DANNA
A Neuromorphic Computing VLSI Chip

A Thesis Presented for the

Master of Science

Degree

The University of Tennessee, Knoxville

Christopher Paul Daffron

August 2015

© by Christopher Paul Daffron, 2015
All Rights Reserved.

Abstract

Dynamic Adaptive Neural Network Arrays (DANNAs) are neuromorphic systems that exhibit spiking behaviors and can be designed using evolutionary optimization. Array elements are rapidly reconfigurable and can function as either neurons or synapses with programmable interconnections and parameters. Currently, DANNAs are implemented using a Field Programmable Gate Array (FPGA) and are constrained by this technology. To alleviate these constraints and introduce new and improved features, a semi-custom Very Large Scale Integration (VLSI) implementation has been created. This implementation improves upon the FPGA implementation in three key areas. The density of the array is improved, with 5,625 elements on a single chip using a 130 nm process, compared to approximately 4,225 in the FPGA implementation using a 28 nm process with twice the die size. The speed of the array is improved, with clock speeds ten times faster than is currently implemented on the FPGA, allowing for shorter running times on larger array sizes. Finally, the power consumption of the array is improved by eliminating unused transistors from the design. In addition to alleviating the effects of these three constraints, the VLSI design allows for near real time monitoring of the individual elements in the array.

Table of Contents

1 Introduction **1**

2 Related Work **3**

3 Completed Work **8**

3.1 DANNA Overview 8

3.2 Implemented FPGA Designs 12

3.3 Challenges Overcome 13

3.4 Transitioning to VLSI 15

4 VLSI Design **16**

4.1 General 16

4.2 Areas of Improvement 18

4.2.1 Density 18

4.2.2 Speed 21

4.2.3 Power 23

4.3 Monitoring 23

4.4 Conditions Affecting Density 26

4.4.1 Investigated Solutions 28

5 VLSI Preparation **29**

5.1 Tools and Resources 29

5.2 Component Creation 33

6 VLSI Element Implementation **34**

6.1 Design Flow Stages . . . 34

6.2 Element Components . . . 35

6.2.1 Accumulator . . . 36

6.2.2 Input Multiplexer Control . . . 36

6.2.3 LTP/LTD State Machine . . . 36

6.2.4 Monitoring . . . 37

6.2.5 Output Control . . . 37

6.2.6 Registers . . . 37

6.2.7 Synapse Distance . . . 37

6.3 Accumulator Design Flow . . . 38

6.4 Input Multiplexer Design Flow . . . 42

6.5 LTP/LTD State Machine Design Flow . . . 45

6.6 Monitoring Design Flow . . . 51

6.7 Output Control Design Flow . . . 58

6.8 Registers Design Flow . . . 61

6.9 Synapse Distance Design Flow . . . 64

6.10 Combined Element Design Flow . . . 68

6.11 ASIC Element Comparison . . . 74

7 VLSI Element Testing **77**

7.1 Accumulator Testing . . . 77

7.2 Input Multiplexer Testing . . . 81

7.3 LTP/LTD State Machine Testing . . . 84

7.4 Monitoring Testing . . . 85

7.5 Output Control Testing . . . 87

7.6 Registers Testing . . . 90

7.7 Synapse Distance Testing . . . 91

7.8 Combined Element Testing . . . 95

8 VLSI Array Implementation **104**
8.1 Preparation for Encounter . . . 104
8.2 Array Layout . . . 108
8.2.1 Layout Planning . . . 108
8.2.2 Initial Attempt . . . 110
8.2.3 Alternate Strategies . . . 111
8.2.4 Strategy 2 . . . 114
8.2.5 Strategy 1 . . . 116

9 VLSI Supporting Component Implementation **122**
9.1 Clocking Element . . . 123
9.2 FIFO Logic . . . 124
9.3 Random Number Generator . . . 125
9.4 Programming Logic . . . 126

10 Analysis **129**
10.1 Layout Analysis . . . 129
10.2 Delay Modeling . . . 132

11 System Structure **135**
11.1 Chip Floorplan . . . 136
11.2 External Support Chips . . . 137
11.3 Communication Chips . . . 138
11.4 External Processes . . . 138

12 Future Work **139**

13 Conclusion **142**

Bibliography **144**

Appendix **150**

A VHDL Code **151**

A.1 Accumulator 151

A.2 Input Multiplexer Control 155

A.3 LTP/LTD State Machine 157

A.4 Monitoring 160

A.5 Output Control 163

A.6 Registers 165

A.7 Synapse Distance 168

A.8 Clocking Element 170

A.9 Random Number Generator 172

A.10 FIFO Logic 173

A.11 Programming Interface 180

B Element RTL Schematic **199**

Vita **201**

List of Tables

5.1 Available Standard Cells 32

6.1 Multiplexer Truth Table 46

8.1 Density/Process Extrapolation 121

10.1 Element Output Delays 134

11.1 Required Clocks 137

List of Figures

3.1 Image Caption 10
3.2 Relationship between the 4 array clocks 10
3.3 Block diagram of element parts 11

4.1 Virtex-7 2000T Die Size[30] 20
4.2 Combine pin input with programming interface input 21
4.3 Illustration of Stacked Silicon Interface and Super Logic Regions [31] 22
4.4 Current array monitoring architecture using shift registers 25
4.5 Synapse Distance Layout with area of $2628\lambda \times 1886\lambda$ 27

6.1 Vivado RTL Elaboration for the Accumulator 38
6.2 Schematic of the Accumulator 40
6.3 Cadence Viruoso Layout and position of the Accumulator 41
6.4 Vivado RTL Elaboration for the Input Multiplexer 43
6.5 Schematic of the Input Multiplexer 43
6.6 Cadence Virtuoso layout and position of the Input Multiplexer 44
6.7 Vivado RTL Elaboration for the LTP/LTD State Machine 46
6.8 Schematic for the LTP/LTD State Machine 48
6.9 Cadence Virtuoso Layout and position for the LTP/LTD State Machine 50
6.10 Vivado RTL Elaboration for the Monitoring 51
6.11 Schematic for the 8-bit up/down counter 52
6.12 Cadence Virtuoso Layout for the 8-bit up/down counter 53

6.13 Schematic for the 16-bit up counter 54
6.14 Cadence Virtuoso Layout for the 16-bit up counter 54
6.15 Schematic for the Monitoring 56
6.16 Cadence Virtuoso Layout and position for the Monitoring 57
6.17 Vivado RTL Elaboration for the Output Control 58
6.18 Schematic for the Output Control 59
6.19 Cadence Virtuoso Layout and position for the Output Control 60
6.20 Vivado RTL Elaboration for the Registers 61
6.21 Schematic for the Registers 63
6.22 Cadence Virtuoso Layout and position for the Registers 64
6.23 Synapse Distance timing diagram with 0 distance 65
6.24 Vivado RTL Elaboration for the Synapse Distance 65
6.25 Schematic for the Synapse Distance 67
6.26 Cadence Virtuoso Layout for the Synapse Distance 68
6.27 Vivado RTL Elaboration for the Element 69
6.28 Schematic for the Element 70
6.29 Cadence Virtuoso Floorplan for the Element 71
6.30 Element Routing Congestion 72
6.31 Cadence Virtuoso Layout for the Element (top level) 72
6.32 Cadence Virtuoso Layout for the Element (all levels) 73
6.33 ASIC Element Layout from Cadence Encounter. Synthesized with Synopsys Design Compiler 75
6.34 Custom Element Layout from Cadence Virtuoso 76

7.1 Accumulator - Neuron Operation Test 78
7.2 Accumulator - Neuron Refractory Test 79
7.3 Accumulator - Overrun Test 79
7.4 Accumulator - Potentiation / Depression Test - Schematic Level . . . 80
7.5 Accumulator - Potentiation / Depression Test - Layout Level 81

7.6 Input Multiplexer - Input on Enabled Input Test 82
7.7 Input Multiplexer - Input on Multiple Enabled Inputs Test 82
7.8 Input Multiplexer - Synapse Operation Test 83
7.9 LTP/LTD State Machine - Potentiation Test 84
7.10 LTP/LTD State Machine - Depression Test 85
7.11 Monitoring - Synapse, Single Fire Sent, Single Fire in Distance Register Test 86
7.12 Monitoring - Reset After Partial Output Test 87
7.13 Output Control - Neuron Fire Test 89
7.14 Output Control - Synapse Fire Test 89
7.15 Registers - Normal Programming Test 90
7.16 Registers - Wrong Address Programming Test 91
7.17 Synapse Distance - Single Input with Zero Distance Test 92
7.18 Synapse Distance - Single Input with One Distance Test 92
7.19 Synapse Distance - Disabled Input Test 94
7.20 Synapse Distance - Multiple Stored Fires Test 94
7.21 Synapse Distance - Reset Stored Fires Test 95
7.22 Combined Element - Neuron, Fire Causes Output Test 96
7.23 Combined Element - Neuron, Two Inputs Cause Fire Test 96
7.24 Combined Element - Neuron, Zero Threshold, Zero Weight Input Causes Fire 97
7.25 Combined Element - Neuron, Zero Threshold, Negative Input Test . . 97
7.26 Combined Element - Synapse, Zero Distance Test 98
7.27 Combined Element - Synapse Depression Test 99
7.28 Combined Element - Synapse Potentiation Test 100
7.29 Combined Element - Neuron Refractory Period Test 101
7.30 Combined Element - Wrong Programming Address Test 101
7.31 Combined Element - Neuron with Multiple Inputs Test 102
7.32 Combined Element - Monitoring Test 103

8.1 Single pad measurement . . . 109
8.2 Floorplan for the full chip (Not to Scale) . . . 109
8.3 Pin placement guide for the array . . . 110
8.4 Placement for a single row of an 80 × 80 array . . . 112
8.5 Placement for a single row of a 75 × 75 array . . . 113
8.6 Placement for a 75 × 75 array using Strategy 2 . . . 114
8.7 Routing of a 75 × 75 array using Strategy 2 . . . 115
8.8 DRC violations of a 75 × 75 array using Strategy 2 . . . 116
8.9 Manually created clock tree to drive the array rows . . . 117
8.10 Placement of a 75 × 75 array using Strategy 1 with missing global signals 118
8.11 Placement of a 75 × 75 array using Strategy 1 . . . 119
8.12 Routing of a 75 × 75 array using Strategy 1 . . . 120
8.13 Wasted space by inverters and buffers . . . 120

9.1 Clock Element layout from Cadence Encounter . . . 124
9.2 FIFO Logic layout from Cadence Encounter . . . 125
9.3 Random Number Generator layout from Cadence Encounter . . . 126
9.4 Programming Logic layout from Cadence Encounter . . . 128

10.1 Position of several rows within the array, shown in red . . . 130
10.2 Global Network Clock tree from pin to each row . . . 131
10.3 ModelSim simulation of the delay model . . . 133
10.4 ModelSim simulation showing the difference in signal delays . . . 133

11.1 System diagram showing all components . . . 135
11.2 Full chip floorplan showing all components . . . 136

B.1 RTL Schematic of a single element . . . 200

Chapter 1

Introduction

Dynamic Adaptive Neural Network Arrays (DANNAs) are neuromorphic systems which exhibit spiking behvaiors and can be designed using evolutionary optimization. DANNAs are a new, distinct architecture from the von Neumann architecture and are intended to leverage select characteristics of the brain to implement a computational structure well sutited to certain applications, not to simulate the brain and its functions. DANNAs are comprised of an array of generic elements which can be programmed to act as neurons or synapses and are connected to their nearest neighbors. In the current implementation, the elements are connected to their 16 nearest neighbors. They are connected to their eight immediate neighbors and then eight additional neighbors one layer out in the same direction as the inner eight connections. The choice of connecting to the 16 nearest neighbors is arbitrary and can be expanded in multiples of eight by adding additional connections. Additional details about the DANNA architecture and element connections can be found in Section 3.1. Currently, DANNAs are implemented using Field Programmable Gate Arrays (FPGAs), but array sizes are limited by the resources available on the device. Because of this, a custom Very Large Scale Integration (VLSI) array has been implemented. Creation of this design allows for improvements in the density, speed, and power consumption of the architecture. Transitioning from an

FPGA implementation to a VLSI one also allows for improvements in the array monitoring functionality. Implementing these areas of improvement presented many challenges. These challenges included optimizing the element and ensuring that it worked properly, many issues with multiple tools involved in the process, achieving the desired densities, creating all of the clock trees and global signal distribution trees that were required, and the insertion of additional buffers to meet the fanout requirements for the components in the design. These improvements allow for complex networks to be implemented on the device with quicker runtimes and more detailed information about the inner workings of the elements.

Chapter 2

Related Work

Many different groups are currently working to implement neuromorphic hardware devices and many of them are taking vastly different approaches. Some groups are working within the current von Neumann architecture while others are inventing new architectures. Groups are using different combinations of digital logic, traditional analog devices, and new types of silicon devices such as memristors. Groups are also taking different approaches in their design inspiration, with some creating biologically inspired devices and others creating computationally inspired devices.

One of the more established projects in neuromorphic hardware is the SpiNNaker project at the University of Manchester, which is associated with the Human Brain Project. They have created a digital, computationally inspired system which consists of a large array of chips where each chip contains multiple ARM cores and a packet router [11]. The chips are then networked together on multiple printed circuit boards to eventually create a system with one million cores. This system utilizes the von Neumann architecture through its use of ARM cores, but has the flexibility to implement different neuron architectures through software.

The Human Brain Project has developed another neuromorphic hardware device which does not utilize the von Neumann architecture called BrainScaleS. It is

biologically inspired and is focused on replicating the biological functions of neural networks and contains dedicated hardware for neurons and synapses on each chip [23]. Each chip has 512 analog neurons and their operation can be modified by adjusting parameters associated with them. There are approximately 3,000 parameters that define the operation of each neuron [24], though some of those are shared among neurons on the chip. Each of the neurons on the chip can be connected to up to 16,000 synapses. Because of the large number of synapses compared to neurons, the synapses dominate the physical area of the chip [18]. The final device is a wafer-scale device and is made up of 20 wafers in total, which allows for very large simulations.

The IBM TrueNorth project has also created a custom digital chip which does not utilize a von Neumann architecture. The basic building block of this chip is a neurosynaptic core which contains 256 digital integrate-and-fire neurons and a 1024×256 bit SRAM crossbar memory to act as the synapses [16]. More recently, the IBM TrueNorth group combined 4,096 of these cores to create a single 5.4 billion transistor chip that contains a total of 1 million neurons and 256 million synapses [17]. The chip is physically quite large, taking up 4.3 cm^2 in a 28 nanometer process. Multiple chips can be tiled together to create even larger networks. Each core has 104,488 bits of memory, with 65,536 bits dedicated to synapse state and 31,232 bits dedicated to neuron state and parameters. This works out to 122 bits of combined state and parameters for each of the neurons, significantly more than the 41 bits required for DANNA neurons.

Researchers at Stanford University have created Neurogrid, a system with sixteen mixed-analog-digital chips which are able to simulate 1 million neurons with billions of non-plastic synapse connections in real-time [1]. Though the chips are a mixture of analog and digital circuitry, they are primarily analog except for the circuits implementing the axonal arbor. This allows the transistor count to be reduced. The full system that does this is comprised of sixteen Neurocore chips on a single circuit board and only consumes three watts of power. This is especially impressive considering that the other system capable of simulating 1 million neurons, the IBM

TrueNorth chip, uses approximately 20 watts per cm^2 and has an area of 4.3 cm^2. The creators of the SpiNNaker system are able to simulate ten thousand neurons and four million synapses using less than two watts [26]. For comparison, they estimate that it would take approximately 8.4 GW to simulate 1.6×10^9 neurons using a supercomputer.

Several different groups are working on implementing neuromorphic hardware devices utilizing memristors. Because the resistance of memristors is related to the history of inputs to that device, they present possibilities for usage in replacing the charge storage logic for neurons. When successfully implemented, they can be much smaller than the digital logic required to implement the same functionality, allowing for increased network density. A group at Rochester Institute of Technology is currently working on implementing Neural Logic Blocks using memristors which can be used to implement nonlinearly separable functions [27]. They have tested their device on several problems and are achieving competitive results with significantly lower power consumption than other devices. Other groups, such as one at The University of Dayton, are implementing new types of neuromorphic circuits which allow for both reading from and writing to the memristors [32]. Another analog device which has been used to implement analog neural networks is the FPAA (Field Programmable Analog Array). This device was used by Farquhar et. al. to create a reprogrammable neural network platform [10].

One of the challenges with working with neuromorphic hardware and an area where DANNA excels is the ease of programming the system with different networks and providing those networks with external stimulus. For example, the 3,000 parameters that define the operation of each of the neurons in the BrainScaleS system add complexity to the networks that are created and also increase the amount of data required to program the device. Many smaller groups are also working on neuromorphic systems. In the FPGA architecture described by Leiner et. al., all of the inputs must be stored on the device before the processing is started by the network [15]. This is acceptable since the given application of their system is processing static

images, but streaming data is required to process other kinds of data. Neural networks developed using less common technologies can also require complex programming. The neural networks using synapses made out of nanoscale phase change devices by Jackson et. al. allow for compact design and low power consumption but are complex to program because the phase change devices are programmed by heating them [13]. In the VLSI device described by Indiveri et. al., only some of the parameters are programmable individually for each of their synapses, with the parameters for the synaptic weights being individual but the other bias parameters being global [12]. Programming the SpiNNaker system is much different than any of the systems which have dedicated neuromorphic VLSI or FPGA implementations because much of the behavior of the system is defined by software. To create a network, the software that implements the element models and the routing tables must be loaded into the system using a host machine [14].

The connectivity scheme of these neuromorphic hardware devices can also vary widely. The SpiNNaker system mentioned previously uses an integrated network to communicate events between neurons [19]. The system uses an on-chip router to facilitate communication both within the chip and between the chips of the system, but this comes with a large cost of chip area, consuming approximately 10 percent of the available area. Once all of the chips in the system are connected, they form a three dimensional torus where each chip is connected to six of its nearest neighbors, though any neuron can be connected to any other neuron due to the routing capabilities of the chips [14]. Another popular method of connecting elements in a neuromorphic chip is to utilize a crossbar array. The neurosynaptic core used to make the IBM TrueNorth device uses this type of connection [16] as well as the nanoscale device described by Rajendran et. al. [21]. In the VLSI device proposed by Indiveri et. al., there are a total of 32 integrate-and-fire neurons and each neuron is connected to 2 inhibitory and 6 excitatory synapses [12]. This is similar to DANNA in that there are a relatively small number of connections, but the inhibitory or excitatory nature of the synapses is programmable in our implementation.

Closely related to the on-chip connectivity used by these devices are the communication protocols used to enable off-chip communication. The FPGA-based design implemented by Stepanova et. al. utilizes the PCI bus to communicate with the host machine, taking advantage of the speed of the protocol to allow for real-time data transfer with the device [28]. The Real-Time Computing Platform for Spiking Neurons (RT-Spike) proposed by Ros et. al. uses a system that is similar to the current DANNA implementation, with the simulation of the network being performed on an FPGA which communicates with a host machine over PCI and the host machine provides the stimulus [22]. Other devices use on-chip storage that is loaded with the input stimulus before the network is started. This approach is utilized by the Spikey chip, which stores all of the input before starting processing on it [20].

As has been shown, many different approaches to neuromorphic hardware exist. Some implementations are computationally inspired and others are biologically inspired. Some groups are working within the current von Neumann architecture and others are inventing new architectures. Various groups are using different combinations of digital logic, traditional analog devices, and new silicon devices such as memristors. We believe that our design, which is computationally inspired, uses a custom architecture other than the von Neumann architecture, and utilizes digital devices, is well suited to produce strong results in the field of neuromorphic computing.

Chapter 3

Completed Work

3.1 DANNA Overview

A Dynamic Adaptive Neural Network Array (DANNA) is an array of generic, dynamically programmable neuromorphic elements that collectively form an Artificial Neural Network (ANN) [9, 7, 3, 25]. Each of the neuromorphic elements can be programmed to act as a neuron, synapse, or pass-through whose operation can be altered by multiple dynamically programmable parameters. Neurons have a programmable threshold and programmable input enables for each of the 16 connected elements. The choice of connecting to 16 elements is arbitrary and can be expanded in multiples of eight. The connectivity scheme used for the 16 connections is to have the element connect to its eight nearest neighbors and then connect to eight additional neighbors one layer out in the same direction as the inner eight connection as illustrated in Figure 3.1. The directionality of these connections and the referenced placement of the connected elements is for the theoretical grid layout, which does not necessarily reflect the actual placement of the elements. To increase the number of connections, additional connections are simply added to further out layers. Synapses have a programmable weight, refractory period, delay, and input enable to a connected element. Only one input to the synapse can be enabled at the same time. The

synapse also uses one port as the output, designated during the programming of the element. This port provides feedback which is used for long-term potentiation (LTP) and long-term depression (LTD) [5]. The pass-through is a specially programmed synapse with one input and multiple outputs. Because of this, they can be chained together to create long distance connections and can also be used to increase the fanout of a connected element. The array utilizes four different clocks. The Global Network Clock (GNC) operates at 0.5 MHz in the FPGA implementation and defines one cycle of the network. It is also used to drive fires out for a synapse. The Acquire Fire Clock (AFC) controls the sampling of inputs and currently operates at 8 MHz in the FPGA design. This frequency was chosen to have 16 cycles of the AFC for each cycle of the GNC, allowing each of the 16 inputs to be sampled during a GNC cycle. Two additional clocks are used to control the accumulator, the Accumulator Clock (AC) and the Accumulator Enable Clock (AEC). In the FPGA design, the AC operates at 16 MHz and the AEC operates at 8 MHz with a 90 degree phase shift from the AFC. When both the AC and the AFC are high, the accumulator accumulates charge. During the other half of the cycle, the accumulator may be reset to the initialization value if the element has fired. This means that the accumulator is active a maximum of 32 times during each GNC cycle. The relationship between these clocks is shown in Figure 3.2. The input that is sampled at a given time is controlled by the Global Input Select (GIS), a counter that sequentially goes through all inputs in each GNC cycle starting with a random input determined using a linear-feedback shift register. The GIS starting point is randomly determined to prevent a bias from forming where certain inputs are more likely to cause a fire. While we felt randomness was important, the number of nonrepeating sequences was arbitrarily determined. Future analysis will be required to determine the appropriate level of randomness.

