count==1)
    begin
      r1 <= y1[0];
      r2 <= y1[1];
      r3 <= y1[2];
      r4 <= y1[3];
      r5 <= y1[4];
      r6 <= y1[5];
      r7 <= y1[6];
      r8 <= y1[7];

   end
 end

always @(posedge clk)
  begin
    if (count==2)
    begin
```

```
        r1 <= y2[0];
        r2 <= y2[1];
        r3 <= y2[2];
        r4 <= y2[3];
        r5 <= y2[4];
        r6 <= y2[5];
        r7 <= y2[6];
        r8 <= y2[7];

    end
  end

always @(posedge clk)
    begin
      if (count==3)
      begin
        r1 <= y3[0];
        r2 <= y3[1];
        r3 <= y3[2];
        r4 <= y3[3];
        r5 <= y3[4];
        r6 <= y3[5];
        r7 <= y3[6];
        r8 <= y3[7];

    end
  end
```

```
always @(posedge clk)
  begin
    if (count==4)
    begin
      r1 <= y4[0];
      r2 <= y4[1];
      r3 <= y4[2];
      r4 <= y4[3];
      r5 <= y4[4];
      r6 <= y4[5];
      r7 <= y4[6];
      r8 <= y4[7];

   end
 end

always @(posedge clk)
  begin
    if (count==5)
    begin
      r1 <= y5[0];
      r2 <= y5[1];
      r3 <= y5[2];
      r4 <= y5[3];
      r5 <= y5[4];
      r6 <= y5[5];
```

```
      r7 <= y5[6];
      r8 <= y5[7];

    end
  end


always @(posedge clk)
   begin
     if (count==6)
     begin
       r1 <= y6[0];
       r2 <= y6[1];
       r3 <= y6[2];
       r4 <= y6[3];
       r5 <= y6[4];
       r6 <= y6[5];
       r7 <= y6[6];
       r8 <= y6[7];

    end
  end

always @(posedge clk)
   begin
     if (count==7)
     begin
```

```
      r1 <= y7[0];
      r2 <= y7[1];
      r3 <= y7[2];
      r4 <= y7[3];
      r5 <= y7[4];
      r6 <= y7[5];
      r7 <= y7[6];
      r8 <= y7[7];

    end
  end


always @(posedge clk)
   begin
     if (count==8)
     begin
       r1 <= y8[0];
       r2 <= y8[1];
       r3 <= y8[2];
       r4 <= y8[3];
       r5 <= y8[4];
       r6 <= y8[5];
       r7 <= y8[6];
       r8 <= y8[7];

    end
```

```
  end

always @(posedge clk)
   begin
      if (count==9)
      begin
        r1 <= y9[0];
        r2 <= y9[1];
        r3 <= y9[2];
        r4 <= y9[3];
        r5 <= y9[4];
        r6 <= y9[5];
        r7 <= y9[6];

    end
  end

always @(posedge clk)
   begin
      if (count==10)
      begin
        r1 <= y10[0];
        r2 <= y10[1];
        r3 <= y10[2];
        r4 <= y10[3];
        r5 <= y10[4];
        r6 <= y10[5];
```

```
      r7 <= y10[6];
      r8 <= y10[7];

    end
  end


always @(posedge clk)
   begin
     if (count==11)
     begin
       r1 <= y11[0];
       r2 <= y11[1];
       r3 <= y11[2];
       r4 <= y11[3];
       r5 <= y11[4];
       r6 <= y11[5];
       r7 <= y11[6];
       r8 <= y11[7];

    end
  end


//output comparator results cells (a1561-a1592)
```

```
dff_cp_1x a5241 ( .clr(clr), .d(r1), .clk(clk), .pre(pre), .q(r[0]));
dff_cp_1x a5242 ( .clr(clr), .d(r2), .clk(clk), .pre(pre), .q(r[1]));
dff_cp_1x a5243 ( .clr(clr), .d(r3), .clk(clk), .pre(pre), .q(r[2]));
dff_cp_1x a5244 ( .clr(clr), .d(r4), .clk(clk), .pre(pre), .q(r[3]));
dff_cp_1x a5245 ( .clr(clr), .d(r5), .clk(clk), .pre(pre), .q(r[4]));
dff_cp_1x a5246 ( .clr(clr), .d(r6), .clk(clk), .pre(pre), .q(r[5]));
dff_cp_1x a5247 ( .clr(clr), .d(r7), .clk(clk), .pre(pre), .q(r[6]));
dff_cp_1x a5248 ( .clr(clr), .d(r8), .clk(clk), .pre(pre), .q(r[7]));