Internally, the functionality of the element can be divided into four major components as illustrated in Figure 3.3. The input sampling component is responsible for sampling the output of the connected elements and samples them sequentially in

a neuron, starting with the input indicated by GIS. In a synapse, this component always passes through the single enabled input. The input sampling component references the programmable input enable parameter to determine if an input is enabled and fires should be passed along to the next component or disabled and the fires should be ignored. The accumulate and fire component fulfills different functions in a neuron and synapse. In a neuron, this component implements the neuron's threshold and charge logic and monitors for when the neuron enters a fire condition. In a synapse, this component stores the weight and is also used to increment and

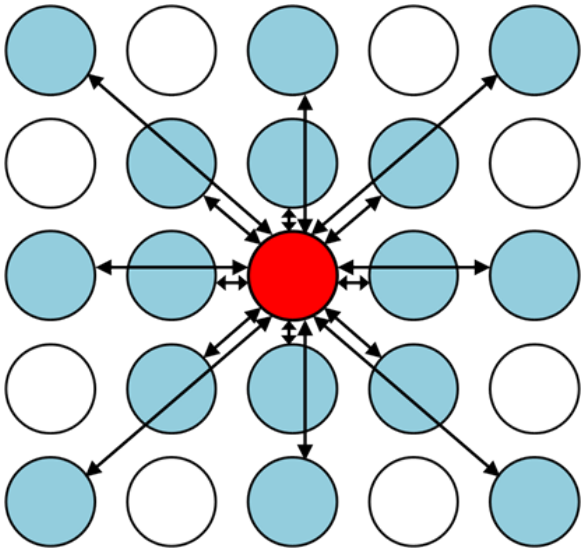


Figure 3.1: Connection scheme of a single element[8]

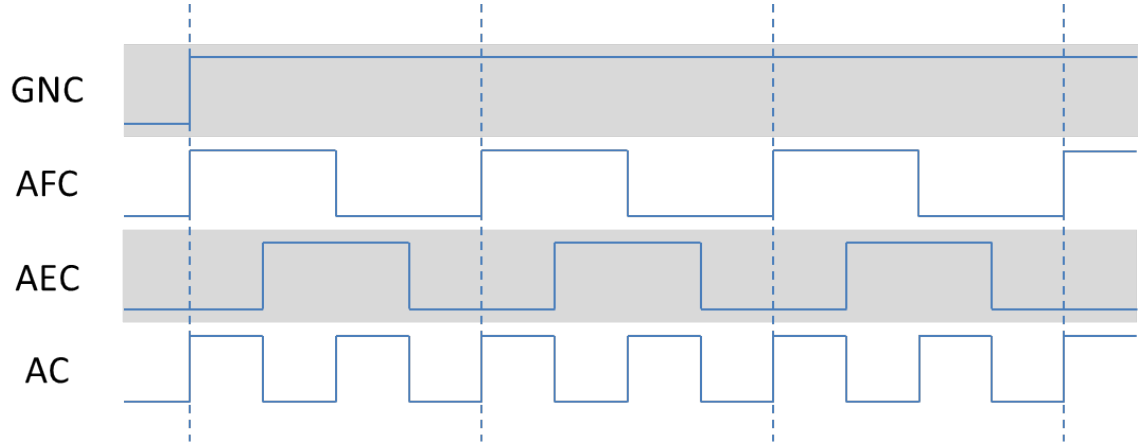


Figure 3.2: Relationship between the 4 array clocks

decrement the weight during potentiation and depression. The LTP/LTD component of the element is only used when it is acting as a synapse. It monitors the state of one element determined by a programmable parameter and controls the accumulator when potentiation or depression conditions are met. The synapse delay component implements the programmable delay and is implemented using an addressable shift register. A gate level schematic of the current element design is shown in Appendix B in Figure B.1.

The programming interface has the capability to provide stimulus to the array, to monitor the output of the array in real time, and to program the elements in the array to form a network. It communicates through first-in, first-out buffers (FIFOs), making it independent of the communication protocol used.

The DANNA architecture is new and distinct from the von Neumann architecture and is intended to leverage select characteristics of the brain to implement a computational structure well suited to certain applications, not to simulate the brain and its functions. The traditional von Neumann architecture has bottlenecks in its structure, mostly concerning data movement. These bottlenecks are partially caused by the separation between the computing components and the memory components. In the DANNA architecture, the memory components are distributed throughout the array and are incorporated into each element, close to where they are used. The

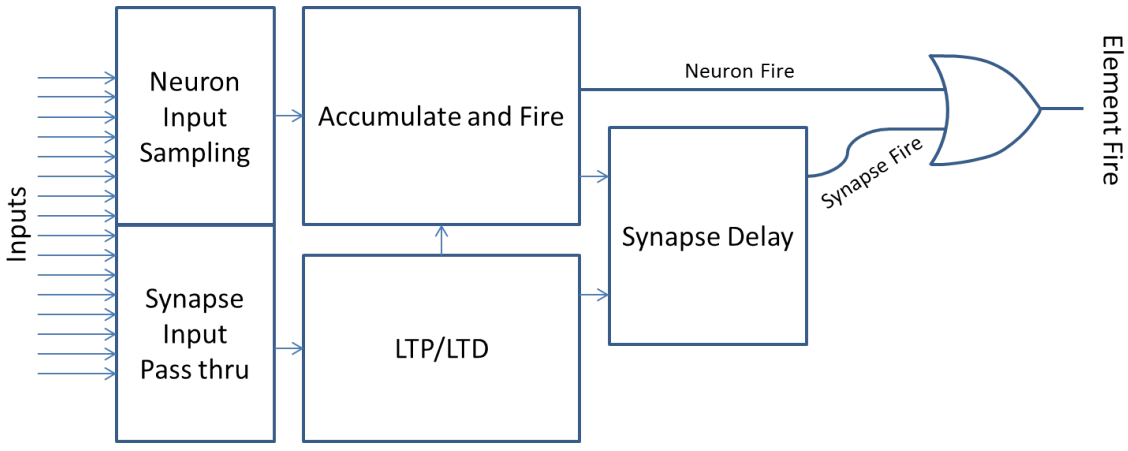


Figure 3.3: Block diagram of element parts

DANNA architecture also includes spatio-temporal capabilities, unlike several of the architectures described in Chapter 2. Because of these capabilities, the timing of the input influences the results of the network. This is especially useful in applications such as handwritten digit recognition where the input image can be scanned with a few input elements instead of being input all at once using many more input elements. Additionally, visualization techniques have been developed to assist with analysis and creation of networks [2]. This visualization technology has also been used to conduct research into the possibility of identifying useful substructures that can be reused in other arrays [4].

3.2 Implemented FPGA Designs

Three different FPGAs have been used for creating DANNA arrays. The Xilinx XC7VX485T FPGA is a Virtex-7 family FPGA with 485,760 logic cells. This device has been used to support arrays as large as 16×16, though those arrays did not approach the maximum resource utilization and the device could theoretically support up to a 30×30 array. The Xilinx XC7VX690T FPGA is a Virtex-7 family FPGA with 693,120 logic cells. This device has been used to support arrays as large as 50×50. These arrays did approach the maximum resource utilization of the device and required extensive design work to achieve the desired functionality. The Xilinx XC7V2000T FPGA is a Virtex-7 family FPGA with 1,954,560 logic cells. This device is unique from the other two because it includes Xilinx Stacked Silicon Interconnect (SSI) technology to allow for such a large number of logic cells. This device has been used to support arrays as large as 80×80, which approaches the maximum resource utilization of the device. Because of the size of this device and the SSI technology present, this device presented many unique challenges while working to get the neural network functional.

3.3 Challenges Overcome

Due to the massively parallel nature of DANNA and fine granular control over the number of elements present in the array, we have been able to achieve very high utilization on these FPGAs. Because we are achieving such high utilization on FPGAs with many logic cells, we have encountered and solved several issues that are unique to high utilization designs on large FPGAs. The majority of these issues involve the timing of the design. Because all of the elements operate in parallel, it is crucial that they all receive clock signals that are properly aligned with each other. Achieving this becomes quite difficult with large array sizes due to the load on each of the clocks and the long routing delays that can be present across the device. For the 16×16 array, timing did not have to be considered as carefully due to the relatively small size of the array, but moving up to a 50×50 array introduced these issues. The primary way of solving clocking issues caused by a large load is to add clock buffers to the device to distribute the load. This is a function that the design software claimed to automatically implement but failed to actually implement as it did not consider the load on the clocks to be excessive, even though some clocks were driving approximately 90,000 endpoints. Upon manually adding a clock buffer for each clock, we discovered the second important consideration in large designs such as this one: The placement of clocking resources is very important to the overall timing of the device. When the clock buffer was placed near one edge of the device, which is where the board clock enters the device and is the default tool placement, the operation of the device was inconsistent, but when it was placed in the center of the device, the array worked correctly as long as it was not too large. In our experience, a single clock buffer per clock, placed in the center of the device, was sufficient to achieve adequate timing in the 50×50 array, but not in the 80×80 array.

For the 80×80 array, the design was divided up into 4 equally sized clocking regions, forming a 2×2 grid, each with a dedicated set of clock buffers. This led to the discovery of the next challenge with implementing these large designs: The tools

optimize away the additional clock buffers unless explicitly prevented from doing so, as the additional distributed clock buffers are considered to be redundant. In the Xilinx Vivado Design Suite, this is achieved by placing `DONT_TOUCH` attributes on specific signals and components within the design. The designer must be careful when assigning these attributes because too few of the attributes will result in unconnected signals in the final netlist, but too many of these attributes could negatively affect the rest of the design. For our design, necessary resources were retained by placing the attribute on the component creating the desired signal and the output signal from the component.

Currently, the array is implemented in a square layout that defines the connections between elements. When the layout is implemented on an FPGA, its placement is physically different than its logical square structure, but the connection scheme of the layout is preserved. The physical structure presents an important challenge when routing fire events over long distances. For example, synapses or pass-throughs must be chained together to route a fire event across the array. To alleviate this problem, we propose expanding the connections in the current array to form a torus. This would be achieved by connecting the top elements of the square directly to the bottom elements of the square and the left elements of the square to the right elements of the square. Only some of the elements would have the wrapping connection, with the connections distributed evenly across the width and height of the array. For the elements that are connected to the opposite side of the array, they would only have a single connection straight across the array and would not be fully connected like the non-wrapping connections. Routing issues with global clock signals have already been encountered, and the likelihood of these additional connections increasing routing challenges in the FPGA implementation is high.

3.4 Transitioning to VLSI

The challenges that have been overcome while working with the FPGA design have shown areas that need to be focused on in the VLSI design. In order to get the improvements that will be discussed in the next section, the timing of the array will be very important. Correct and accurate clock distribution will be especially important due to the large capacitive loads that will be on each clock. The delay within a single element will need to be considered carefully so that the sampling of nearby elements will line up closely with the firing of the elements. For the inputs and outputs that are driven to the array and sampled from the array, special consideration needs to be taken to ensure that the signals are driven based on the loaded clocks and not the free running clocks that drive the programming interface and have much less load, and therefore delay, on them. The other important consideration from the FPGA design that will be reduced, but not eliminated, with the VLSI design is routing congestion. A large amount of the chip area will have to be dedicated to routing of signals between elements. Each of the elements will have several hundred bits of data going in and out which will need to be routed to all of the surrounding elements.

Chapter 4

VLSI Design

4.1 General

The VLSI chip is designed to improve upon the performance of the FPGA designs in several ways, including improvements in the density, speed, and power consumption of the array. The VLSI chip also enables improvements in the real-time monitoring capabilities of the array. The design of the array was divided into several major pieces. First, a fully functional element was designed at the schematic level and was fully tested to ensure that the design was correct and functioned as expected. At this point, the improvements in the speed and power consumption of the element started to become evident. The logic needed to implement the improvements in the monitoring functionality also needed to be incorporated into the element at this point.

After completing the design of the element at the schematic level, the layout of the element was completed. To achieve the density goals associated with the development of this chip, this layout needed to be made as compact as possible in a square or rectangular shape to facilitate combining the elements to form an array. To assist with the layout of the element, basic components of the cg_lib13 library [29] created by the University of Washington and provided to us by Dr. Holleman's* research

*UTK EECS Professor

group were utilized. This library provides all of the necessary logic gates in multiple sizes and designs. The library also provides clock inverters, buffers, basic flops, and small adders, all of which will be necessary to complete the layout of the element in a timely manner while maintaining the improved density targeted by this design.

Following the completion of the layout of a single element, the layout of the array was implemented. This layout includes approximately 5,625 elements, creating a 75×75 array. The layout originally targeted 90,000 elements in a 300×300 array, but density issues presented themselves during development. These issues are discussed in Section 4.4. The approach taken when implementing the array was to import the element as a basic block into Cadence Encounter and use the Application Specific Integrated Circuit (ASIC) tools of that program to place the components in the chip, run the fanout correction flow, implement the clock tree in the design, and route all of the signals between the instances.

Following the completion of the layout of the array, all of the supporting components were designed. The components to be designed include the clocking logic, the programming logic, the FIFO logic, and the random number generator. The clocking logic manages the four necessary clocks for the elements, as well as the clocks for the supporting logic. The programming logic facilitates the programming of the individual elements in the array to form a network, send stimulus in the form of input fires to the appropriate elements while the array is running, monitor the internal output elements, and communicate with the elements for the real-time monitoring functionality. It includes logic to interpret incoming programming packets and drive the appropriate signals out to the elements and it includes logic to encapsulate the internal output fires for sending off chip. The programming logic communicates with the off-chip communication logic through the FIFO logic. This logic loads and unloads off-chip FIFOs that form the connection between the communication interface and the programming interface. The off-chip communication logic implements the necessary communication protocol to communicate with the host machine using USB and is implemented on a separate chip from the array. External communication is

accomplished by using a system based on the FX3 SuperSpeed Explorer Development Kit. In conjunction with the off-chip communication logic, it translates between the packets needed by the programming logic and the off-chip communication protocol. Finally, the random number generator uses a linear feedback shift register to generate the random starting point for the counter that creates the Global Input Select signal used by the elements to determine which input to sample.

For this chip, the design was done in the IBM-8RF process provided by the university and MOSIS. The cg_lib13 library mentioned earlier also targets this process, so it was compatible with this design. There are a total of eight metal layers in this process, so there were enough metal layers for the extensive signal routing required and all of the connections to power and ground. Even though this process was used for the initial design of the array, it will likely not be the fabrication process actually used to create the chip. Instead, the design will likely undergo both a die and process shrink and be fabricated in a 65nm or smaller process.

4.2 Areas of Improvement

4.2.1 Density

The first aspect of the array which will be improved greatly by moving from the current FPGA implementation to a full-custom VLSI implementation is the density of the array. The first improvement targeted by the VLSI design over the FPGA design is the density of the array. We estimate that the largest fully functional array able to be achieved with the largest FPGA owned by the research group, the XC7V2000T FPGA, which has 1,954,560 logic cells and is fabricated using an 28nm process is a 70×70 array. This estimation takes into account recent changes made to the design such as the expansion of the programming interface and addition of monitoring. Without these features, the largest array achieved is an 80×80 array. These arrays use most of the available lookup tables on the FPGA, including the

logic used for the programming interface, FIFO logic, clocking logic, and includes the monitoring shift registers. A larger array of 85×85 has been attempted on this board and was able to successfully implement using a previous version of the supporting logic and used over 90 percent of the available lookup tables, but was unable to route using the current version of the programming logic, which introduces important latching features to improve the timing of the array, and did not include the monitoring shift registers. While the 70×70 array achieves high utilization of the lookup tables present in the FPGA, the array uses few of the flip-flops and memory lookup tables. Even though these types of resources have low utilization, they cannot be fully utilized without a significant element redesign due to the high utilization of lookup tables. Because the VLSI implementation eliminated the wasted hardware resources of the FPGA and allows for more compact, optimized components, higher densities are possible. The initial design is an array containing 5,625 elements in a 75×75 configuration. This array has more elements than the FPGA implementation even though it is designed in the 130 nm process, which is nearly five times larger than the 28 nm process that the FPGA is fabricated in. Using the picture in Figure 4.1, we have estimated that the FPGA die size is $600mm^2$. This size is the total combined size of the four dies that are combined on an interposer. Assuming that we are able to achieve quadratic scaling in future revisions, this means that, in the same process size as the FPGA, we could fit the 5,625 elements of the VLSI implementation in $1/33^{rd}$ of the space that the FPGA uses for 6,400 elements. With the same die size and process size as the FPGA, we could fit 185,543 elements in the same space that 6,400 elements occupy in the FPGA. Additional extrapolation to other processes and densities can be found in Section 8.2.5.

One additional area of improvement in the design is in the implementation of off-chip input and output ports. The current FPGA design includes 16 input and 16 output ports driven by the programming interface. In the VLSI implementation of the programming interface, which has also now been implemented on the FPGA, there is room for 32 input elements and 32 output elements. An improvement which the

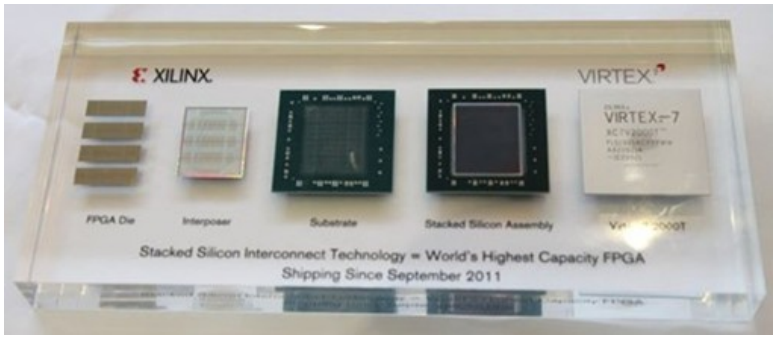


Figure 4.1: Virtex-7 2000T Die Size[30]

completed VLSI chip should allow for is the ability to drive a subset of the external inputs and outputs directly from the pins of the chip in addition to going through the programming interface. Taking this into consideration, there are 5 types of elements when categorized by stimulus connectivity. Elements not connected directly to any external stimulus, either through the programming interface or directly, are called hidden elements, while those that interact with some stimulus are called I/O Elements. I/O Elements with the ability to be driven directly by the pins of the chip are more specifically called External Input Elements and those that drive out to the pins of the chip are more specifically called External Output Elements. Finally, elements that receive stimulus exclusively from the programming interface are called Internal Input Elements and similarly those whose fires are sent off-chip exclusively by the programming interface are called Internal Output Elements. Due to limitations in the number of pins available in a package, not all of the I/O elements are able to have this feature. Instead, the design allows for 16 External Input Elements and 16 External Output Elements. This implementation requires 288 pins of the package be dedicated to this feature. Very few changes to the logic of the design are required to implement this functionality. For the External Output Elements, no extra logic is required, just extra wires, as there is still only a single driver of the signals. For the External Input Elements, logic needs to be added so that a bit being driven from either the programming interface or the external pins will cause that bit to be driven

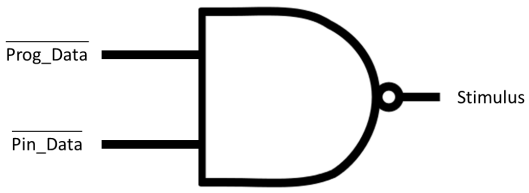


Figure 4.2: Combine pin input with programming interface input

to the element. This can be achieved using a single gate per bit as shown in Figure 4.2.

4.2.2 Speed

The clock rates at which the array can run while maintaining the desired functionality will be improved significantly in the VLSI implementation. The clock speeds at which the VLSI design is intended to operate are much faster than the FPGA clock speeds while maintaining all of the desired functionality.

In the current FPGA implementation, there are four clocks: The Global Network Clock runs at 0.5 MHz and defines how long a single cycle of the network is and is used to determine how long elements should fire and to measure their refractory period. The Acquire/Fire Clock runs at 8 MHz and is used to clock the sampling within an element, with one of the input ports sampled on each cycle. This clock is also used to run the logic that decides if a neuron is ready to fire. The Accumulator Enable Clock is also running at 8 MHz, but has a 90 degree phase shift from the Acquire/Fire Clock. It is used in conjunction with the Accumulator Clock to enable or disable the operation of the accumulator. Finally, the Accumulator Clock is running at 16 MHz and is used to clock the operation of the accumulator in the element.

For this implementation, the VLSI array is designed to operate at the following clock speeds: 5 MHz Global Network Clock, 80 MHz Acquire/Fire Clock, 80 MHz Accumulator Enable Clock, and 160 MHz Accumulator Clock. These speeds represent

a ten times improvement over the clock speeds that are currently used with the FPGA implementation.

In the recent FPGA implementations, the timing of the array gets worse as the array gets larger, with a considerable amount of work required to even maintain the functionality of the array. To combat this gradual worsening of the timing, the current FPGA design has been divided into several different clock regions. Originally, the clock regions were arranged in a 2×2 square grid, using the positioning of elements in the logical layout of the array, not the actual layout on the FPGA. As an optimization to better align with the hardware capabilities of the FPGA, the clocking regions were changed to be four rows, with each row approximately one fourth of the height of the overall array. This change was made because the 2000T FPGA is physically divided into four Super Logic Regions (SLRs) internally as illustrated in Figure 4.3. By changing the layout of the clock regions, the layout on the physical FPGA is allowed to better reflect the internal structure of the FPGA. In the VLSI implementation, the clocks are subdivided into much more than four clocking regions, with trees designed at multiple levels in the hierarchy. The clock tree was designed both using the automated tools present in the ASIC tools and by hand as described in Sections 8.2.5 and 10.1.

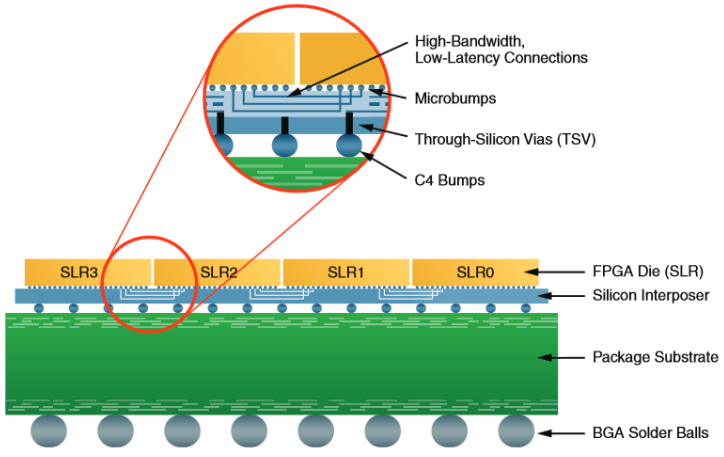


Figure 4.3: Illustration of Stacked Silicon Interface and Super Logic Regions [31]

4.2.3 Power

The power consumption of the current FPGA implementation is unknown, but the power consumption of the VLSI implementation should be an improvement. Even if the power consumption of the FPGA implementation was known, it would not be directly comparable to the VLSI implementation due to the large amount of unnecessary logic that is present on the FPGA and is drawing idle power. With the VLSI design, only the logic that is necessary for the functionality of the array is present, so the idle power draw of the unnecessary components is eliminated.

4.3 Monitoring

In the current FPGA implementation of DANNA, the array monitoring capabilities are implemented utilizing shift registers to gradually shift certain internal values out of all the elements in the array [6]. The structure of this monitoring architecture is shown in Figure 4.4. This architecture functions by capturing three variables from inside the individual elements: the current value in the accumulator, the number of times the element has fired since the last shift register load, and the number of fires that are currently stored in the synapse distance shift register. The current value in the accumulator represents different values in a neuron or synapse. In a neuron, this value is the accumulated charge since the last fire and in a synapse it is the current weight including the effects of potentiation and depression. There is a shift register inside of each element and they are all loaded in parallel when a global signal is sent. Then, after they have all been loaded, the data in each shift register is shifted up each column to the next element one bit at a time when a signal is sent at a maximum rate of one shift per Global Network Clock cycle. In the top element for each column, the bit that is to be shifted out is sent off chip. This is done for every column in parallel. The current design of the off-chip output logic allows for up to 128 columns to be outputted in parallel. Because the information in the shift

registers is only output at the current Global Network Clock rate and only a single bit is outputted at a time per column, dumping a full array can take a relatively long time. The data to be output from each element is 32 bits wide, so outputting a full 80×80 array takes at least 2,560 Global Network Clock cycles. This means that, with the clocks constantly running, the current state of the array can only be captured approximately every 2,500 cycles, during which time many important events in the array might be missed. Luckily, because the monitoring functionality is clocked by the strobe signal, the clocks can be stopped while the data is being outputted and data for every cycle can be captured. In the VLSI implementation, two improvements have been implemented for the monitoring capabilities. First, the clock speeds of the array have been increased by ten times. This greatly reduces the amount of time taken to output the array, though not the number of cycles. This also means that the amount of data produced by the array in a given amount of time is much greater. Second, the clock used to run the monitoring functionality has been changed from the Global Network Clock to the strobe. Making this change allows the clocks to be stopped while the data is shifted out if desired. This means that the full state of the array can be saved for every cycle instead of only periodically as it is in the current implementation. This also means that the frequency of the strobe signal could be increased independently of the clocks to make the array output even faster.