endmodule
```

APPENDIX J

Code for a 4x4 serial-implemented full adder multiplier testbench

This is the code for a 4x4 serial-implemented full adder multiplier testbench. The testbench supplies the input signals, **B**, **C1**, **SETH**, **SETH1** through **SETH11**, **SETL**, and **SETL1** through **SETL11**, for the multipliers and displays the results for the comparator, **compare** signal, and the number of total errors, **count1** signal, for each set of inputs.

CODE:

```
`timescale 1ns / 100ps


module inputs_and_multi_small4x4_SET_stim;

  integer count, count1, SETH, SETL, SETH1, SETL1, SETH2, SETL2, SETH3, SETL3,
SETH4, SETL4, SETH5, SETL5, SETH6, SETL6, SETH7, SETL7, SETH8, SETL8,
SETH9, SETL9, SETH10, SETL10, SETH11, SETL11;
  reg [3:0] B;
  reg [3:0] C1;
  reg s44, s45, s46, s47;
  reg s144, s145, s146, s147;
  reg s244, s245, s246, s247;
  reg s344, s345, s346, s347;
  reg s444, s445, s446, s447;
  reg s544, s545, s546, s547;
```

```
  reg s644, s645, s646, s647;

  reg s744, s745, s746, s747;

  reg s844, s845, s846, s847;

  reg s944, s945, s946, s947;

  reg s1044, s1045, s1046, s1047;


  reg clr, pre, clk, clr1, c, cin;

  reg clk2, clk3, clk4, clk5, clk6, clk7, clk8, clk9, clk10, clk11, clk12;

  wire [7:0] r;


   inputs_and_multi_small4x4_SET inputs_and_multi_small4x4_SET_instance1(B, C1,
cin, clk, pre, clr, clr1, c, r, s44, s45, s46, s47, s144, s145, s146, s147, s244, s245, s246,
s247, s344, s345, s346, s347, s444, s445, s446, s447, s544, s545, s546, s547, s644, s645,
s646, s647, s744, s745, s746, s747, s844, s845, s846, s847, s944, s945, s946, s947,
s1044, s1045, s1046, s1047, clk2, clk3, clk4, clk5, clk6, clk7, clk8, clk9, clk10, clk11,
clk12, count, count1, SETH, SETL, SETH1, SETL1, SETH2, SETL2, SETH3, SETL3,
SETH4, SETL4, SETH5, SETL5, SETH6, SETL6, SETH7, SETL7, SETH8, SETL8,
SETH9, SETL9, SETH10, SETL10, SETH11, SETL11);


  // stimulus information

  initial

    clk=1'd0;

  always

    #1 clk=~clk;