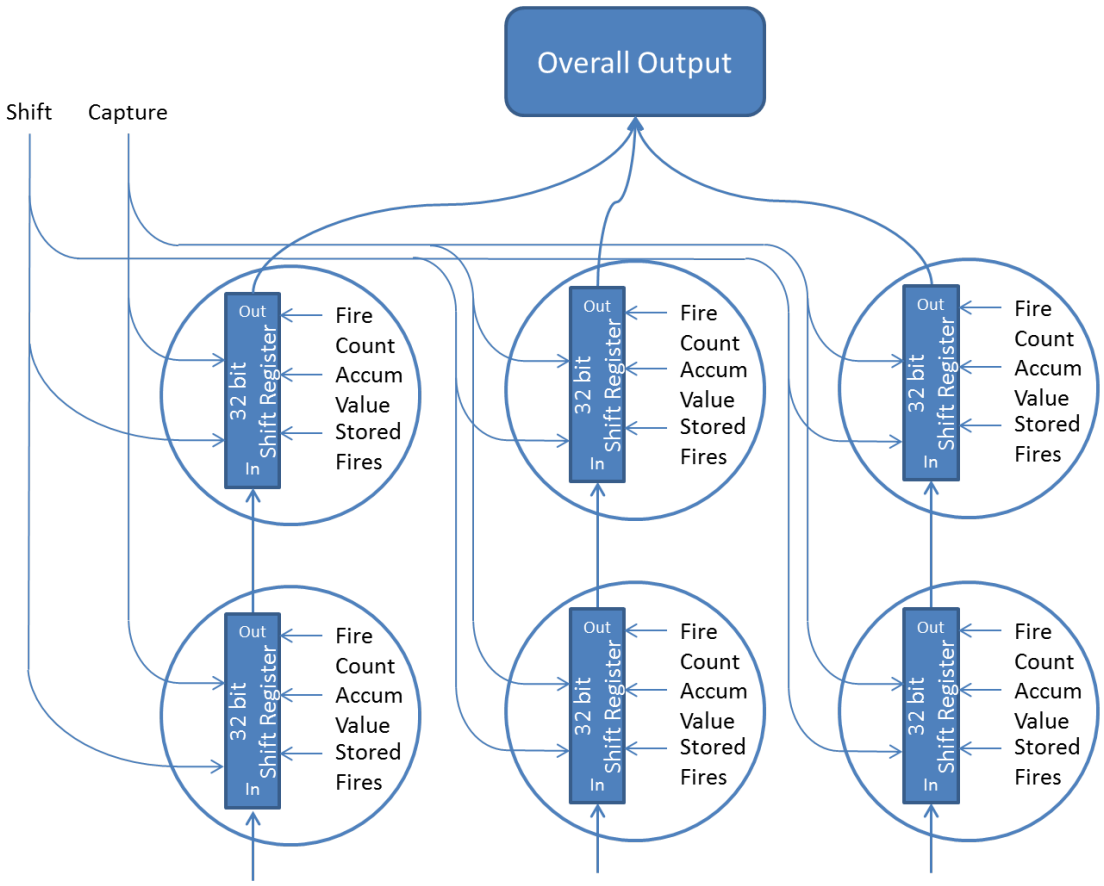


Figure 4.4: Current array monitoring architecture using shift registers

4.4 Conditions Affecting Density

After implementing the layouts for the element, the transistor density was less than expected. Additionally, the original target for the number of elements was a rough estimate that turned out to be flawed. Because of this, the expected density was reduced from 90,000 elements to 5,625 elements. A good illustration of the factors leading to reduced density is provided by the 256-bit shift register that is used in the synapse distance function. In this component, only approximately 11,000 transistors are able to be fit in a space that is larger than $2000\lambda \times 2000\lambda$ as illustrated in Figure 4.5. Ideally, if all transistors were the minimum design size of $2\lambda \times 2\lambda$ and no additional space was necessary for routing, then 1,000,000 transistors would fit in this space. Realistically, a minimum sized NMOS transistor is $18.8\lambda \times 9.6\lambda$ and a PMOS transistor that is approximately sized to create CMOS logic with the minimum size NMOS transistor is $28\lambda \times 14.8\lambda$. Based on this, the upper limit for the number of transistors able to fit in a $2000\lambda \times 2000\lambda$ area is 297,440 transistors, still much larger than the number that has actually been implemented in this space so far. The primary reason for this large difference is the amount of routing space required to connect all of the transistors. Also, to make layout simpler, all of the cells in the standard cell library are the same height. This allows adequate room for routing and allows the layout to be created with consistent patterns, but does mean that some space will be wasted in the cells that do not require very much routing or are near the edge of the design where fewer signals will need to pass over them.

When the target density was originally calculated, we were targeting 100,000 elements, but reduced it to 90,000 to allow for an array size of 300×300. We estimated that each element would occupy approximately $2000\lambda \times 2000\lambda$ of area, and we determined that a die size of $7mm \times 7mm$ would allow this array size with some space included for routing channels. It then came to our attention that those calculations had an error in them and a die size of $7mm \times 7mm$ with $2000\lambda \times 2000\lambda$ elements can actually only support a 53×53 array. This array size is especially

disappointing because the current largest FPGA implementation supports a 70×70 array. Because of this, the allowable die size has been increased to approximately $20mm \times 18mm$ and the target array size was been changed to 100×100 with 10,000 elements. In the end, due to the number of buffers required in addition to the elements, a 75×75 array was implemented. The process used to determine the maximum array size is discussed in Section 8.2.3.

Finally, as the layout progressed, it became apparent that the original estimate of the element occupying $2000\lambda \times 2000\lambda$ was too small. For example, the 256-bit shift register that is part of the synapse distance function occupies $2628\lambda \times 1886\lambda$ when implemented using flops. Because just one piece of the element is larger than the original estimate for a full element, the overall density was reduced even further. To minimize the effect of the shift register on the overall size of the element, the depth was reduced to 128 entries. This discovery further reinforced the need to change from a die size of $7mm \times 7mm$ to $20mm \times 18mm$.

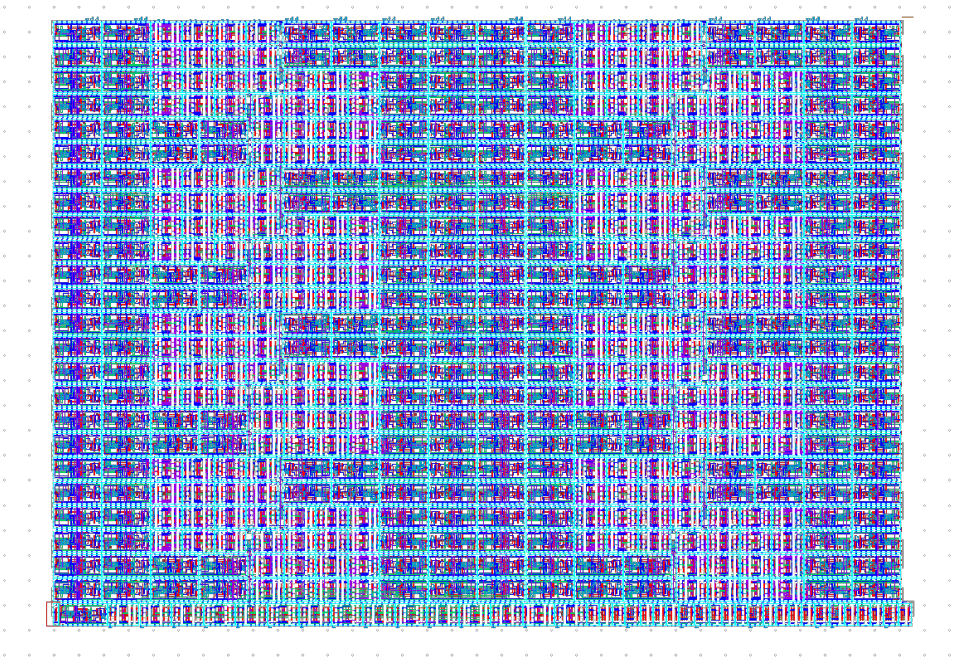


Figure 4.5: Synapse Distance Layout with area of $2628\lambda \times 1886\lambda$

4.4.1 Investigated Solutions

Several solutions to the density problems were investigated. The first was replacing some of the larger collections of flops with SRAM arrays. The pieces that could possibly be replaced are the 256-bit shift register in the synapse distance function, the 41 bits of programming data in the register file, and the FIFOs in the programming interface. When considering replacing flops with SRAM cells, one must keep in mind that additional logic will be required to interface with the SRAM array, so it must be large enough for the reductions in memory area to offset the increase in interface logic area. The primary challenge that was encountered regarding SRAM cells was their current availability to the university. None of the other research groups in the department have utilized SRAM cells in this process and we did not have a license to use the SRAM IP that has been designed for this process. Because we did not have any SRAM cells available to us and did not have the time to devote to designing the SRAM cells, precharging cells, sense amplifiers, or the memory generation scripts, it was decided to stick with the flop based approach.

Because the accumulator as implemented in the FPGA is quite complex and implementing it using a ROM is not feasible, it was redesigned using the full adder cells from the standard cell library. Additional logic was required to implement the protections that are included in the current accumulator design, but it is still much more efficient than implementation that is based on the RTL design and uses only two-input gates and flops instead of dedicated adder cells. By redesigning the accumulator, the transistor count was able to be reduced from 2,358 to 880.

Chapter 5

VLSI Preparation

To prepare for the implementation of the VLSI design, several preparation steps had to be completed. First, access to the various tools, design resources, and premade components needed to be obtained. Then, after access to those resources was obtained, many components that would form the building blocks of the element and array needed to be created.

5.1 Tools and Resources

To successfully complete the VLSI implementation, many different resources were required. The first of those resources was the design software. For this implementation, several pieces of Cadence software were used. First, Cadence Virtuoso is the custom layout tool from Cadence that is used for all of the custom design work such as the creation of all of the components that go into the element and the element itself. Next, Cadence Encounter is the ASIC implementation tool from Cadence that is used in a more automated design flow. It is capable of going from a Verilog netlist to a completed design. For this implementation, it was used to go from a netlist describing how an array should be constructed to an array layout that includes the necessary components to create the clock tree and account for the high fanout of the global nets. The university already had licenses to both of these Cadence products, so no work

was required to gain access to them. One other piece of software is used while creating the chip. This software is designed by Mentor Graphics and is called Calibre. Calibre is used for several purposes. It is used to perform Design Rule Checking (DRC), Layout-vs-Schematic Checking (LVS), and Parasitic Extraction (PEX). These tools integrate with the Cadence environment but are separate pieces of software with a eparate license. Challenges were encountered regarding the license for this software, but they were able to be resolved and did not affect the implementation of the design.

Once all of the necessary software was accessible, several design resources were required. First the Process Design Kit (PDK) for the IBM 8RF process was needed. Originally, the files that make up the base PDK were obtained from other research groups in the department that were also using the process. The problem with this was that the PDK has many different pieces and the research groups in the department only downloaded the pieces that they needed for their research work. Because of this, several necessary pieces of the PDK were not initially available. After going through a process that lasted several months, access was gained to the full PDK through the MOSIS website. Gaining direct access to the full PDK was especially important because, while other research groups in the department had used the portions of the PDK that work with Cadence Virtuoso, no other group had used or downloaded the portions that are required to work properly with Cadence Encounter. The next design resource that was needed was the standard cell library. The designs in the standard cell library are the building blocks that everything else is made out of. A typical standard cell library contains gates, flops, buffers, and sometime basic components such as adders. There are two officially supported standard cell libraries for the IBM 8RF process. One is created by IBM and is called the Digital Library and the other is create by ARM and is called the Artisan Library. Both of these libraries require legal contracts that would take too long to get approved given the timeframe of this thesis. Because of this, an unofficial standard cell library for this process was found. This standard cell library, the cg_lib library, was created by the University of Washington and was provided for this thesis by the research lab of Dr. Holleman, a professor in

the EECS department at The University of Tennessee, Knoxville [29]. This library is fully characterized for usage with Cadence Encounter and provides multiple sizes of the cells listed in Table 5.1. Finally, an attempt was made to obtain SRAM cells and a memory generator. This would allow SRAM memory arrays to be created and used in the element and array. These arrays could be used as the register file in the elements, the storage element of the many shift registers in the element, the FIFOs used for off-chip communication, and many other places. SRAM cells are provided in the two officially supported standard cell libraries and the ARM Artisan library also provides a memory generator, but, as stated above, these libraries were not available for this thesis. After checking with the other research groups in the department and several professors, no memory generators were able to be located. Because of this, the design continued without the inclusion of memory cells.

Table 5.1: Available Standard Cells

Cell	Number
1 bit full adder	8
1 bit half adder	4
2,3,4 input AND	12
2-1 AOI	4
2-2 AOI	4
1-2 OR-AND	4
2-2 AOI, 2 inputs INV	4
Buffer	13
Various D-Flops	20
Inverter	27
2-1 MUX	4
2-1 MUX Invert	4
2,3,4 input NAND	12
2 input OR, 1 input INV	4
3 input OR, 2 inputs INV	4
4 input OR, 3 inputs INV	4
4 input OR, 2 inputs INV	4
2,3 input NOR	8
2 input AND, 1 input INV	4
3 input AND, 2 inputs INV	4
1-2 OAI	4
2-2 OAI	4
1-2 AND-OR	4
2-2 AND-OR	4
2,3 input OR	4
2 input XNOR	4
2 input XOR	4

5.2 Component Creation

After all of the necessary resources were made available, many components needed to be created to support the development of the element. These components were made by combining the standard cells. A 24-bit comparator was made using 2 input XNOR cells to check if the address being driven into the element is the address assigned to that element. A 32-bit wide bitwise AND was created for usage with the monitoring functionality. Components implementing the increment disable and decrement disable logic functions for the LTP/LTD functionality were created. A D-latch was created, as well as an 8-bit wide D-latch and an 8-bit wide D-latch with an enable for the eighth bit. Next, many variations on the provided D-flops were created, including versions with a clock enable, versions with an active-high reset, and versions with both of these features. Additionally, these D-flops were created based on several different flops including flops with both synchronous and asynchronous reset. A logic multiplexer implementing the logic function used for Global Input Select in the LTP/LTD functionality was created. Several different multiplexer widths were created, including a 4-1 multiplexer, an 8-1 multiplexer, a 16-1 multiplexer, an 8-bit wide 2-1 multiplexer, and a 32-bit wide 2-1 multiplexer. Next, a 16-bit addressable shift register was implemented using flops. Using this 16-bit shift register, a 256-bit shift register and a 128-bit shift register were implemented. After this, two counters were created, a 16-bit up counter and an eight bit up/down counter. Finally, a custom shift register used for controlling the output of an element was created. Additionally, throughout the process of creating these components, several standard gates that are wider than what is provided in the standard cell library were created. As these components were created, they were all tested extensively to ensure that they functioned as expected and met the timing requirements of the design. This testing is detailed in Chapter 7.

Chapter 6

VLSI Element Implementation

6.1 Design Flow Stages

The design of the DANNA element followed a very specific process to ensure that the final design behaved as expected, met all of the constraints that were placed on it, and would interact appropriately with the other components such as the surrounding elements in the array and the programming interface signals. Additionally, to reduce the amount of complexity present in the design and accelerate the design process, the element was divided into several distinct pieces. These pieces were all designed separately and then combined to create the full element. Each of the pieces followed the same basic design process, which was as follows. First, the VHDL code from the FPGA implementation was elaborated into a gate-level schematic using the Xilinx Vivado Design Suite. Vivado is the design tool that was used to create the FPGA implementation of DANNA and primarily targets FPGAs, but it does have the ability to translate a design into standard logic gates. The next step was to optimize the VHDL implementation of the component. Because the VHDL code was targeted for FPGAs, the implementation of certain features was often not ideal for implementation in a VLSI design as it translated well to lookup tables but not logic gates. After optimizing the design, any included IP blocks or primitives such as counters and

shift registers had to be created in VLSI. Next, a schematic of the component was created. This schematic represents a transistor-level electrical description of the VLSI design. After this schematic was created, it was simulated using the Cadence Spectre Simulator. At this stage, timing issues were often discovered that required modifications and additions to be made. The stimulus for this stage was designed to replicate both the speed and pattern of signals that this component will encounter during operation. To be certain that the component was behaving as expected, the same stimulus was run through a VHDL simulation in Vivado and the schematic in Spectre. Once all testing at this stage was complete, the layout of the element was created in Virtuoso. For the individual pieces, an attempt was made to keep the shape rectangular to help with the eventual assembling into an element. For the element, an attempt was made to keep the shape square so that the array would also be square. Throughout the layout process, both DRC and LVS were run periodically to ensure that no design rules were being broken and that the layout was implementing the schematic correctly without any shorts. Finally, after the layout was completed, PEX was run to created an extracted netlist for the layout that includes all of the parasitics that were introduced during the layout process. This extracted netlist was then simulated with Spectre using the same stimulus as both the VHDL simulation and the schematic simulation to ensure that the operation is still as expected.

6.2 Element Components

As was stated in the previous section, the element was divided into several pieces to reduce the complexity of the design and accelerate the design process. Based on an analysis of the functionality of the element and using information from the UCNC paper [9], seven distinct pieces were identified.

6.2.1 Accumulator

This piece contains the accumulator for the element. It is used to implement the threshold and charge logic in a neuron and is used to store the weight in a synapse. It is also used to increment and decrement the weight during potentiation and depression, though the identification of when these conditions are met is done by a different piece.

6.2.2 Input Multiplexer Control

This piece is used to interpret the various multiplexers which are used to sample the inputs to the element. All of the sampling multiplexers are 16-1 multiplexers, and there are nine of them. Eight of the multiplexers receive a selected bit from each of the 16 inputs. The ninth receives the fire signals from all connected elements. The control pins for these multiplexers are driven by the Global Input Select signal. The output of each of these multiplexers is then received by the Input Multiplexer Control piece, where they are interpreted for usage in the element. Taking into account the current value of the Global Input Select and the Input Enable register for the element, this piece decides whether to register an input and pass it along to the rest of the element.

6.2.3 LTP/LTD State Machine

This piece is used to generate the signals that control when the weight in the accumulator is incremented or decremented. In conjunction with an external multiplexer, it samples the element specified by the output select register whenever the element fires and it is acting as a synapse to determine if a potentiation or depression condition has been met. If it has, the appropriate accumulator control signal is output. The refractory period for potentiation and depression is also contained within this piece and is used when determining whether to potentiate or depress.

6.2.4 Monitoring

This piece implements the monitoring functionality which has been expanded in this implementation from the FPGA implementation. It contains a 32-bit shift register and the necessary logic to control and load that shift register. It also contains the counters that are used to populate the fields of the shift register that tell the user how many times the element has fired since the last capture and the number of fires currently stored in the synapse distance function shift register.

6.2.5 Output Control

This piece determines when the element should fire. It monitors both the accumulator and the output of the synapse distance function to determine when the element should fire and drives the appropriate signals out when a fire condition is reached. It also uses internal components to ensure that the element fires for the appropriate amount of time.

6.2.6 Registers

This piece contains the register file for the element. It interprets the data coming from the programming interface and the strobe signal to program the correct registers and then drives the values in those registers out to the correct components in the element. It is used to store 41-bits of parameters that are used by the element and also contains the comparator that is used to determine if the address on the global address lines matches the address of the element.

6.2.7 Synapse Distance

This piece contains a large shift register that is used to implement the programmable distance of the synapse. It uses the programmable distance stored in the register file to address into the shift register to allow the fire out of the shift register at the

appropriate cycle. Originally, this shift register had 256 positions, but it was reduced down to 128 positions for the reasons described in Section 4.4.

6.3 Accumulator Design Flow

The accumulator piece of the element contains all of the necessary logic to implement the full functionality of the accumulator. In the original VHDL implementation, the accumulator is a carry-chain adder. This type of adder was chosen for the FPGA implementation because it translates well to the lookup tables that make up an FPGA. This piece uses the clocks, control signals from the LTP/LTD State Machine, fire signals from the Output Control, inputs from the Input Multiplexer, and values from the Registers to operate properly. It outputs several signals. The special reset signal is used when the element is reset or programmed. The registered sum of the accumulator represents the accumulated charge in a neuron or the current weight in a synapse. It also outputs a signal to specify when the accumulator gets reset to its initial value, which usually signals a fire condition. The RTL Elaboration of this piece was produced using the VHDL code from Appendix A.1 and is shown in Figure 6.1.

Upon studying the RTL Elaboration schematic, it became clear that this design was not optimal for implementation in the VLSI chip. Because of the structure of the code, two accumulators are implemented in the RTL Elaboration, though they do share some components. The VHDL code contains two cases for the operation of

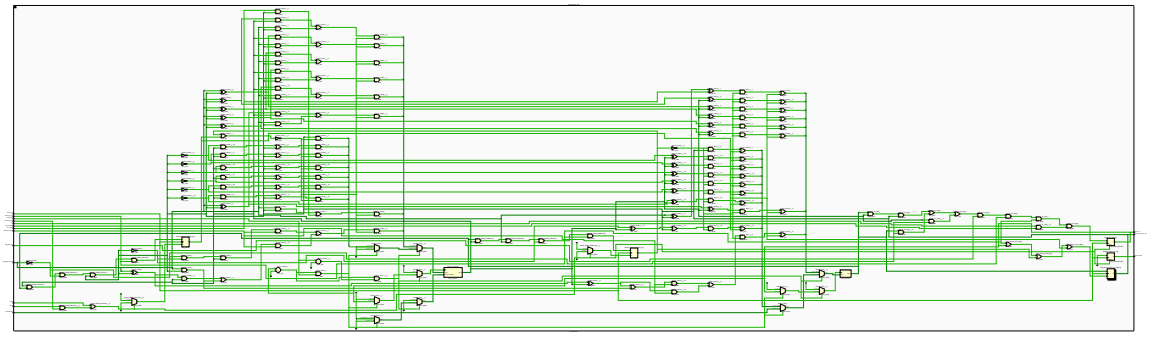


Figure 6.1: Vivado RTL Elaboration for the Accumulator

the accumulator: normal operation and when the stored weight is being decremented. Unfortunately, instead of using a single accumulator and controlling the structure of the inputs, the RTL Elaboration creates two accumulators to serve these two purposes. Additionally, the accumulators are made out of many two input logic gates. While these translate well to the lookup tables of the FPGA, they do not translate well to the VLSI design and could be made more efficient by using dedicated full adders. Because this design is so inefficient, it was decided to do a complete redesign of the accumulator based solely on the functionality of the accumulator and using the same inputs and outputs, but not referencing how it is implemented in the FPGA design. The primary optimization that is present in the VLSI implementation is the usage of a single accumulator. Instead of using two accumulators with fixed inputs, the inputs to a single accumulator are changed. This requires additional multiplexers, but greatly reduces the transistor count overall. Many of the other functions present in the accumulator, such as the control signals, remained unchanged as they were simply basic logic equations. By redesigning the accumulator in this way, the transistor count was reduced from 2,358 transistors to 880 transistors. The optimized schematic for the accumulator from Cadence Virtuoso can be found in Figure 6.2.

After the schematic for the accumulator was completed, it was tested extensively. Details of the tests run and their results can be found in Section 7.1. After testing the accumulator and verifying that it behaves properly on both the behavioral and transistor level and verifying that the timing is appropriate for the expected input signals and clock speeds, work on creating the layout of the accumulator began. While creating the layout for the accumulator, there were several primary design principles. These were to maintain the functionality of the accumulator, to keep the timing in compliance, to create the layout in a square or rectangular shape, and to keep the layout as compact as possible. Additionally, an attempt was made to keep connected components as close together as possible to minimize the distance traveled by the signals, thereby reducing the parasitics for that net and the need to insert additional

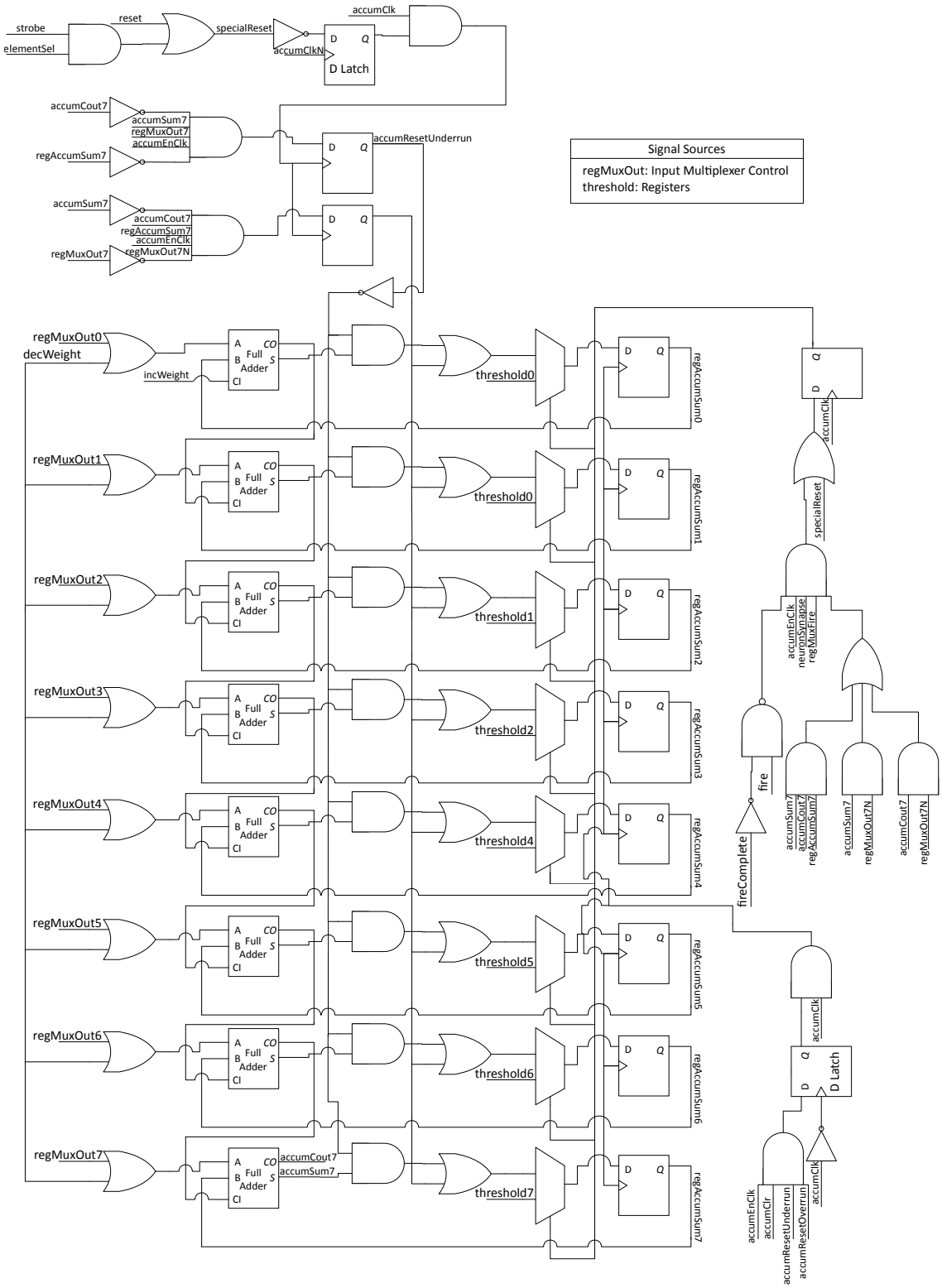


Figure 6.2: Schematic of the Accumulator

buffers. Using the schematic from the previous design stage as the guide, the floorplan was created.

After the placement was defined in the floorplan, the wires to connect the components were added as well as the power rails and all of the pins. Figure 6.3 shows the layout including all levels of the nested components. Note that the scale used in this and all other layout figures is measured in microns.

Finally, once this layout was completed, the same tests that were run on the schematic of the Accumulator were run to ensure that the functionality had not changed and the design still met the timing requirements. These results can also be found in Section 7.1.

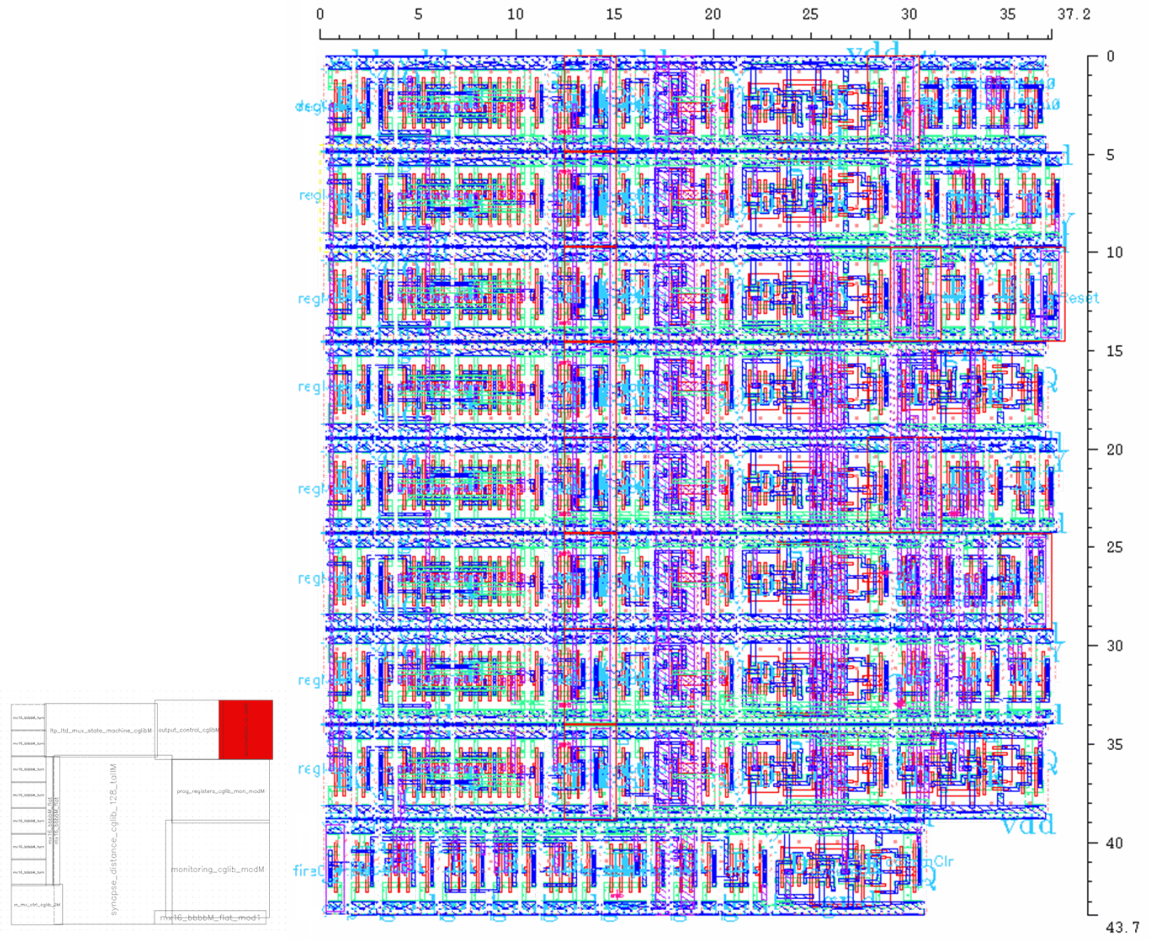


Figure 6.3: Cadence Viruoso Layout and position of the Accumulator

6.4 Input Multiplexer Design Flow

The input multiplexer piece of the element contains all of the necessary logic to register the multiplexer inputs from the surrounding elements when the correct input is being sampled. It determines which inputs should be registered using the 16-bit input enable value provided by the registers piece. It uses the multiplexers located along the left side of the element to sample the surrounding elements. This piece takes the Global Input Select, the Input Enable value, Neuron/Synapse select value from the registers, and the outputs of the sampling multiplexers. It is clocked using the Acquire/Fire Clock since this is the clock used to sample the inputs and this piece is the one doing the sampling. The first step of creating this piece was to reference the RTL Elaboration. The VHDL code that was used to create this elaboration can be found in Appendix A.2, and the elaboration is shown in Figure 6.4.

Because this piece only contains flops and multiplexers, very few optimizations were necessary. Specifically, two optimizations were made to this design. First, the two multiplexers that receive the value from the 16-input multiplexer as their select signal are actually implementing inverters, so they were simply replaced with inverters to reduce transistor count. Additionally, both of the replaced multiplexers receive the same inputs, so they were consolidated into a single inverter. Second, instead of having the clock enable logic present inside each of the flops, the logic to implement this feature was consolidated into a single set of gates that was then driven out to all of the flops. The resulting schematic from Cadence Virtuoso can be found in Figure 6.5. The gates at the top of the schematic are the consolidated clock enable logic.

After the schematic for the input multiplexer control piece was completed, it was tested extensively. During the testing, special attention was paid to the centralized clock enabling logic to ensure that it accurately duplicated the original functionality and met the timing requirements of the design. All details and results of the tests run on this piece can be found in Section 7.2.

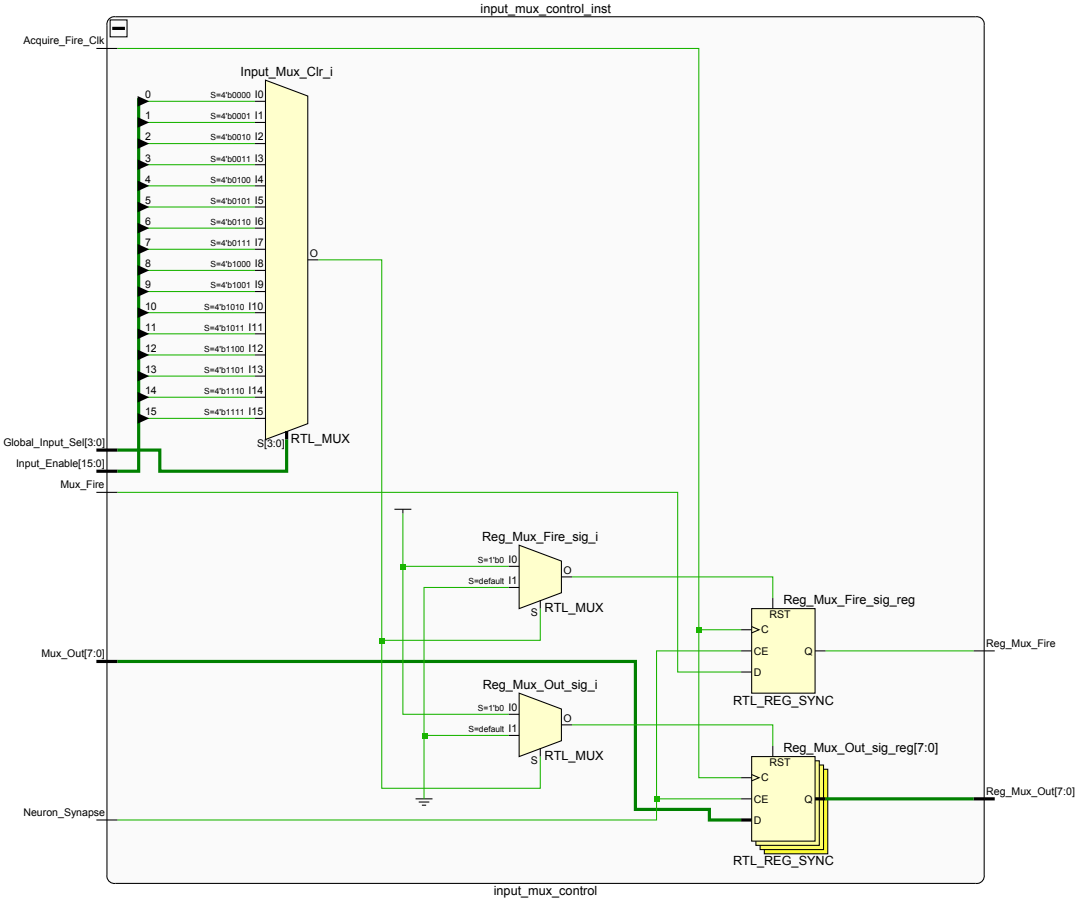


Figure 6.4: Vivado RTL Elaboration for the Input Multiplexer

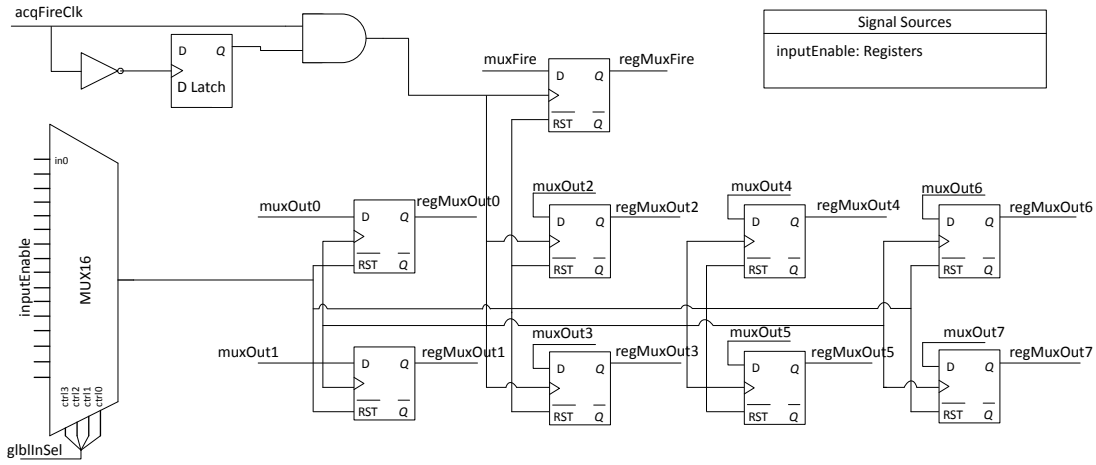


Figure 6.5: Schematic of the Input Multiplexer

After fully testing the input multiplexer control piece and verifying that it both behaves properly and meets the timing requirements of the design, the process of creating the layout of the piece was started. The goals for the layout of the input multiplexer control piece were the same as those of the accumulator: to maintain the functionality of the component, to keep the timing in compliance, to create the layout in a square or rectangular shape, and to keep the layout as compact as possible. Due to the small number of components present in this component compared to the accumulator, most of these goals were simple to implement. Unfortunately, due to the small number of components in this design and one component being much larger than the others, achieving a perfectly square or rectangular shape was not possible. Instead, the design was made to be as close as possible.

After the placement was defined in the floorplan, the wires to connect the components were added as well as the power rails and all of the pins. The completed layout including the layout of all nested components is shown in Figure 6.6.

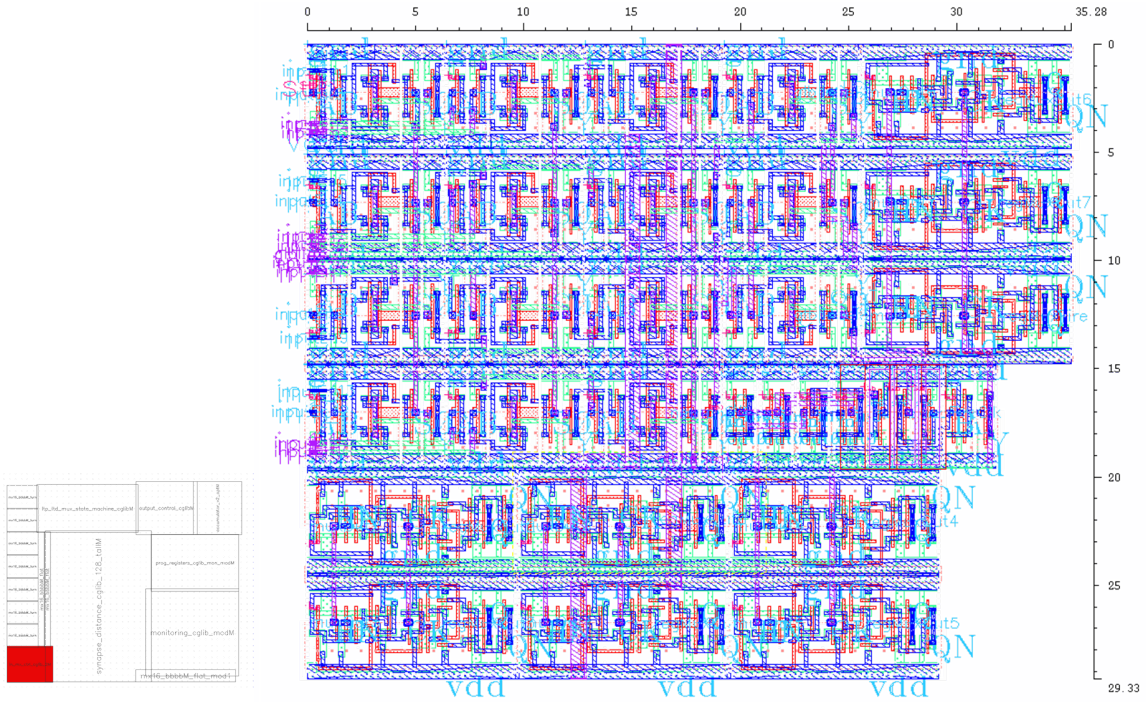


Figure 6.6: Cadence Virtuoso layout and position of the Input Multiplexer

Once the layout of the component was completed, the same tests that were run on the schematic were run on the extracted version of the layout. This ensured that the functionality of the component was not affected by the additional parasitics of the wiring and that the design still met the timing requirements. The results of these tests can be found in Section 7.2.

6.5 LTP/LTD State Machine Design Flow

The LTP/LTD State Machine piece of the element implements several important features of the element related to Long Term Potentiation and Long Term Depression. Primarily, this component determines when a potentiation or depression condition is met and controls the accumulator accordingly. To do this, it uses many signals received from other components and also from surrounding elements. It uses the Fire Out signal from the Output Control component to determine when the element is firing and this is what triggers the check for potentiation or depression. It also uses the values of the Global Input Select, the Output Select, and the Fire signal from the monitored element, selected with the output select signal and sampled using an external multiplexer. Using these signals and several clocks, it outputs signals to tell the accumulator when to increment the stored weight or to decrement the stored weight. Once the element has potentiated or depressed, this component also implements the refractory period. Using the refractory period value received from the registers, it enters a period after potentiating or depressing where the increment and decrement signals cannot be driven high, even if all other conditions are met. Using the VHDL code in Appendix A.3, the RTL Elaboration shown in Figure 6.7 was produced.

Upon studying the RTL elaboration, several needed changes were identified. First, the design contains many multiplexers with input 0 receiving input from a gate, input 1 tied to a constant '1', and the select also being input from a gate. Using this information, the truth table shown in Table 6.1 was developed. Based on this truth

table, the logic equation for this gate was determined to be $Output = Sel + Input$. Because these multiplexers simply implement a two-input OR gate of the select signal and the first input, they were replaced with OR gates to reduce the transistor count.

Next, the two ROMs present in the design needed to be replaced with logic components because this VLSI design does not utilize any ROMs. The first ROM to be replaced with logic was the increment disable ROM. This ROM only outputs a '1' when the eight-bit input is equal to 127. Using this information, equation 6.1 was developed to define the logic and the full design flow for a small component implementing this equation was done, including the schematic, flooplan, layout, and all testing.

$$Output = A[0] * A[1] * A[2] * A[3] * A[4] * A[5] * A[6] * A[7]' \tag{6.1}$$

Next, the decrement disable ROM needed to be replaced with logic. This ROM only outputs a '1' when the eight-bit input is equal to -128 in two's complement. Using this information, equation 6.2 was developed to define the logic. After determining this, the full design flow was used to develop this component as it was with the increment disable component.

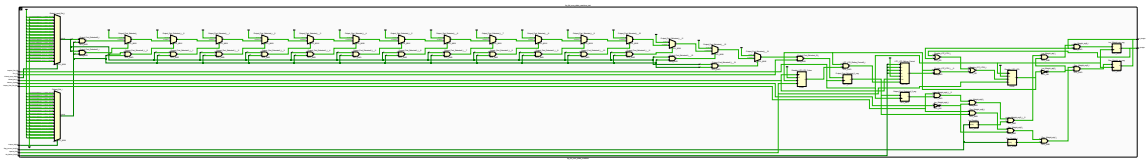


Figure 6.7: Vivado RTL Elaboration for the LTP/LTD State Machine

Table 6.1: Multiplexer Truth Table

Sel	Input 0	Output
0	0	0
0	1	1
1	0	1
1	1	1

$$Output = A[0]' * A[1]' * A[2]' * A[3]' * A[4]' * A[5]' * A[6]' * A[7] \quad (6.2)$$

After all of these changes were made, the optimization of the elaboration was complete and the Virtoso schematic was created. This schematic is shown in Figure 6.8.

After implementing this schematic, it was fully tested and the descriptions of the tests and results can be found in Section 7.3. Next, the floorplan of the component was created with the same goals as the other components. Even though several of the nested components in this piece were much larger than the others, the layout was made to be rectangular.

Next, the components were filled in and all necessary signal and power wiring was added. All of the pins for the component were added as well. The layout showing the full layout of all nested components is shown in Figure 6.9

Finally, all of the tests that were run on the schematic for the component were run on the extracted version of the layout. This ensured that the functionality and timing of the component were still in compliance with the additional parasitics. The results of these tests can be found in Section 7.3.