  initial

    begin

      clr1=1'd1;  B=16'd0; C1=16'd0; cin=1'd0; pre=1'd1; clr=1'd1; c=1'd0; s44=1'd1;
s45=1'd1; s46=1'd1; s47=1'd1; s144=1'd1; s145=1'd1; s146=1'd1; s147=1'd1; s244=1'd1;
s245=1'd1; s246=1'd1; s247=1'd1; s344=1'd1; s345=1'd1; s346=1'd1; s347=1'd1;
s444=1'd1; s445=1'd1; s446=1'd1; s447=1'd1; s544=1'd1; s545=1'd1; s546=1'd1;
s547=1'd1; s644=1'd1; s645=1'd1; s646=1'd1; s647=1'd1; s744=1'd1; s745=1'd1;
s746=1'd1; s747=1'd1; s844=1'd1; s845=1'd1; s846=1'd1; s847=1'd1; s944=1'd1;
s945=1'd1; s946=1'd1; s947=1'd1; s1044=1'd1; s1045=1'd1; s1046=1'd1; s1047=1'd1;
```

clk2=1'd0; clk3=1'd0; clk4=1'd0; clk5=1'd0; clk6=1'd0; clk7=1'd0; clk8=1'd0; clk9=1'd0; clk10=1'd0; clk11=1'd0; clk12=1'd0; count=0; count1=1; SETH=0; SETL=0; SETH1=0; SETL1=0; SETH2=0; SETL2=0; SETH3=0; SETL3=0; SETH4=0; SETL4=0; SETH5=0; SETL5=0; SETH6=0; SETL6=0; SETH7=0; SETL7=0; SETH8=0; SETL8=0; SETH9=0; SETL9=0; SETH10=0; SETL10=0; SETH11=0; SETL11=0;

#1 clk2=1'd1; clk3=1'd1; clk4=1'd1; clk5=1'd1; clk6=1'd1; clk7=1'd1; clk8=1'd1; clk9=1'd1; clk10=1'd1; clk11=1'd1;

#24

#2 B = 4'd1; C1=4'd1; s44=1'd1; s45=1'd1; s46=1'd1; s47=1'd1; clk2=1'd0; count=count+1; SETH2=10;

#1 clk2=1'd1;

#1 s44=1'd0; s45=1'd0; s46=1'd0; s47=1'd0; B = 16'd1; C1=16'd2; s144=1'd1; s145=1'd1; s146=1'd1; s147=1'd1; clk3=1'd0; count=count+1; count1=2;

#1 clk3=1'd1;

#1 s144=1'd0; s145=1'd0; s146=1'd0; s147=1'd0; B = 16'd1; C1=16'd3; s244=1'd1; s245=1'd1; s246=1'd1; s247=1'd1; clk4=1'd0; count=count+1; count1=3;

#1 clk4=1'd1;

#1 s244=1'd0; s245=1'd0; s246=1'd0; s247=1'd0; B = 16'd1; C1=16'd4; s344=1'd1; s345=1'd1; s346=1'd1; s347=1'd1; clk5=1'd0; count=count+1; count1=4;

#1 clk5=1'd1;

#1 s344=1'd0; s345=1'd0; s346=1'd0; s347=1'd0; B = 16'd1; C1=16'd5; s444=1'd1; s445=1'd1; s446=1'd1; s447=1'd1; clk6=1'd0; count=count+1; count1=5;

#1 clk6=1'd1;

#1 s444=1'd0; s445=1'd0; s446=1'd0; s447=1'd0; B = 16'd1; C1=16'd6; s544=1'd1; s545=1'd1; s546=1'd1; s547=1'd1; clk7=1'd0; count=count+1; count1=6;

#1 clk7=1'd1;

#1 s544=1'd0; s545=1'd0; s546=1'd0; s547=1'd0; B = 16'd1; C1=16'd7; s644=1'd1; s645=1'd1; s646=1'd1; s647=1'd1; clk8=1'd0; count=count+1; count1=7;

#1 clk8=1'd1;

#1 s644=1'd0; s645=1'd0; s646=1'd0; s647=1'd0; B = 16'd1; C1=16'd8; s744=1'd1; s745=1'd1; s746=1'd1; s747=1'd1; clk9=1'd0; count=count+1; count1=8;

#1 clk9=1'd1;

#1 s744=1'd0; s745=1'd0; s746=1'd0; s747=1'd0; B = 16'd1; C1=16'd9; s844=1'd1; s845=1'd1; s846=1'd1; s847=1'd1; clk10=1'd0; count=count+1; count1=9;

#1 clk10=1'd1;

#1 s844=1'd0; s845=1'd0; s846=1'd0; s847=1'd0; B = 16'd1; C1=16'd10; s944=1'd1; s945=1'd1; s946=1'd1; s947=1'd1; clk11=1'd0; count=count+1; count1=10;

#1 clk11=1'd1;

#1 s944=1'd0; s945=1'd0; s946=1'd0; s947=1'd0; B = 16'd1; C1=16'd11; s1044=1'd1; s1045=1'd1; s1046=1'd1; s1047=1'd1; clk12=1'd0; count=count+1; count1=11;

#1 clk12=1'd1;

#1 s1044=1'd0; s1045=1'd0; s1046=1'd0; s1047=1'd0; count=count+1; count1=12;

#2 count=count+1;

#2 count=0;

#2 count=count+1;

#2 count=count+1;

#2 count=count+1;

#2 count=count+1;

#2 count=count+1;

#2 count=count+1;

#2 count=count+1;

#2 count=count+1;

#2 count=count+1;

#2 count=count+1;

#2 count=count+1;

#2 count=count+1;

#2 count=count+1;

#2 count=count+1;

#2 count=count+1;

#2 count=count+1;

#2 count=count+1;