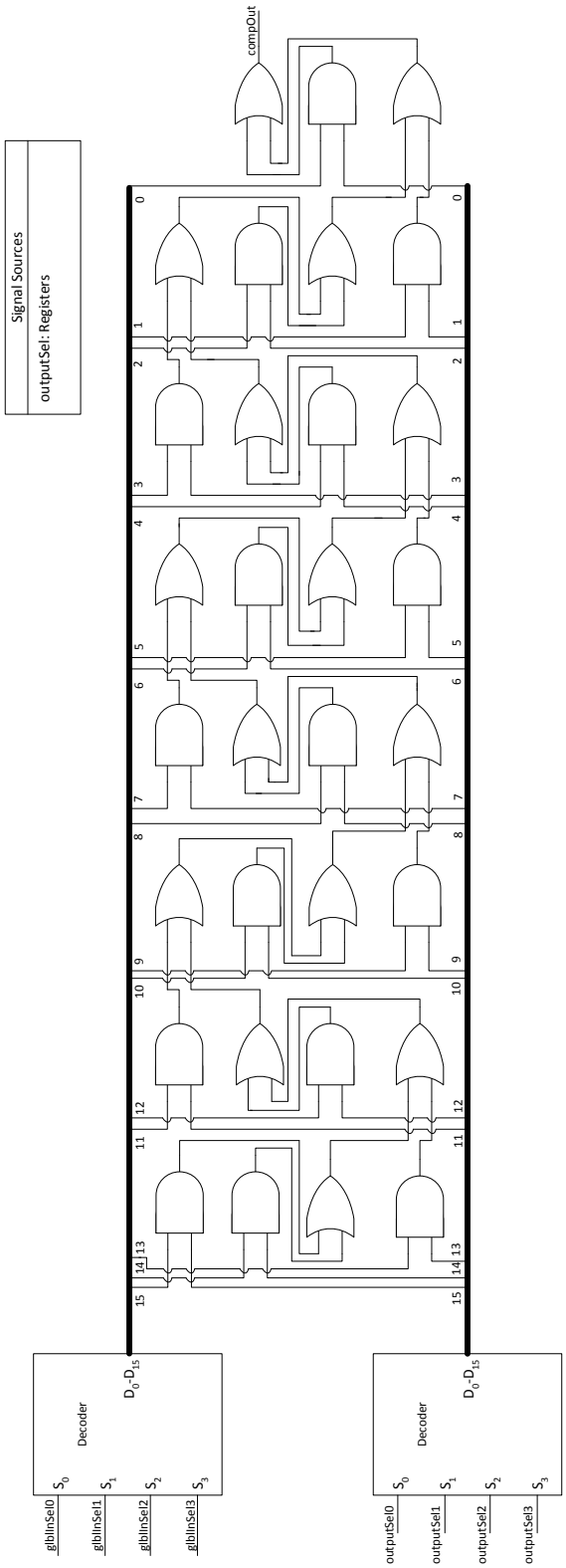


Figure 6.8: Schematic for the LTP/LTD State Machine

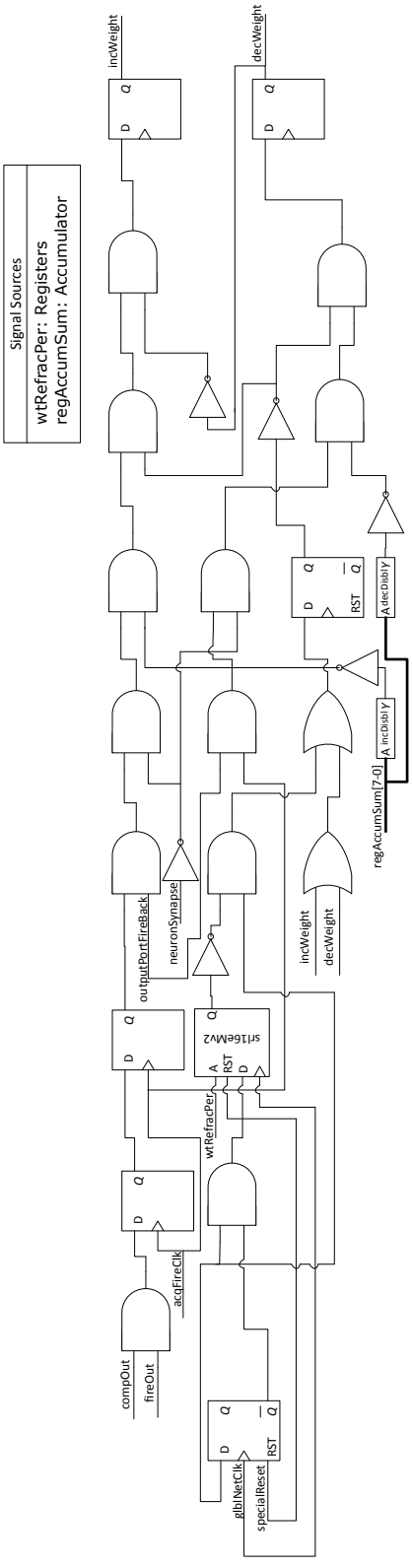


Figure 6.8 (Continued)

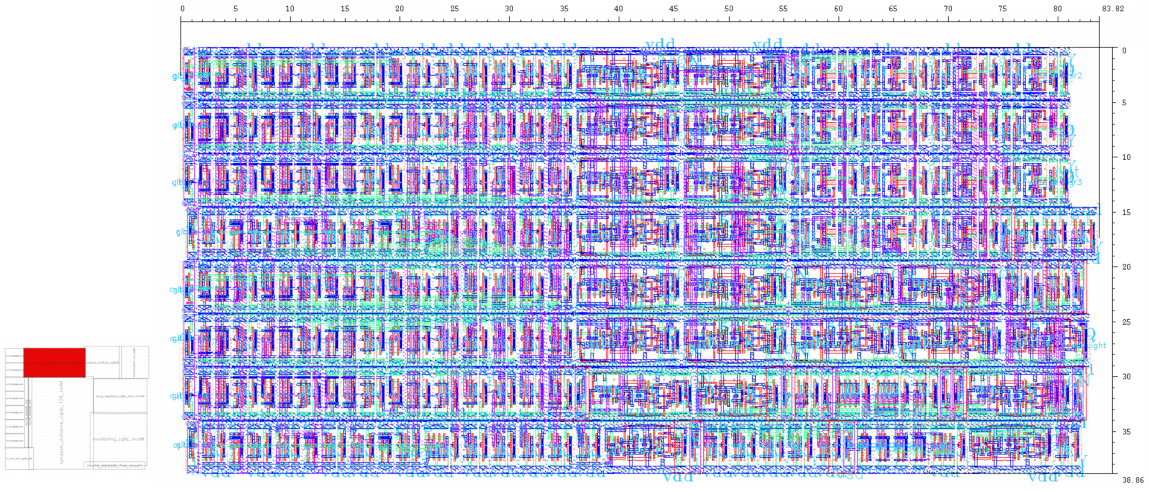


Figure 6.9: Cadence Virtuoso Layout and position for the LTP/LTD State Machine

6.6 Monitoring Design Flow

The monitoring piece of the element implements an important feature of the array that is a relatively new feature. At a high level, it samples or calculates several values from the element and shifts them out of the element serially. When the capture control signal is high and strobe is pulsed, three values are loaded into the shift register. The current value in the accumulator is sampled and represents either the accumulated charge when the element is acting as a neuron or the stored weight, including the effects of potentiation and depression, when the element is acting as a synapse. Additionally, the number of fires since the previous capture command and the number of fires stored in the synapse distance shift register are loaded from internal counters. The counter of the number of fires since the previous capture command is a 16-bit wide up counter that uses the fire signal as its enable and the capture signal as its clear. The counter of the number of fires stored in the synapse distance shift register is an 8-bit up/down counter, with the increment signal tied to the input to the shift register and the decrement signal tied to the output of the shift register. A more detailed description of the operation of the monitoring component and the design decisions that define it can be found in Section 4.3. Using the VHDL code in Appendix A.4, the RTL Elaboration shown in Figure 6.10 was produced.

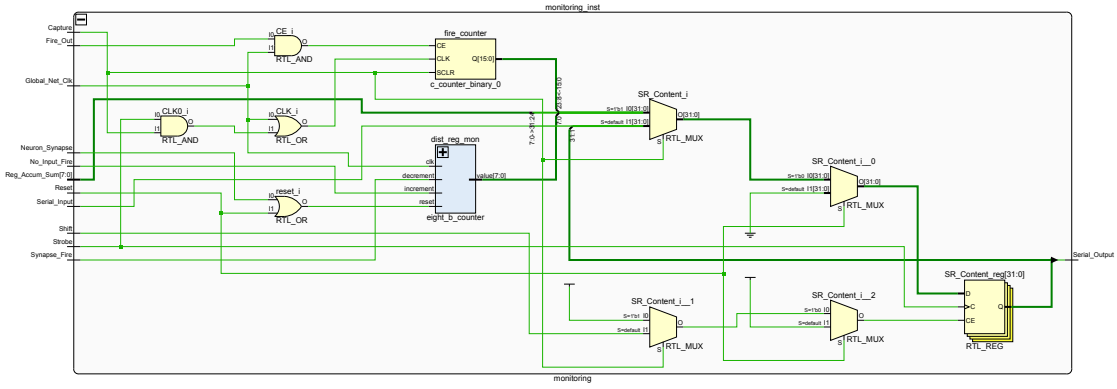


Figure 6.10: Vivado RTL Elaboration for the Monitoring

Upon studying the elaboration, the need for several changes and optimizations became apparent as well as the need to create several supporting components to replace IP blocks in the VHDL. First, the IP blocks used for the counters needed to be replaced with custom components. Because these components are somewhat complex, a short description of the design process for each is included here.

The first supporting component to be created was the eight-bit up/down counter. This counter takes in clock, reset, increment, and decrement signals. When a rising clock edge is received while the increment signal is high, one is added to the output value of the counter. When a rising clock edge is received while the decrement signal is high, one is subtracted from the output value of the counter. If neither signal is high when the rising edge is encountered, nothing happens. If both increment and decrement are high at the same time, the value of the counter will remain unchanged. This counter is implemented using full adders with the inputs to the adders modified by the inputs to the counter. Using this description of the component, the schematic shown in Figure 6.11 was developed.

After the schematic was developed, the counter was tested to ensure functionality and timing. Because of the small number of inputs to the component, all possible combinations of inputs were exercised with a 5 MHz clock, which is the clock speed

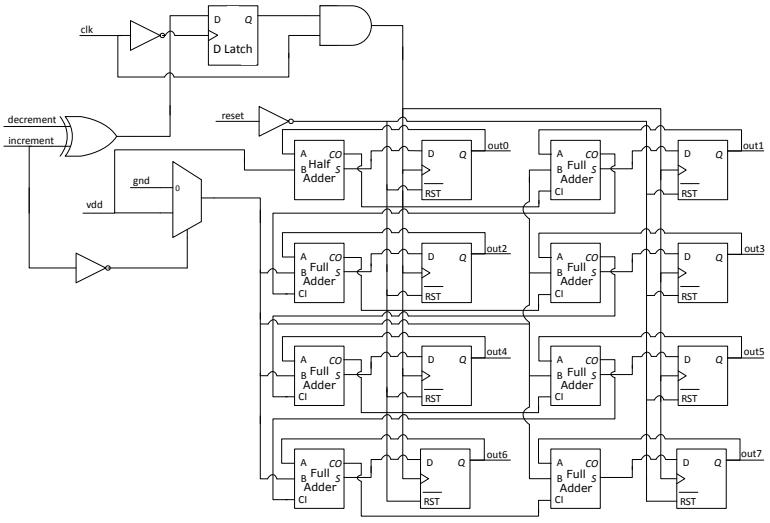


Figure 6.11: Schematic for the 8-bit up/down counter

that is used in the final design. The component passed all of these tests. Next, a layout was created from the schematic while attempting to make the component rectangular and as small as possible. The resulting layout is shown in Figure 6.12 and is quite compact and rectangular.

Finally, the extracted version of this layout was run through the same tests as the schematic and no issues were discovered. Using a similar design methodology, the 16-bit up counter was developed. First, the inputs and outputs were determined. The counter uses a clock, clock enable, and reset as its inputs. When a rising edge of the clock is received, the output value will be reset to zero if the reset input is high, the output value will be incremented if the clock enable is high, or the output value will remain unchanged if both control inputs are low. Using full adders as the base, a schematic for the counter was developed. The counter is based on an eight-bit ripple carry adder that always adds one to the current stored value. To control when this happens, the clock enable signal of the flops used to store the value is manipulated. The schematic that was developed is shown in Figure 6.13.

Next, the counter was tested to ensure functionality and timing, with all possible combinations of inputs being exercised due to the low number of inputs. After successfully completing the testing, the layout for the component was developed and is shown in Figure 6.14. As the figure shows, the layout for this component is very compact but was not able to be made very rectangular, with the left side of the layout

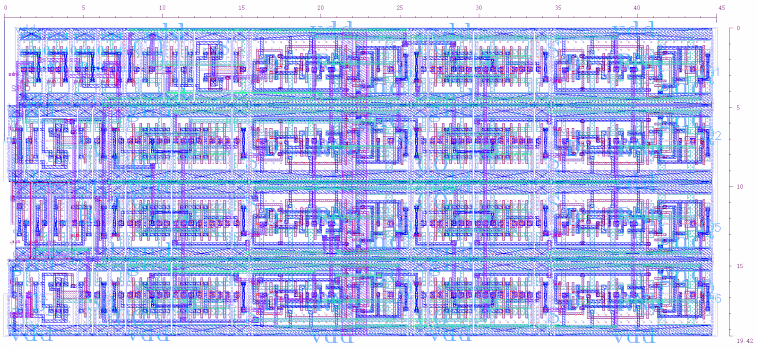


Figure 6.12: Cadence Virtuoso Layout for the 8-bit up/down counter

being particularly jagged. While this was not ideal, it was able to be compensated for higher in the hierarchy.

The extracted version of the layout was then run through the same tests as the schematic and no issues were discovered with it. After completing the design and testing of both of the counters, several other optimizations were necessary. First, the

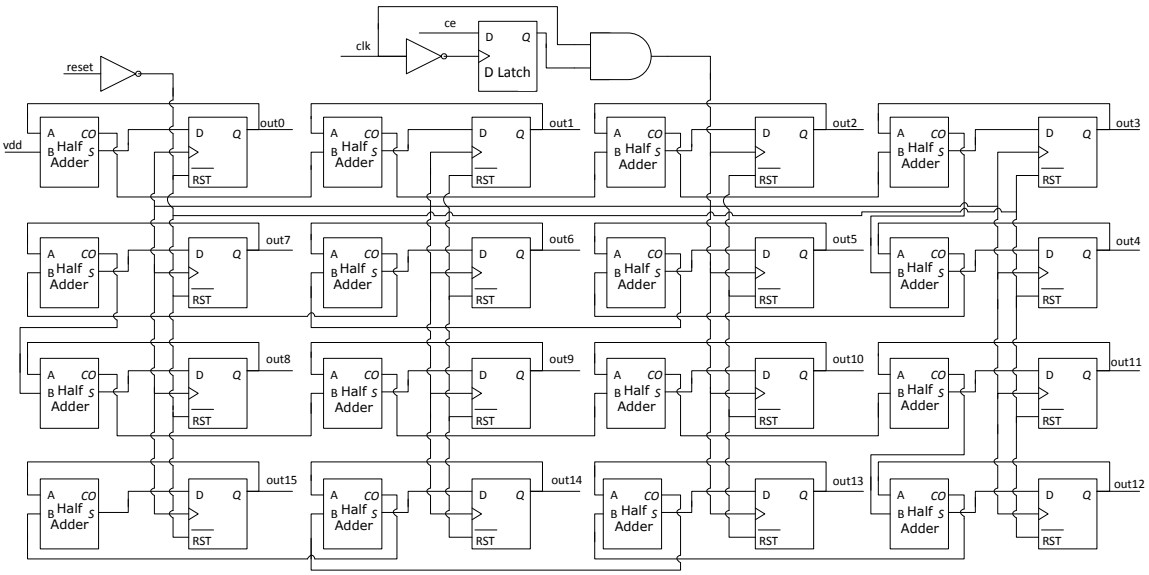


Figure 6.13: Schematic for the 16-bit up counter

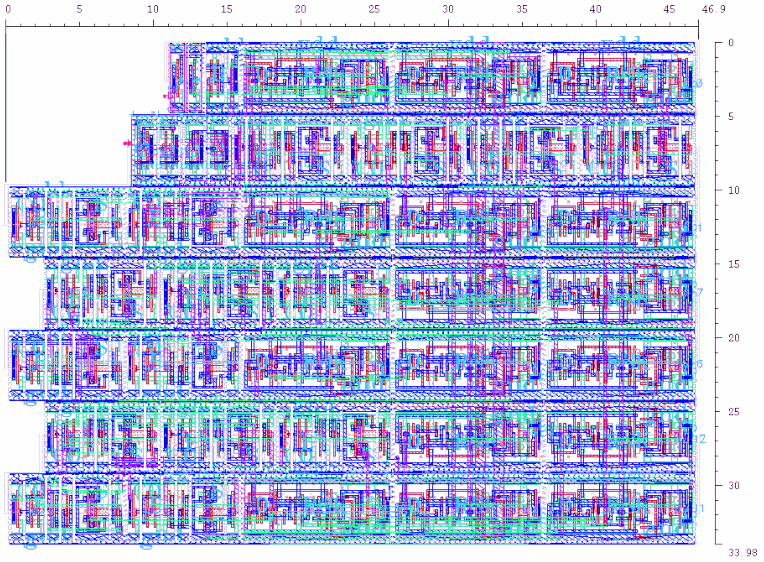


Figure 6.14: Cadence Virtuoso Layout for the 16-bit up counter

multiplexer labeled `SR_Content_i__0` is simply choosing between a logic '0' and the output of `SR_Content_i`. Because of this, it is more efficient to replace it with a two-input AND gate. To maintain the same functionality, the input from `SR_Content_i` is ANDed with the inverse of the select signal. Similarly, both `SR_Content_i__1` and `SR_Content_i__2` are selecting between a logic '1' and a signal so they were replaced with two-input OR gates. Finally, to conserve space, the clock enable logic within each of the 32 flops was consolidated to a single copy. After including these optimizations in the design, a schematic view was produced and the design was tested as described in Section 7.4. During the testing, it was discovered that consolidating the clock enable logic for all 32 flops into a single component caused the design to fail timing and miss inputs due to the large load on the single signal. Because of this, an additional copy of the clock enable logic was added with each copy driving half of the flops. After retesting the design, this was found to fix the problem. In anticipation of the additional delay introduced by the parasitics of the layout, additional distribution of the clock enable logic was performed to bring the total number of copies of the logic to four, with each driving eight flops. The schematic with these changes is shown in Figure 6.15

Next, the floorplan of the component was created with the same goals as the other components of maintaining the functionality, keeping the timing in the allowable range, creating a square or rectangular layout, and creating as compact of layout as possible. As was noted in the discussion on the design of the 16-bit counter, the irregular shape of the counter was overcome by filling the gaps with smaller components to create a layout that is nearly rectangular.

Finally, the components were all connected together with all of the necessary wires and pins added. The view of the layout showing all nested components is in Figure 6.16

After running the extracted version of the layout through all of the tests, it was found to function correctly. Special attention was paid to ensuring that the clock

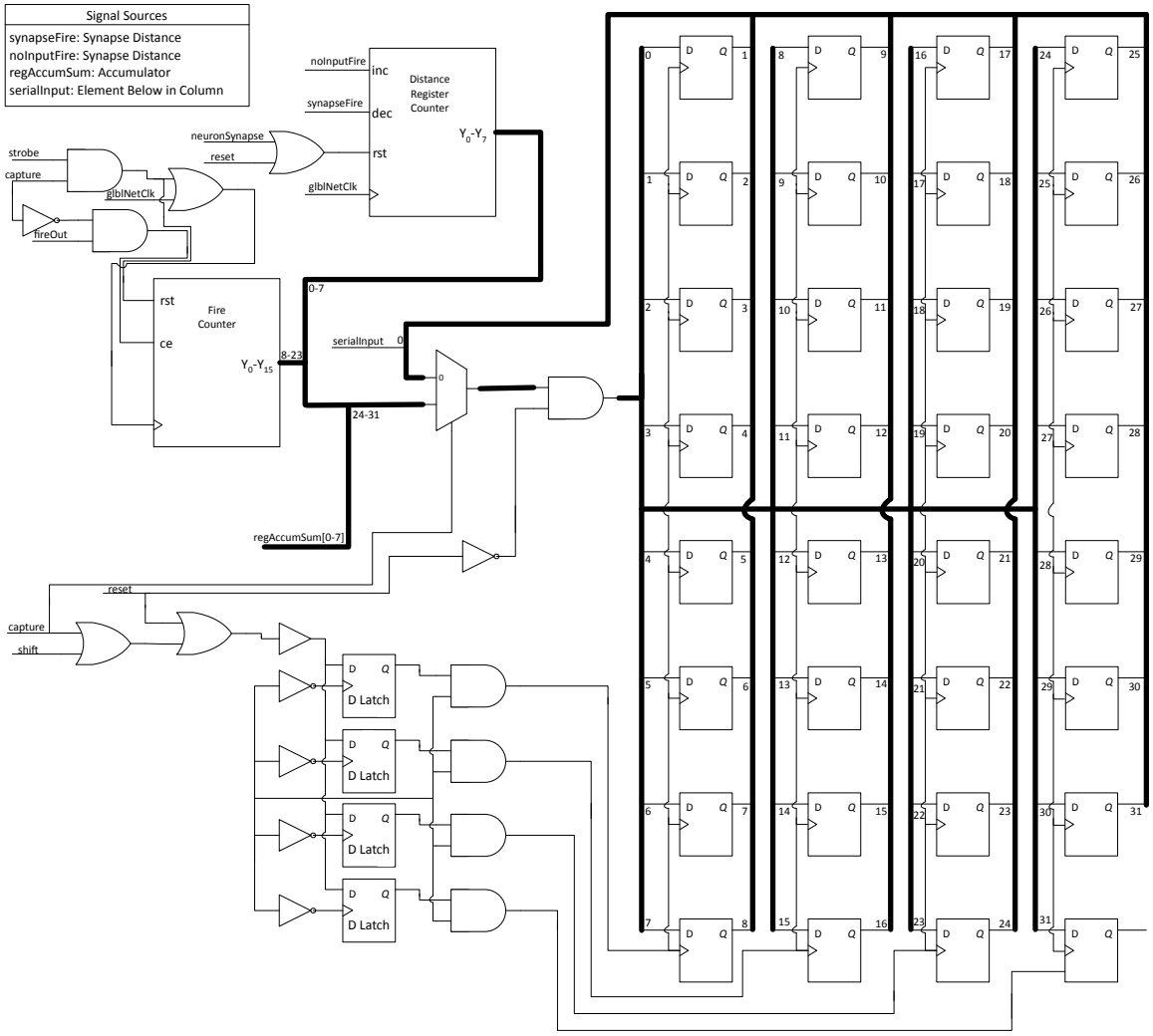


Figure 6.15: Schematic for the Monitoring

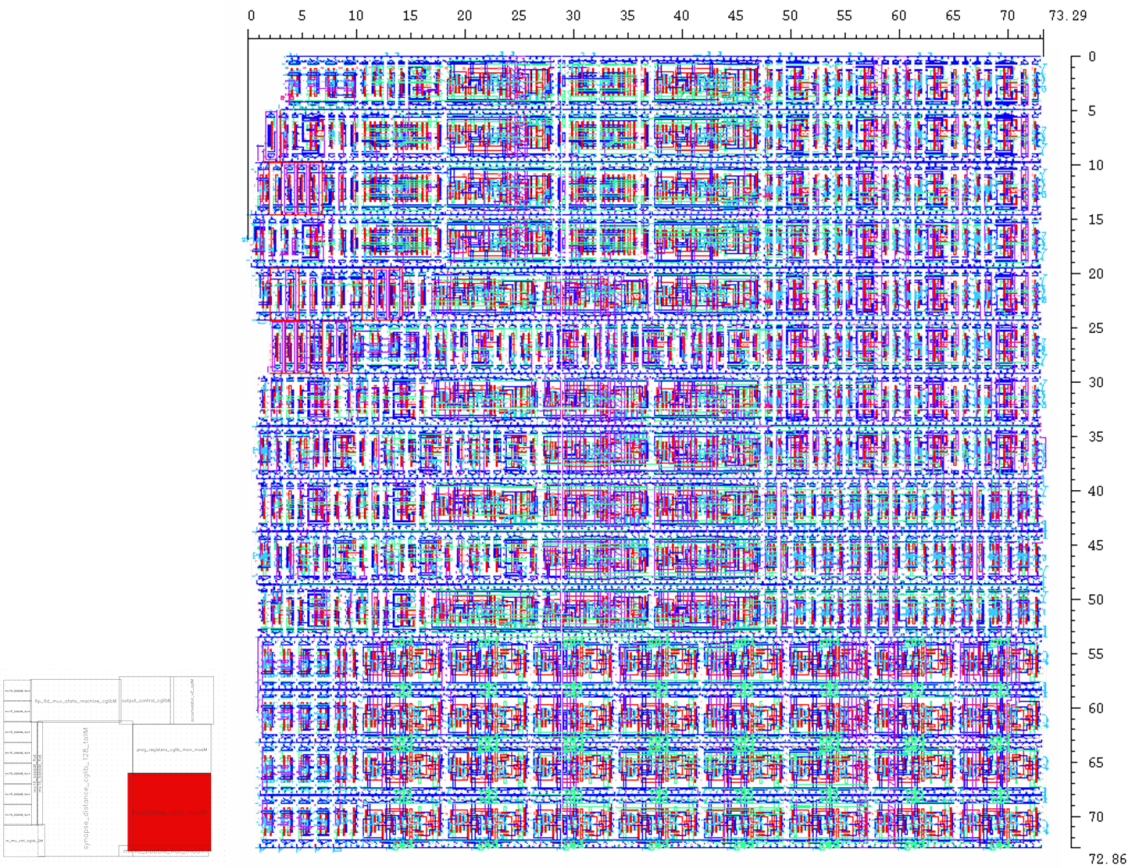


Figure 6.16: Cadence Virtuoso Layout and position for the Monitoring

enable logic still functioned correctly because it required modifications to pass some of the tests on the schematic view.

6.7 Output Control Design Flow

The output control piece of the element implements the features related to deciding when the element should fire and controls how long the element fires. When the element is acting as a neuron, the output control component samples the Accumulator Clear signal that is pulsed when the value in the accumulator gets reset back to the initial value. When the pulse is received, it triggers the component to start firing. It then uses a shift register to keep the element firing for 16 Acquire/Fire Clock cycles. When the element is acting as a synapse, the output control component samples the synapse fire signal from the synapse distance component to determine when it should be firing. Instead of determining how long the element should fire, it simply passes the fire signal through. The component also determines what should be sent out as the 8-bit fire data. When the element is a neuron, the threshold is passed through and when it is acting as a synapse, the weight is passed through. Using the VHDL code in Appendix A.5, the RTL Elaboration shown in Figure 6.17 was produced.

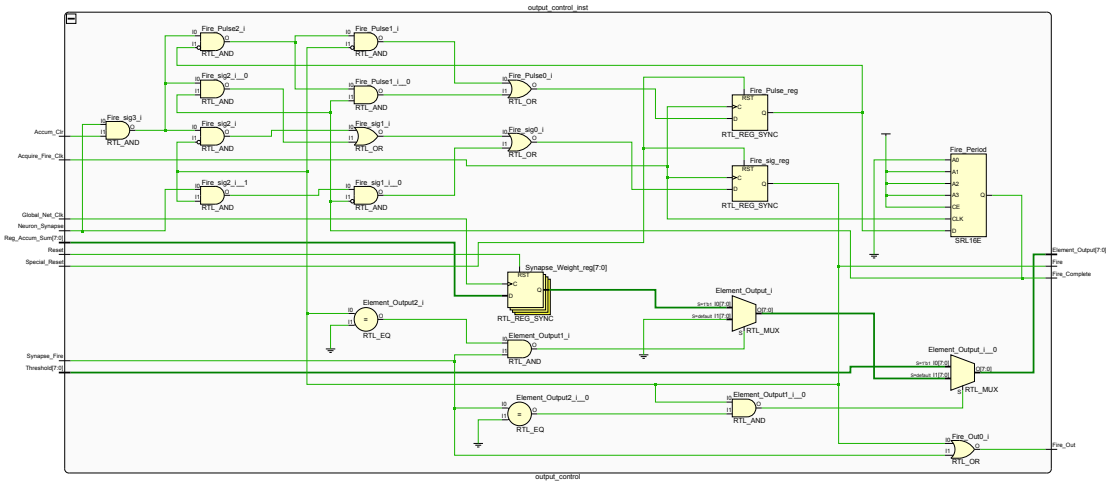


Figure 6.17: Vivado RTL Elaboration for the Output Control

After studying the RTL elaboration, only one change was found to be necessary. The RTL elaboration contains an addressable 16-bit shift register, but the address pins are tied to constants. Because of this, the addressability is not necessary and the full length of the shift register is not necessary. Because the address pins are hardcoded to 15, a non-addressable shift register of length 15 was created and tested for the design. After implementing this change, the schematic shown in Figure 6.18 was created.

Next, this schematic was fully tested to ensure that it functions properly and meets the timing requirements of the design. Details of the tests run and their results can

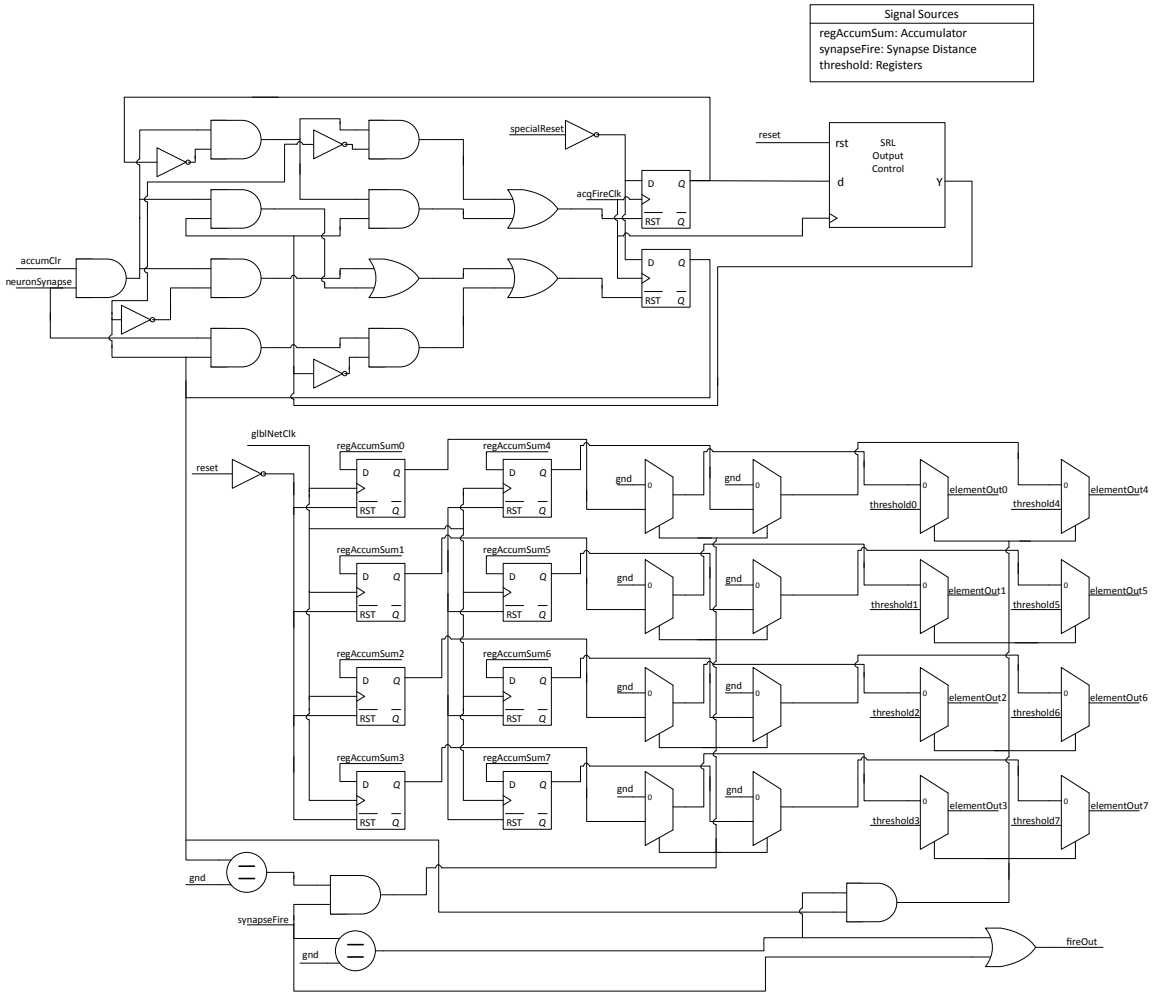


Figure 6.18: Schematic for the Output Control

be found in Section 7.5. After completing the testing of the schematic, the floorplan was created using the same design principles as the other components. The floorplan is very compact and the shape is nearly square, though the corners on the left are both not square.

Using the floorplan as the guide, the layout for the component was created by adding all necessary wires and pins. Figure 6.19 show the layout of all nested components in the design.

Finally, once the layout was completed, the extracted version of it was fully tested to ensure that the functionality was still correct and the design still met the timing requirements. After all of these tests were run and completed successfully, as described in Section 7.5, the output control component was complete.

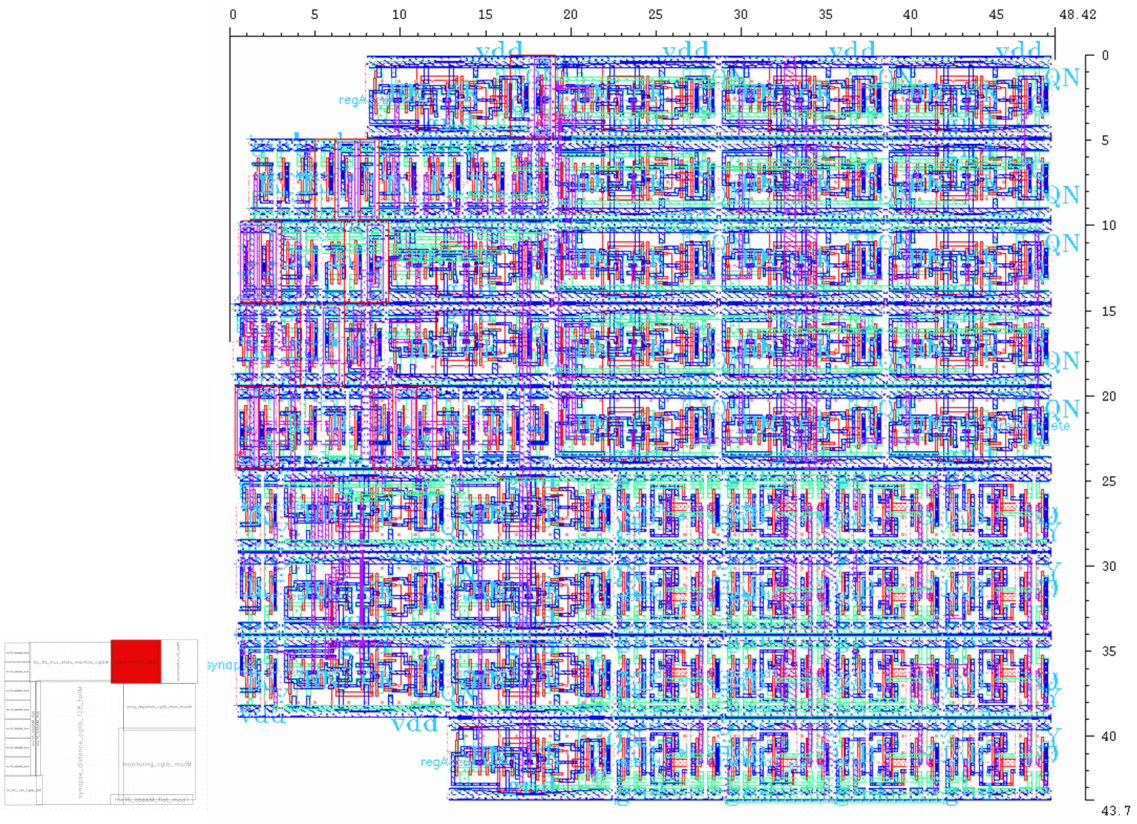


Figure 6.19: Cadence Virtuoso Layout and position for the Output Control

6.8 Registers Design Flow

The registers component of the element receives programming data from the external programming data signals, stores that data, and then distributes that data to the other components that make up the element. Because the sole purpose of this component is to store data, the RTL Elaboration is relatively simple. It is shown in Figure 6.20 and was created using the VHDL code in Appendix A.6.

Upon studying the elaboration, a few necessary optimizations and replacements were identified. First, the ROM that is used to determine if the address being driven on the `Array_Addr` lines matches the hardcoded address of the element needed to be replaced. Because this ROM depends on a value that is hardcoded in each element and is unique to each element in the array, a unique hardcoded logic design would

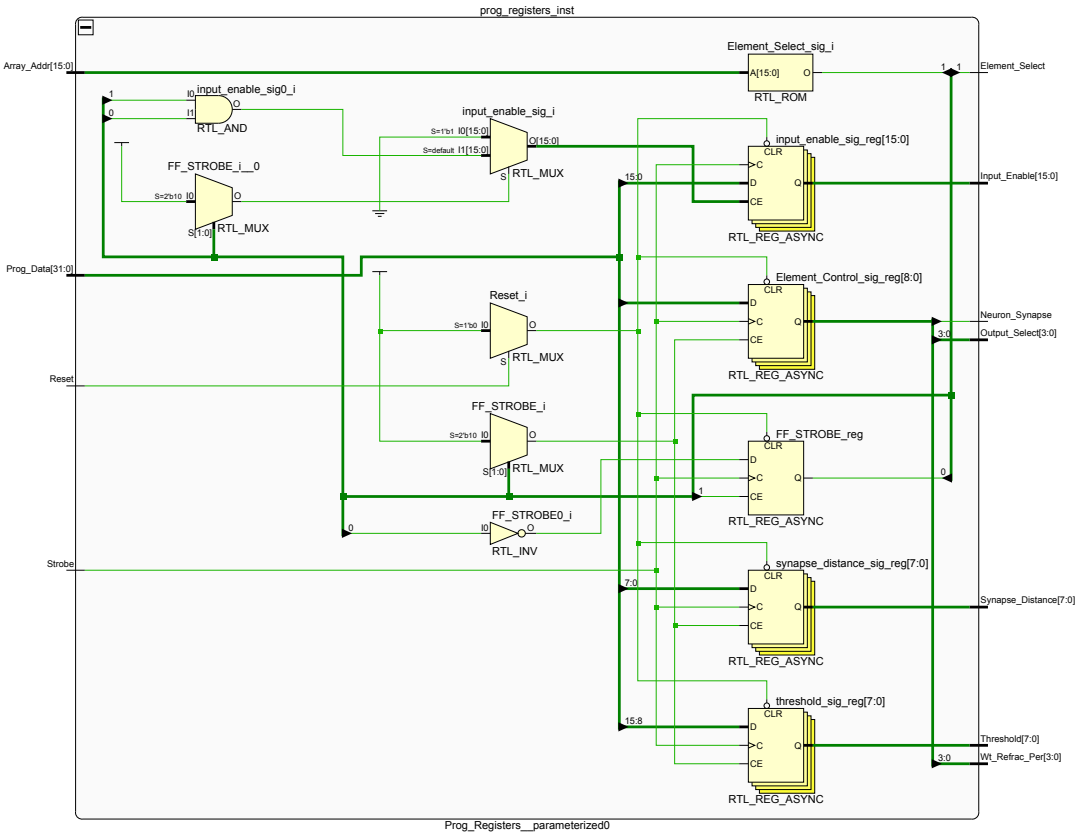


Figure 6.20: Vivado RTL Elaboration for the Registers

be required for each element in the array. To avoid this situation, the ROM was replaced with a 24-bit wide comparator with the unique element driven onto pins that can be tied to either '1' or '0' without having to modify the component. Next, there are four multiplexers in the design with inputs tied to either a logic '1' or '0'. These multiplexers were replaced with AND gates and OR gates to reduce the transistor count. Finally, it was discovered through testing that two additional flops were needed to get the timing correct in the design. The first added flop registers the strobe clock enable signal since it is based on feedback from the `FF_STROBE` register and originally was not stable long enough to be caught by the register. Second, the clock enable for the input enable registers needed to be latched because it is based on a combination of the signal signifying that the element is selected by the array address and the alternating `FF_STROBE` signal and did not stay constant long enough to allow the programming data to be registered in the flops. After making these changes to the design, the schematic shown in Figure 6.18 was created.

Once this schematic was produced, it was tested to ensure proper functionality and timing. Details of these tests can be found in Section 7.6. Next, the floorplan was created to be as compact and rectangular as possible while maintaining the functionality and timing of the design. The floorplan for the registers component was able to be made very rectangular and regular because the majority of the components in the design are flops of a uniform size. In keeping with the goals of maintaining a rectangular shape and compact layout, several of the components were rotated to fit better. Additionally, the smaller components present in the design were used to fill in the gaps at the top of the right side of the design.

Using the floorplan as the guide, the layout for the component was created by adding all necessary wires and pins. Figure 6.22 shows the full layout of all of the nested components.

Finally, once the layout was completed, the extracted version was fully tested to ensure that all of the requirements for the layout were met and that it still functions

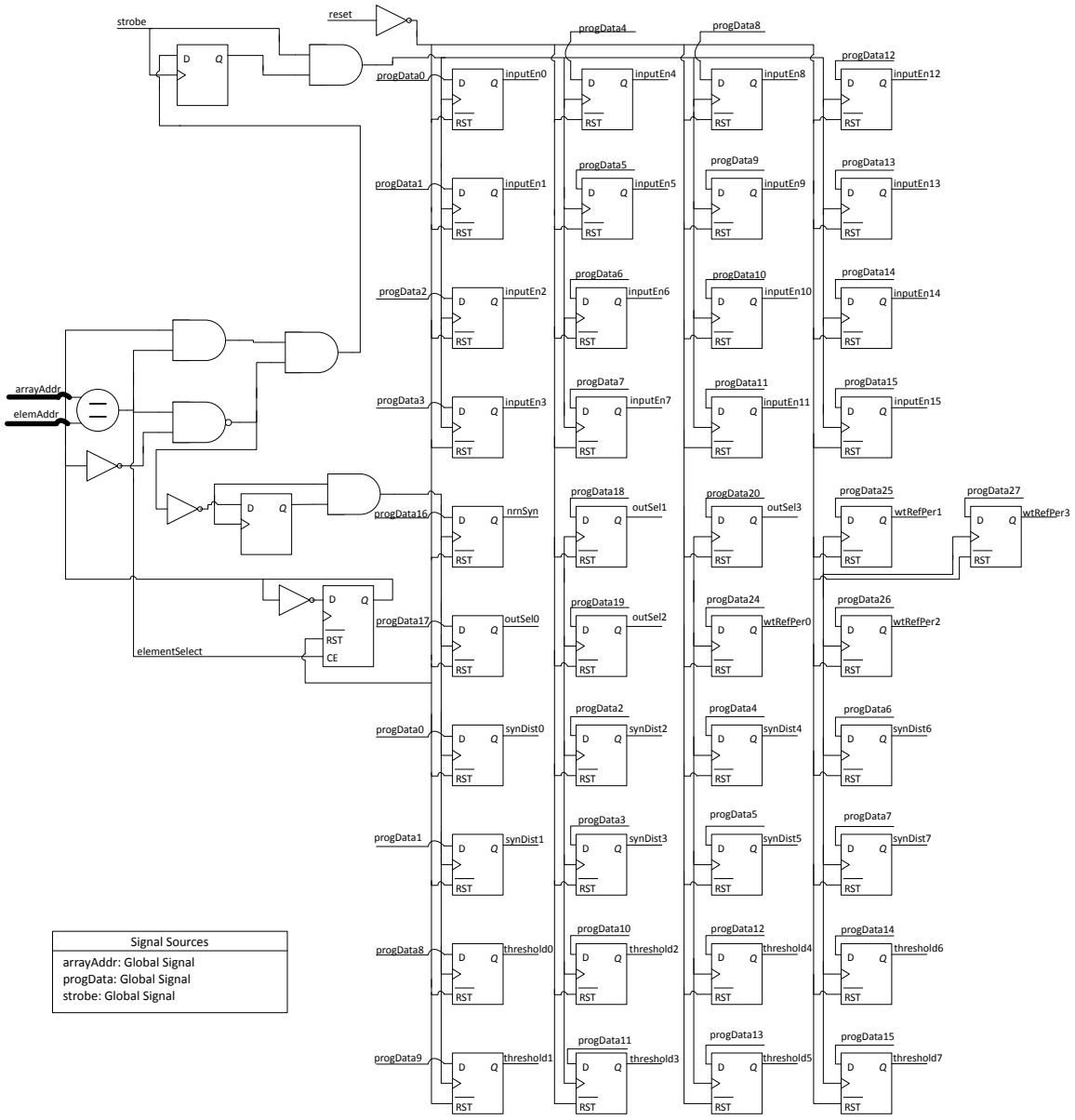


Figure 6.21: Schematic for the Registers

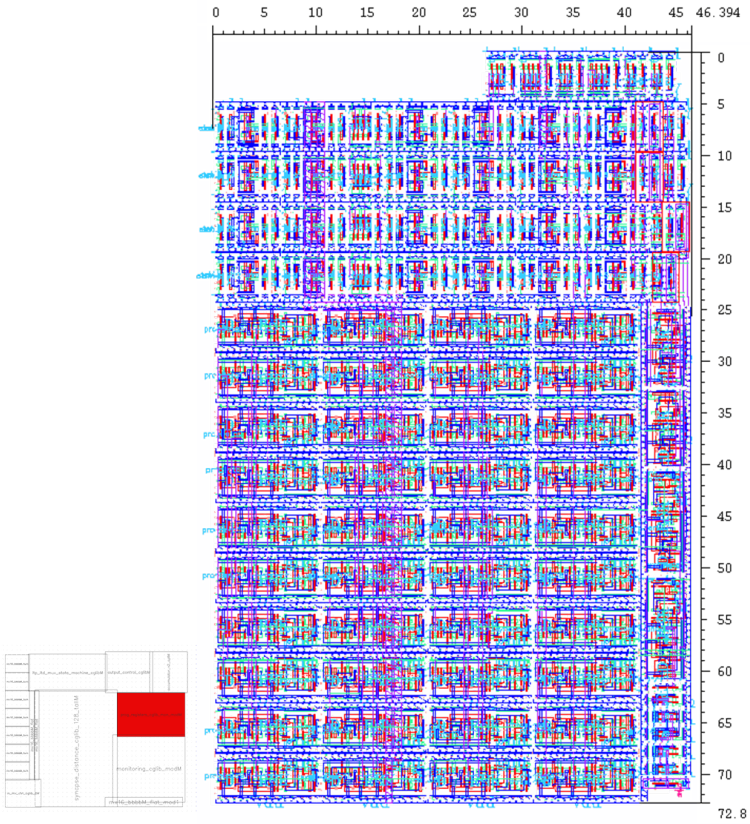


Figure 6.22: Cadence Virtuoso Layout and position for the Registers

properly. As described in Section 7.6, the layout passed all of these tests and the design of the component was complete.