```
      $display("\n Simulation complete\n");
      $finish;
    end

  // probe information

always@(posedge clk)
     begin
      $display($time, "    %d %d %d", B, C1, r, SETH2, SETL2);
    end


endmodule
```

APPENDIX K

Code for the plotting of a FFT (Fast Fourier Transform) in Matlab®

This code gives the list of data from the output of the output of a multiplier that does not have any errors in it. The output is graphed along with the FFT.

CODE:

```
x=[1431590230
1710900400
1979419140
2226922990
2443887530
2621924280
2754217600
2835655760
2863224150
2835655760
2754217600
2621924280
2443887530
2226922990
1979419140
1710900400
1431590230
1152280060
```

883761320

636257470

419292930

241256180

108962860

27524700

0

27524700

108962860

241256180

419292930

636257470

883761320

1152280060

1431590230

1710900400

1979419140

2226922990

2443887530

2621924280

2754217600

2835655760

2863224150

2835655760

2754217600

2621924280

2443887530

2226922990

1979419140

1710900400

1431590230

1152280060

883761320

636257470

419292930

241256180

108962860

27524700

0

27524700

108962860

241256180

419292930

636257470

883761320

1152280060

1431590230

1710900400

1979419140

2226922990

2443887530

2621924280

2754217600

2835655760

2863224150

2835655760

2754217600

2621924280

2443887530

2226922990

1979419140

1710900400

1431590230

1152280060

883761320

636257470

419292930

241256180

108962860

27524700

0

27524700

108962860

241256180

419292930

636257470

883761320

1152280060

1431590230

1710900400

1979419140

2226922990

2443887530

2621924280

2754217600

2835655760

2863224150

2835655760

2754217600

2621924280

2443887530

2226922990

1979419140

1710900400

1431590230

1152280060

883761320

636257470

419292930

241256180

108962860

27524700

0

27524700

108962860

241256180

419292930

636257470

```
883761320
1152280060
];
n=[0:127];
N1=128;
X1=abs(fft(x,N1));
F1=[0:N1-1]/N1;

subplot(2,1,1)
plot(n,x)
xlabel('Sample Points')
ylabel('Output Values')
title('Output signal')

subplot(2,1,2)
plot(F1,20*log10(X1))
xlabel('frequency')
ylabel('FFT (dB scale)')
title('FFT of output signal')
```

APPENDIX L

Printed example of a result for the parallel-implemented multiplier

This is a printed example of a result of one run for the parallel-implemented multiplier along with an explanation as to what it means.

RESULT 1 3 3 duration 0.0 highlow 0.0 node 48.0 phase 0.0

The first number after the RESULT is the number of clock cycles that an error appeared in the output. The second number after the RESULT is the highest number of errors for one clock cycles. The third number after the RESULT is the total number of errors. For this example, there was an error in one clock cycle and it had three errors in it and it had a total number of three errors. The duration is the length that the SET lasted. A duration value of 0.0 corresponds to a duration of 1 clock cycle, a duration value of 1.0 corresponds to a duration of 6 clock cycles and a duration value of 2.0 corresponds to a duration of 36 clock cycles. For this example, the SET is applied for 1 clock cycles. The highlow is the value of the SET. When highlow has a value of 0.0 it means that the SET is held low and a highlow value of 1.0 means that the SET is held high. For this example, the SET is held low. The node value is the node that the SET is applied to. For the parallel implemented multiplier the values for the nodes go from 1 to 746. For the serial implemented multiplier there are two values for node, node1 and node2. Node1 is for the inverters that the input values pass through before they reach the multipliers. The values

for node are 1 through 64. There is also another variable, select, that is used to select if the SET is applied to the inverters or one of the multipliers. The select has values are 1 through 48. A value of 1 for the select applies the SET to the inverters. A value of 2 for the select applies the SET to the first multiplier that the inputs are loaded into and a value of 48 for the select applies the SET to the last multiplier that the inputs are loaded into. The variable node2 is for the nodes of the AND gates and ADDERS in the serial multipliers. The values for the Node2 go from 1 through 79. The phase is the value that the sinusoidal wave starts at. The values for the phase range from 0 to 31. Each value selects a different value from the table. For this example, the phase starts at 0 or the first value in the table.