6.9 Synapse Distance Design Flow

The synapse distance function component implements the programmable delay that is utilized when the element is acting as a synapse. It is primarily made up of a large shift register. It takes the current value of enabled inputs and all sixteen of the input fire ports as its inputs. Those inputs are then multiplexed down to the input of the shift register, which is clocked using the Global Network Clock. Additionally, the synapse distance value from the registers is used to address into the shift register. By doing this, an input fire on the enabled input will come out the other end after

a specified number of cycles. Specifically, after an input fire is received, the count of cycles begins at the next rising edge of the clock. Then, after the number of cycles specified by the synapse distance plus one has elapsed, the fire is sent out of the component. A timing diagram illustrating this is shown in Figure 6.23. Using the VHDL code from Appendix A.7, the RTL Elaboration shown in Figure 6.24 was produced.

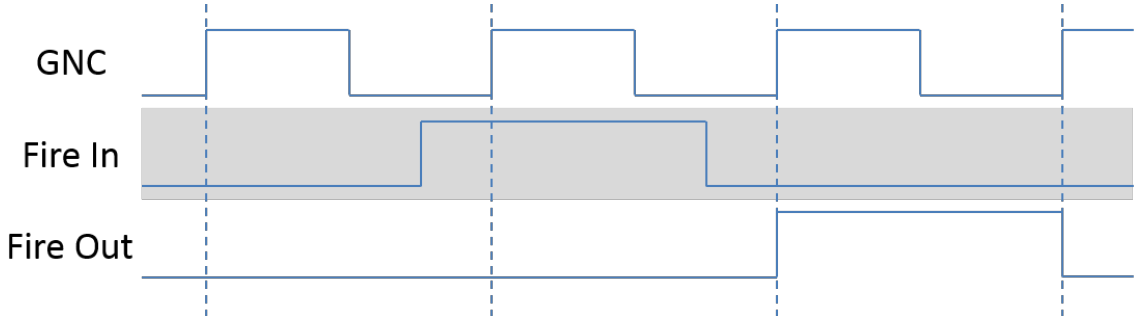


Figure 6.23: Synapse Distance timing diagram with 0 distance

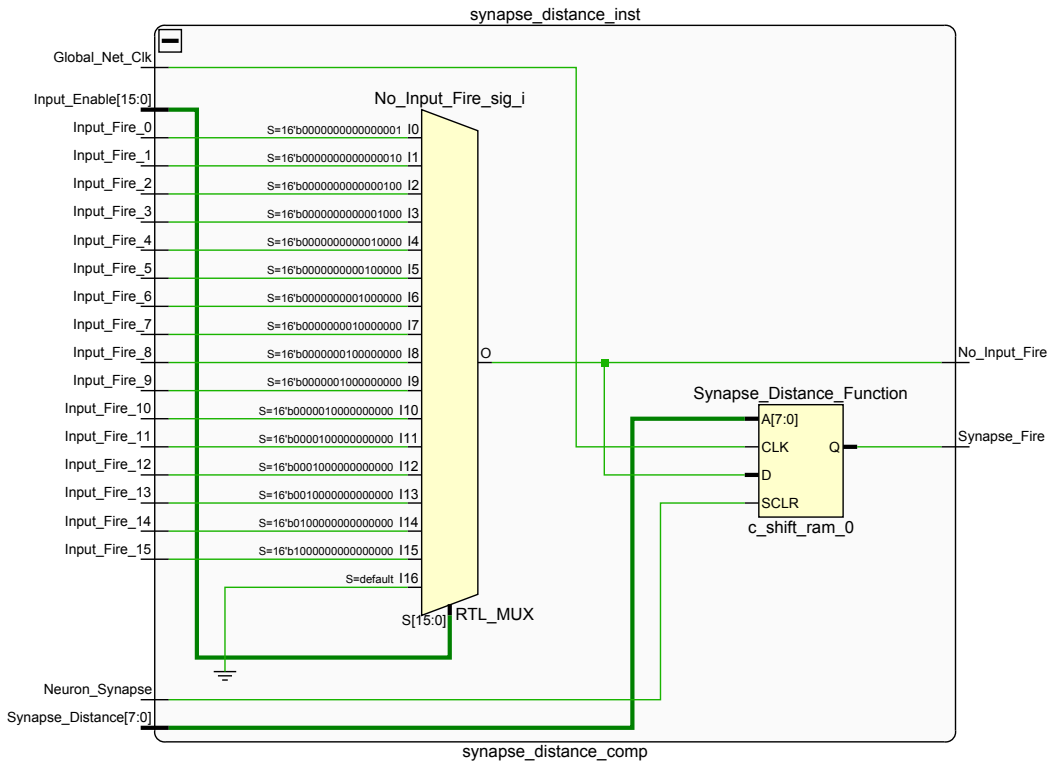


Figure 6.24: Vivado RTL Elaboration for the Synapse Distance

Upon studying the elaboration, one necessary change became evident. The multiplexer used for sampling the input fires to be sent into the shift register uses 16 select bits, driven by the input enable, with all but 16 of the 256 inputs to the multiplexer tied to a logic '0'. Instead of this, a logic function was implemented where a '1' will be passed to the shift register if an input enable bit is high at the same time as that same input fire bit. In addition to the change to the multiplexer, one additional feature was added to the shift register. The FPGA implementation of the shift register is only reset when the element type is set to a neuron and the contents of the shift register are not actually reset, just the output of the component, and the shift register cells are made using a RAM. In the VLSI implementation, the shift register cells are made using flops and they are actually reset when the shift register is reset. Additionally, instead of only being reset when the element is configured as a neuron, the shift register is also reset whenever the internal `specialReset` signal is set. This signal goes high whenever the element is programmed or reset. After these changes were implemented and the new features were added, the schematic shown in Figure 6.25 was created.

As discussed in Section 4.4, the size of the synapse distance function was found to be a challenge once the full element was floorplanned. Additional figures and discussion about the area occupied by this cell can be found in Section 6.10. Because of the large area taken up by the synapse distance function, the decision was made to reduce the size of the shift register from 256-bits to 128-bits. This allowed for a reduction in area of roughly 50 percent. After this change was made and the schematic was modified accordingly, testing was conducted on the component to ensure that it functioned properly at the desired clock speeds. These tests are detailed in Section 7.7. After the testing was completed, the floorplan for the component was created using the same goals as the other components. The floorplan is very compact and regular due to the large shift register, but it was not able to be made completely rectangular. Because the cutout in the bottom left corner was rectangular, this irregular shape was able to be overcome during the layout of the full element.

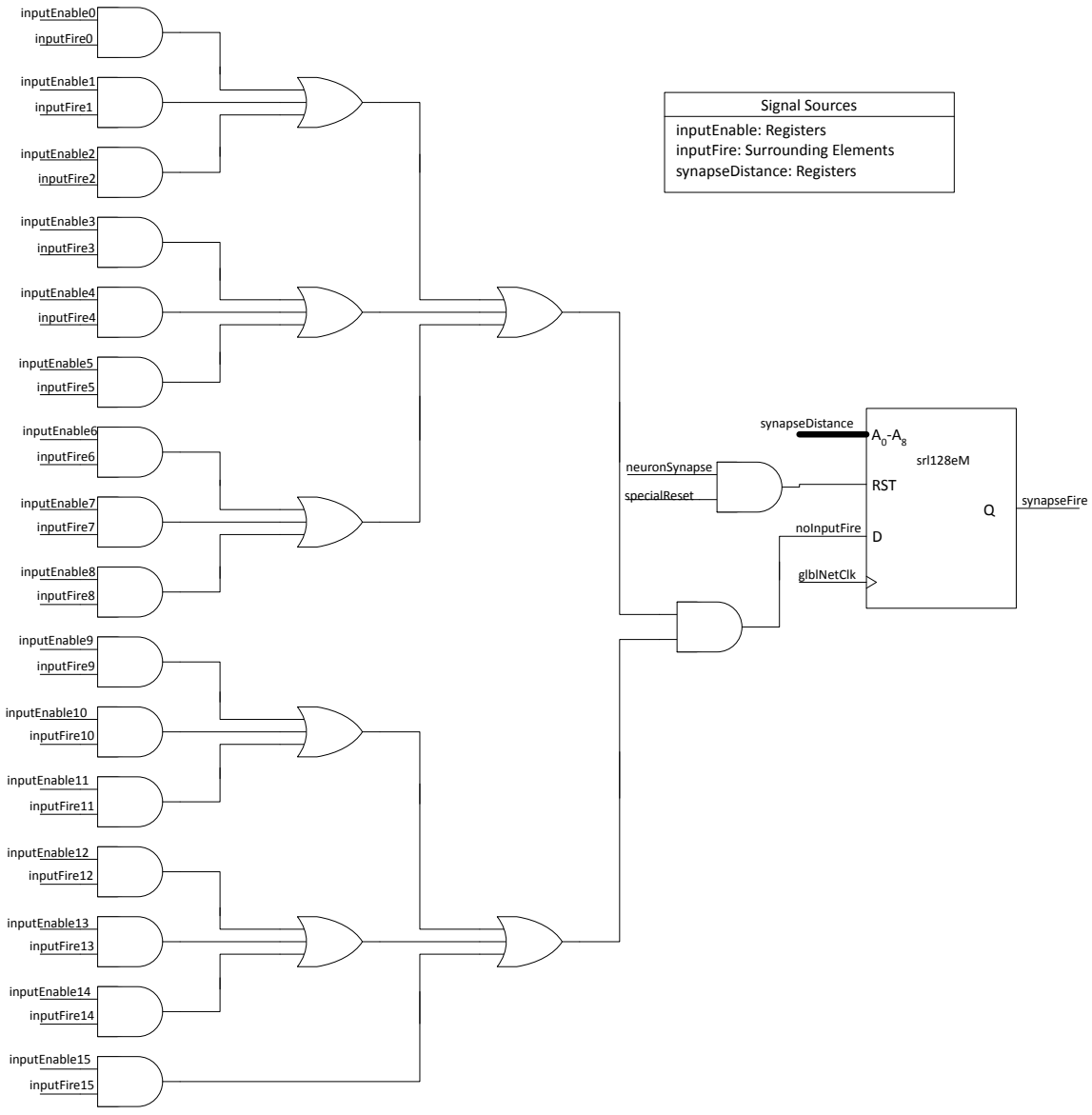


Figure 6.25: Schematic for the Synapse Distance

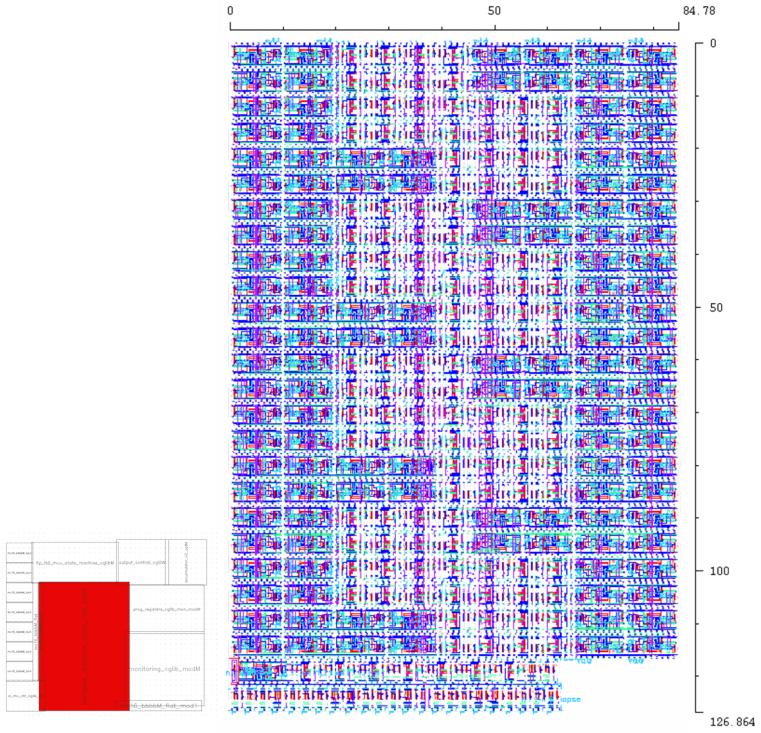


Figure 6.26: Cadence Virtuoso Layout for the Synapse Distance

Using the floorplan as the guide, the layout of the component was created by adding all necessary wires and pins to the design. Figure 6.26 shows all wires at all levels.

Once the layout was completed, the extracted version was created and put through the same tests as the schematic. This ensured that the functionality of the component was not affected and the timing was still correct. Details of these tests and the results can be found in Section 7.7. Once all of these tests were completed, the design of the synapse distance component was complete and the process of putting all of the pieces together to create a full element began.

6.10 Combined Element Design Flow

After all of the individual pieces of the element were created, the process of putting them all together to create the full element began. First, the RTL elaboration

was referenced, though the full element is purely structural and did not require the addition of any new logic. The elaboration is shown in Figure 6.27.

After studying the elaboration, it was determined that the only new components that were required other than those already designed were multiplexers for the sampling, which are marked as `input_mux`, `ltp_ltd_mux`, and `input_mux_fire_select` on the elaboration. Using this information, the schematic shown in Figure 6.28 was produced.

After the schematic was completed, it was tested extensively as detailed in Section 7.8. Once all of these tests were passed, the next step was to create the layout for the element. The primary goals for this layout are to keep the layout as compact as possible, to make it as close to square as possible, and to place components that share signals close to each other to minimize the required distance of wires. Based on this, the floorplan shown in Figure 6.29 was created. As the floorplan shows, all of the design goals for the element are met

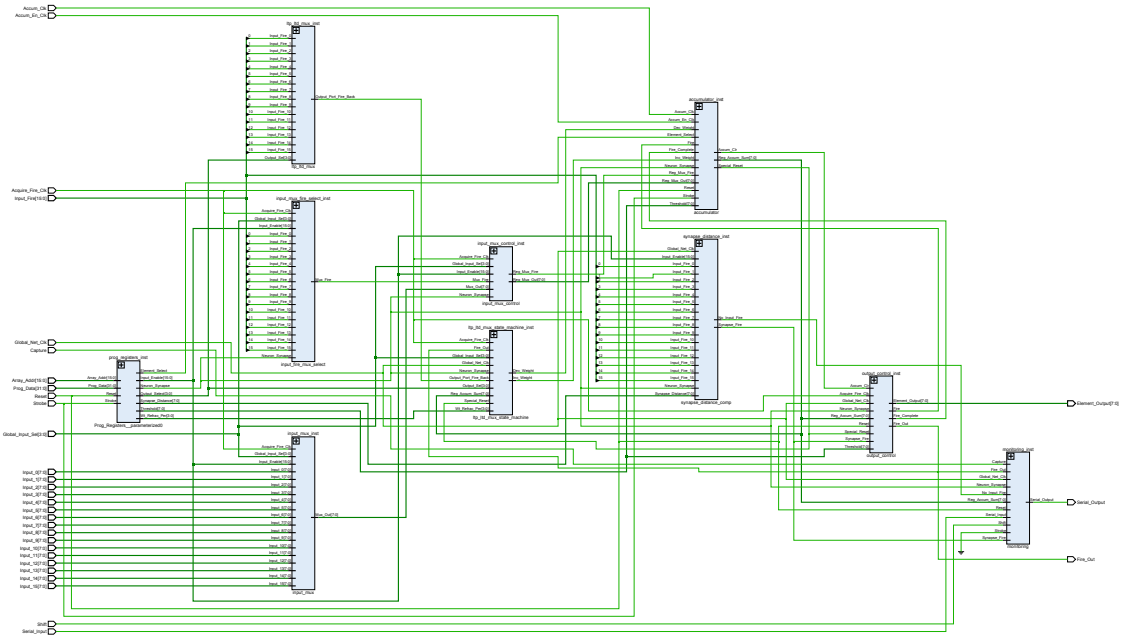


Figure 6.27: Vivado RTL Elaboration for the Element

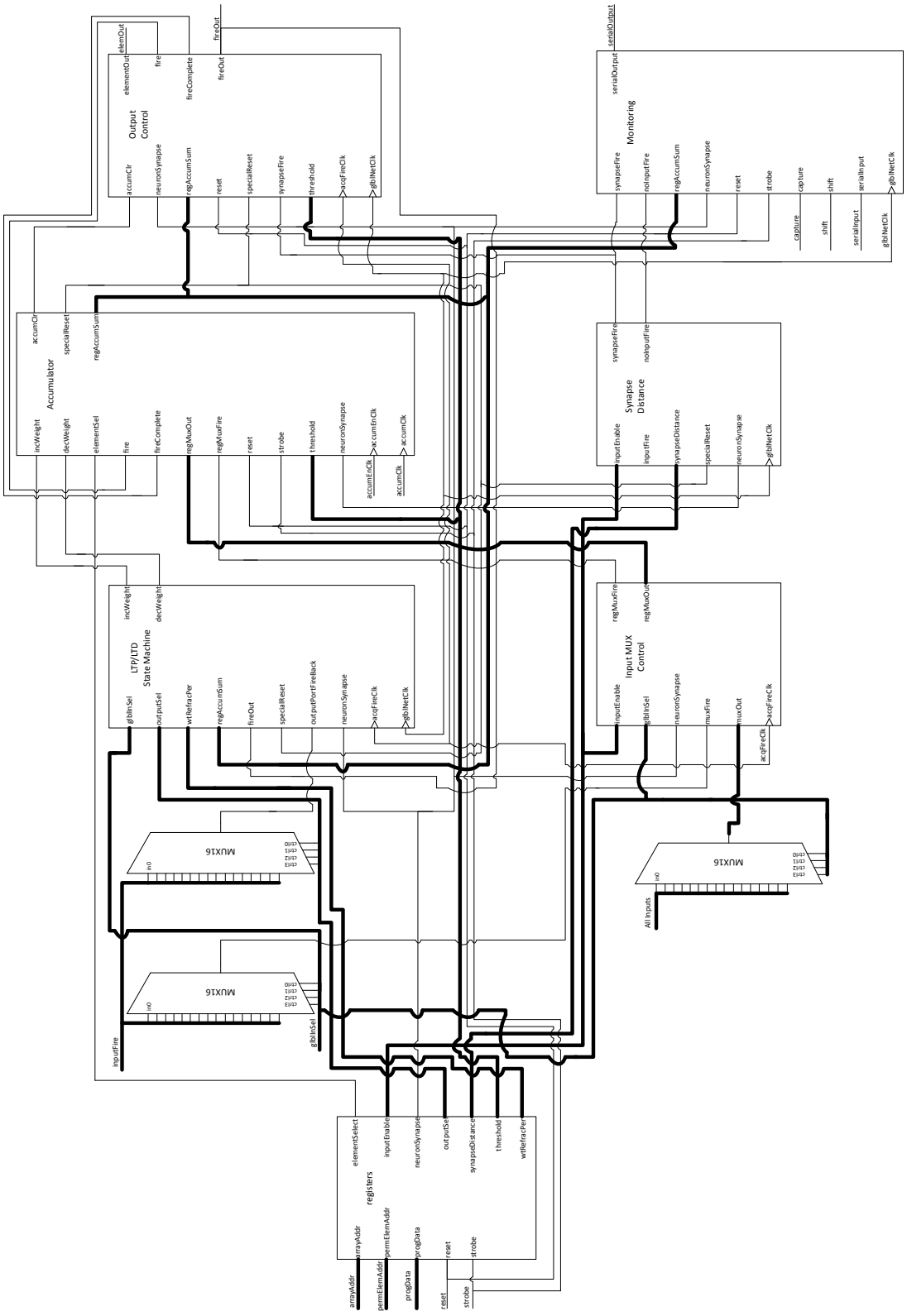


Figure 6.28: Schematic for the Element

After the placement was defined in the floorplan, the process of adding wires to connect the components began. Due to the large number of signals present inside the element and the high routing resource utilization already present inside each of the components, this was a challenge, but the layout for the element was successfully created. Figure 6.30 shows the high density of each of the three metal routing layers after completing the layout. Figure 6.31 shows the completed layout for the element with only the shapes added at the top level and Figure 6.32 shows the layout of the element with the full hierarchy.

After completing the layout of the element, the extracted version of it with all parasitics was created and it was subjected to the same tests that were run on the schematic. These tests ensured that the element functions correctly and meets the timing requirements of the design. The results of these tests are shown in Section 7.8.

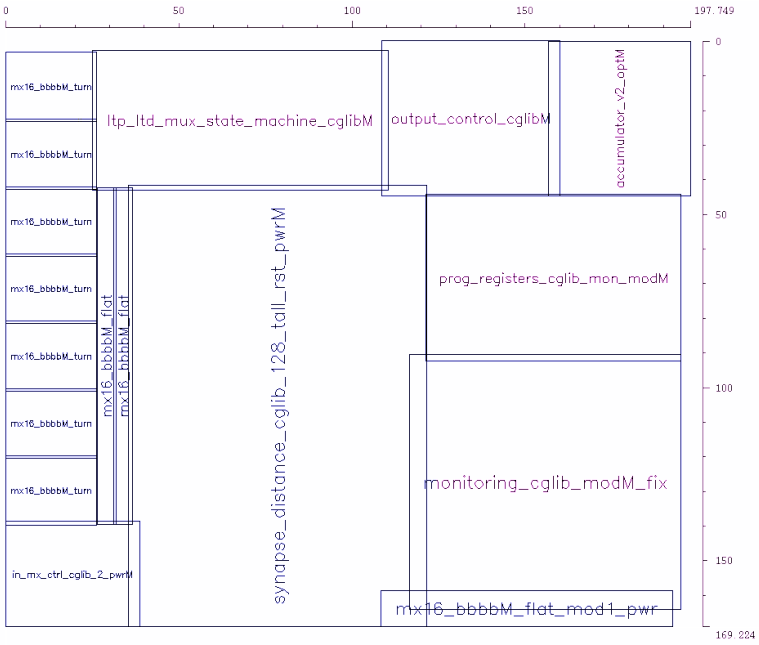


Figure 6.29: Cadence Virtuoso Floorplan for the Element

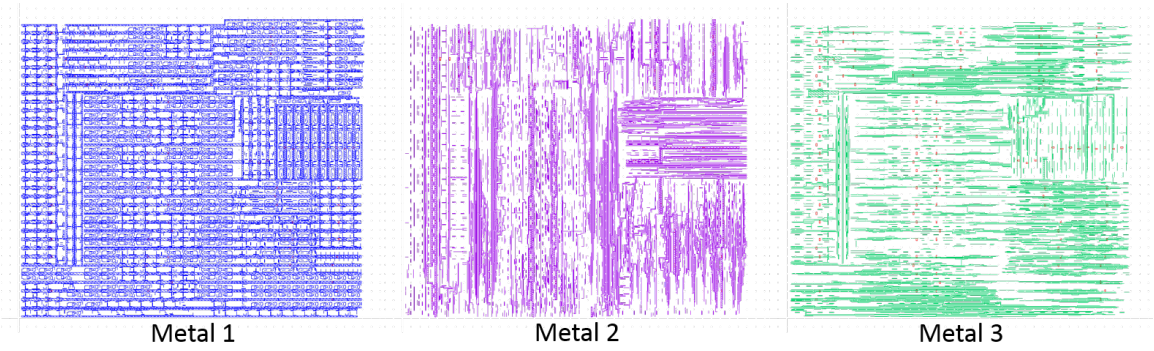


Figure 6.30: Element Routing Congestion

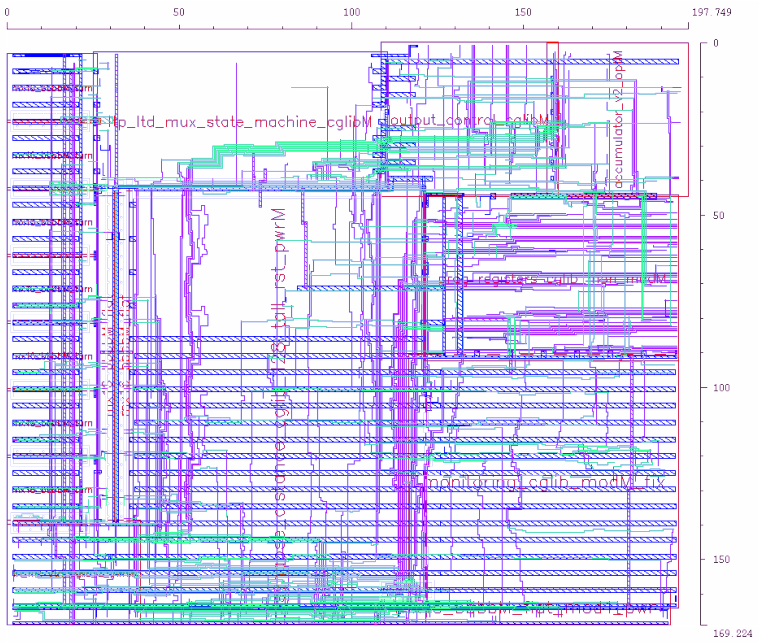


Figure 6.31: Cadence Virtuoso Layout for the Element (top level)



Figure 6.32: Cadence Virtuoso Layout for the Element (all levels)

6.11 ASIC Element Comparison

To provide a comparison for the custom element implementation discussed in this chapter, another version of the element was created using a fully automated ASIC design flow. This element is made of standard cells like the custom implementation, but the way those cells are connected was determined using automated tools beginning with a modified version of the VHDL code used to create the FPGA element. The modification was necessary to replace any proprietary Xilinx IP Blocks or FPGA primitives that were instantiated in the VHDL. That VHDL code was run through Synopsys Design Compiler to create a Verilog netlist describing how the standard cells are connected. Next, Cadence Encounter was used to automatically place the standard cells and route all of the signals between them. The resulting element layout is shown in Figure 6.33. As shown by the measurement scales in the figure, the ASIC element is 223.36 microns tall and 238.64 microns wide.

For comparison, the full custom element layout is shown in Figure 6.34 with measurement scales added. As shown by the scales, the custom element layout is 195 microns wide by 169.224 microns tall.

Based on the shown measurements, the areas of each element implementation were calculated, with the ASIC layout occupying approximately $53302\mu m^2$ and the custom layout occupying approximately $33000\mu m^2$. The increased size of the ASIC layout, 1.6 times larger than the custom layout, shows why the decision was made to create a custom layout for the element. ASIC tools, while very convenient and generally quicker than custom layout, are not able to create as compact of a layout as a person manually can. Part of the reason for this is because all of the standard cells must fit into uniform height tracks and because the tool does not rotate the cells. By comparison, the entire register file was rotated 90 degrees in the element layout and several other smaller components were rotated to make the layout more compact.

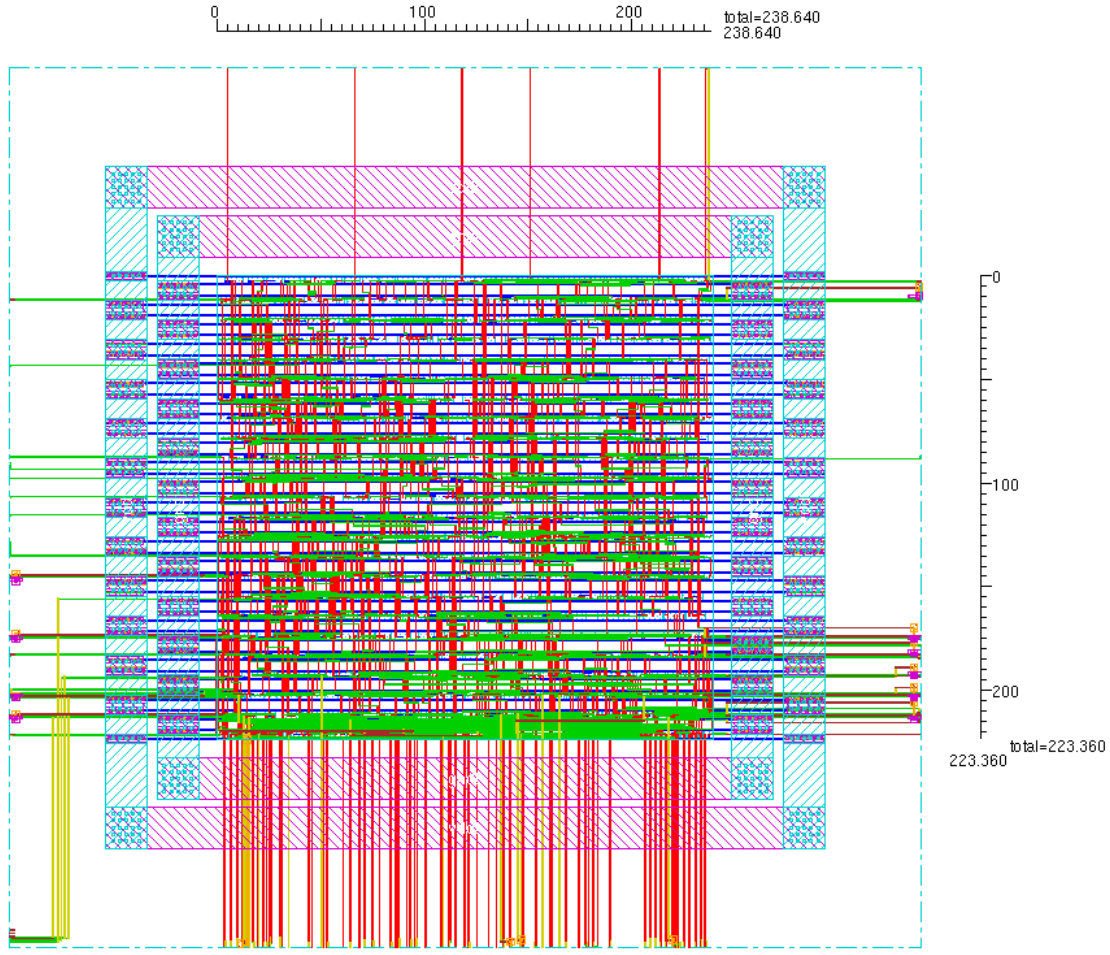


Figure 6.33: ASIC Element Layout from Cadence Encounter. Synthesized with Synopsys Design Compiler



Figure 6.34: Custom Element Layout from Cadence Virtuoso

Chapter 7

VLSI Element Testing

7.1 Accumulator Testing

For the accumulator, four tests were developed to ensure that the component functions correctly and meets the timing requirements of the element. Note that the signals in the waveforms for all tests are shown with the scale going from -0.25 V at the bottom to 1.25 V at the top. The first one implemented tests the accumulator during normal operating conditions with the element programmed as a neuron. In this test, the threshold of the neuron is three, so the threshold register is set at 0x7D (125) because the value programmed into the threshold register is defined as $Reg\_Value = 128 - Threshold$. As shown in Figure 7.1, an input of seven is received and added into the accumulator. This causes the value stored in the accumulator to go from 125 to 132. This meets the firing conditions of a value $\geq$ 128 and at least one fire received. Because of this, the `accumClr` signal goes high, signifying that the accumulator has reset its stored value and the element should fire. As the figure shows, the component passes this test. Additionally, the test was run on the extracted version of the component with all parasitics included and passed that test.

The second test run on the accumulator tests its behavior when the neuron is in its refractory period. The refractory period of a neuron is defined as the time during

which it is firing. During this time, the accumulator is able to accumulate charge, but that charge alone cannot cause it to fire. For this test, the same threshold and input value are used, but the element is already firing when the input is received, as shown in Figure 7.2. When the input is received, the accumulator again changes from a value of 125 to 132, but the `accumClr` signal does not go high because of this and the accumulator value does not get reset. The figure shows that the accumulator does pass this test and the extracted version also passed the test.

Next, a third test was run on the accumulator. This test was designed to verify that the accumulator behaves correctly when an overrun condition is met. This condition happens when the value in the accumulator would roll over in either direction but is not allowed to by the protection logic included in the component. The waveform for the schematic level results of this test is shown in Figure 7.3. In this test, the threshold register is set at 0x16 (22) and an input of 0xC0 (-64) is fed into the component. Without protections, 0x16 + 0xC0 would equal 0xD6, but, because the value stored in the accumulator is interpreted as an eight-bit unsigned integer, the result is instead forced to 0. As the figure shows, the accumulator handles

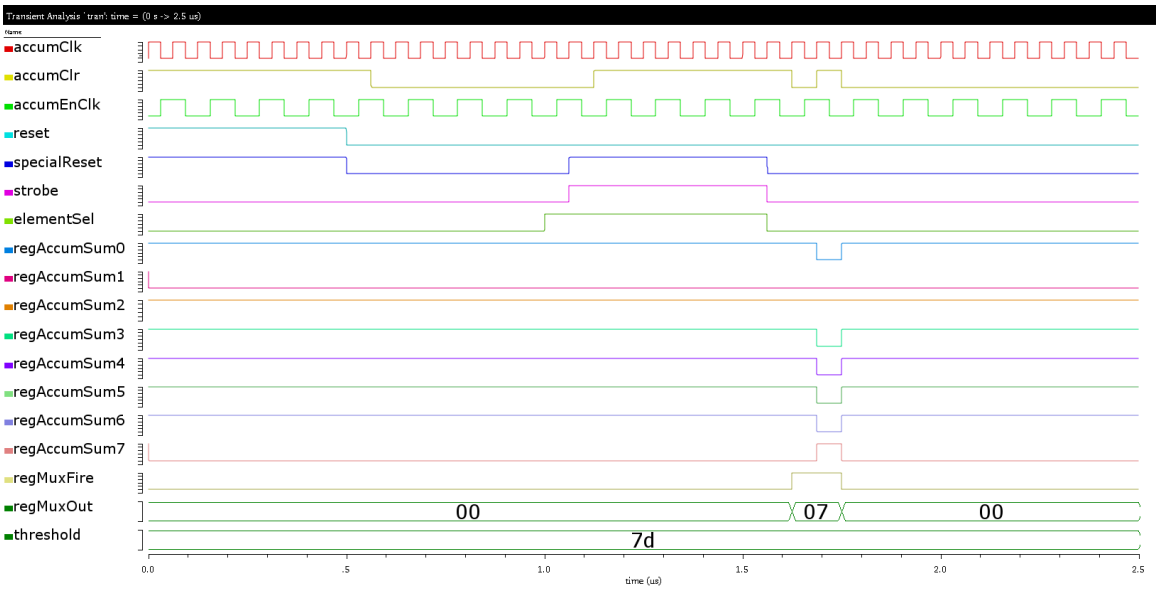


Figure 7.1: Accumulator - Neuron Operation Test

this scenario properly and the results of the extracted tests indicate that it also works properly with parasitics included.

The final test that was run on the accumulator was incrementing and decrementing the weight when the element is programmed as a synapse and a potentiation or depression condition is met. The schematic test result waveform is shown in Figure

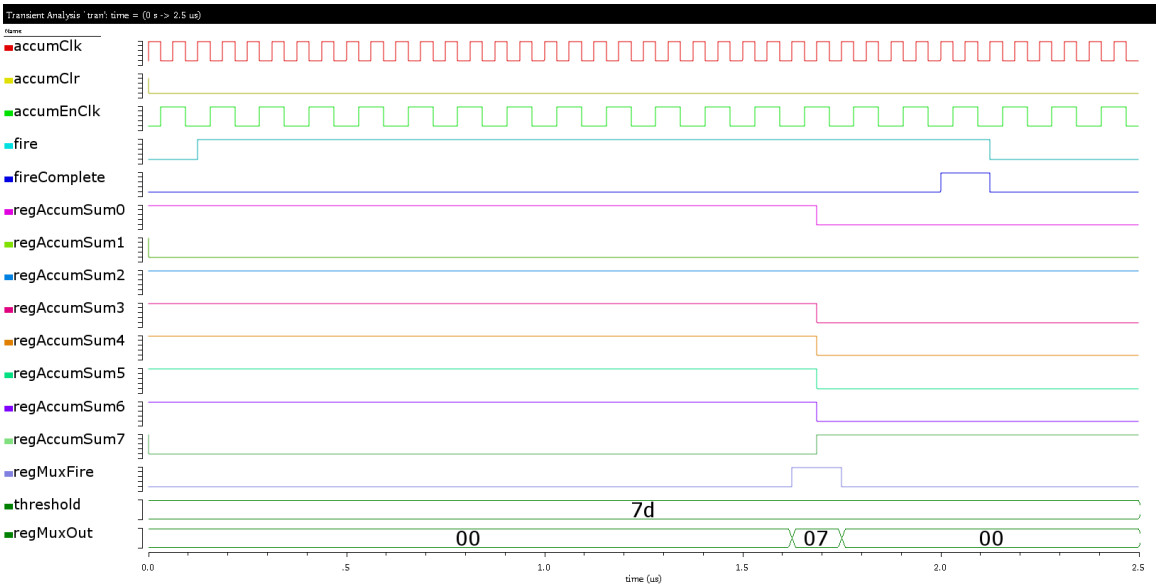


Figure 7.2: Accumulator - Neuron Refractory Test

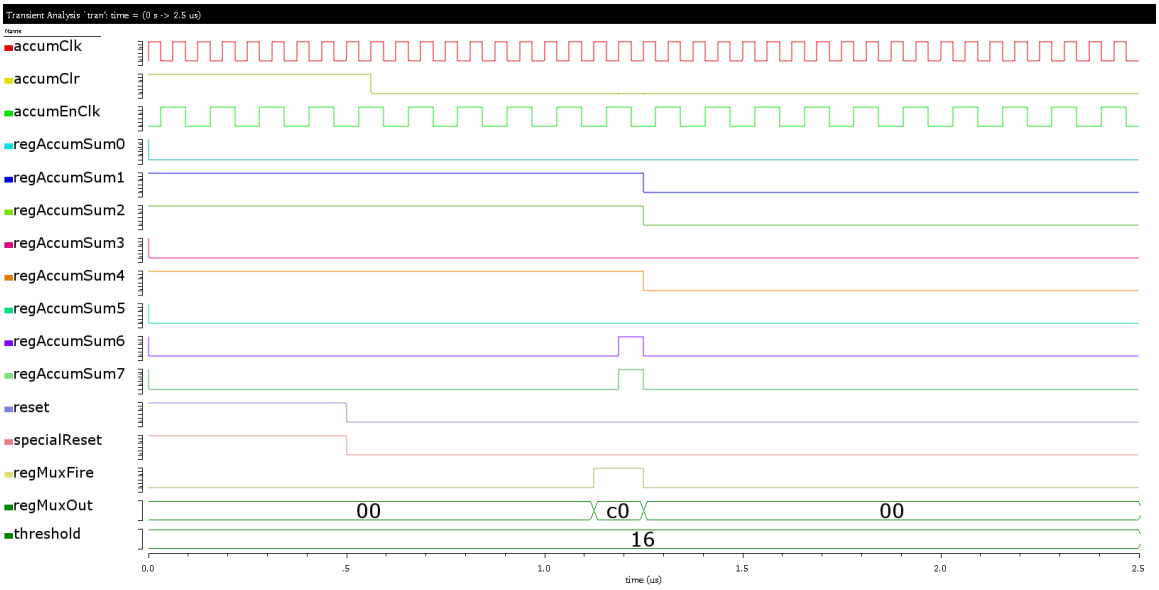


Figure 7.3: Accumulator - Overrun Test

7.4. In this test, the value initially in the accumulator is 0x1. Then, the `incWeight` signal is pulsed, which causes the value in the accumulator to change to 0x2. Next, the `decWeight` signal is pulsed and the value in the accumulator returns to 0x1. Based on this behavior, the component passed this test. In the extracted test shown in Figure 7.5, the initial value in the accumulator is 0x0 instead of 0x1, but the accumulator still behaves properly.

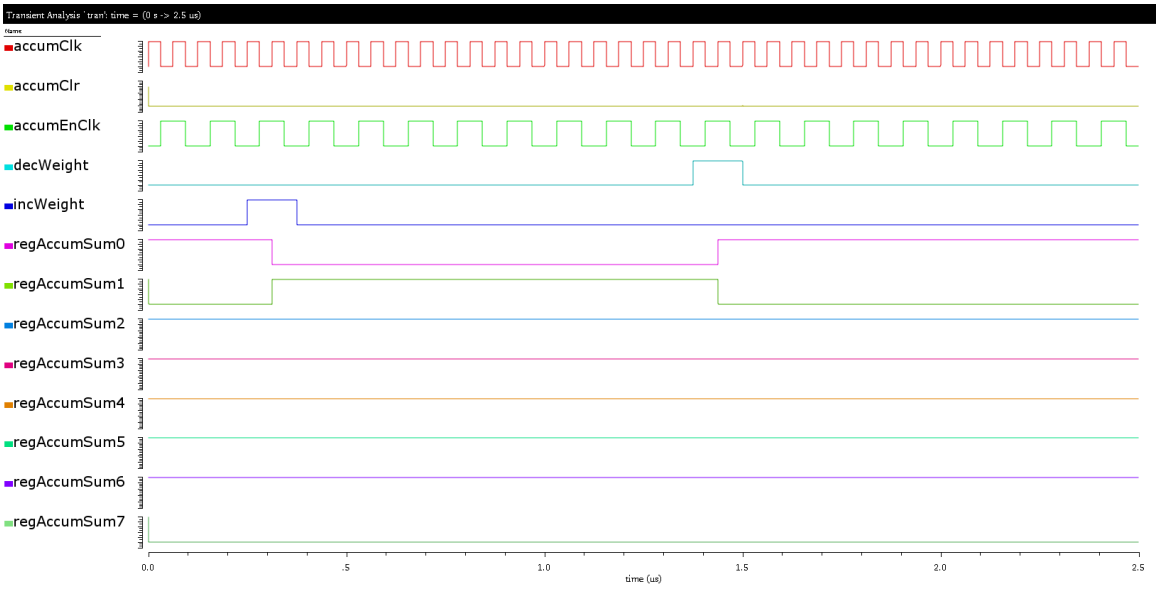


Figure 7.4: Accumulator - Potentiation / Depression Test - Schematic Level

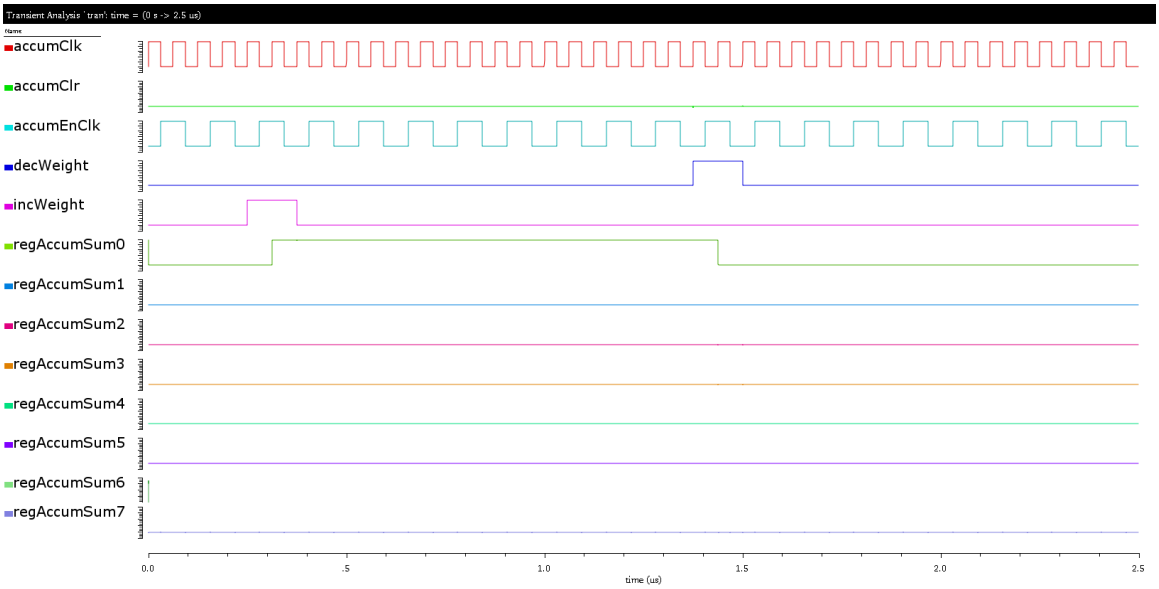


Figure 7.5: Accumulator - Potentiation / Depression Test - Layout Level

7.2 Input Multiplexer Testing

Because the input multiplexer is a simple component compared to the others, fewer tests were required to ensure it operates properly. In the first test, a single input is enabled and the component receives input on that port. When the correct value is driven onto the Global Input Select lines, the input value is passed through as shown in Figure 7.6. This test was also run on the extracted version of the component and the functionality and timing met requirements.

Next, a similar test was run where two input ports are enabled and inputs are being received on both of those ports. As before, when the correct value is driven out on the Global Input Select lines, the input value is registered and passed through as shown in Figure 7.7. The extracted version of the component also passed this test.

In the final test, an input port is enabled, but the element is configured as a synapse so nothing should be passed through to the outputs of the component. As shown in Figure 7.8, nothing happens when the appropriate value is driven onto the Global Input Select lines while input is sent to the enabled input port. This test was also run on the extracted version of the component and was passed.

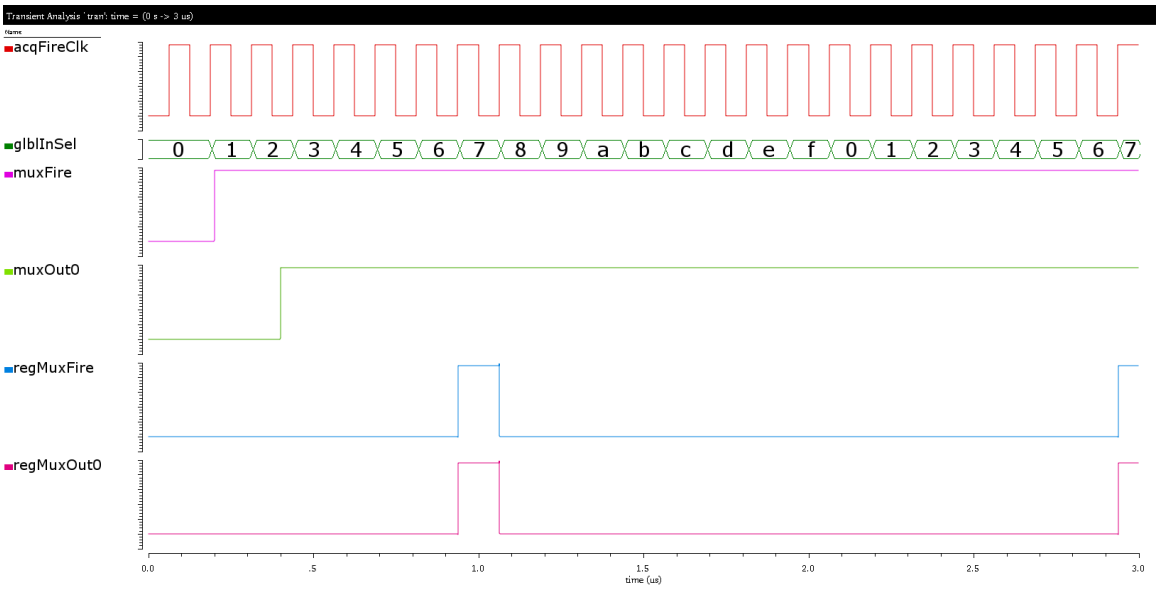


Figure 7.6: Input Multiplexer - Input on Enabled Input Test

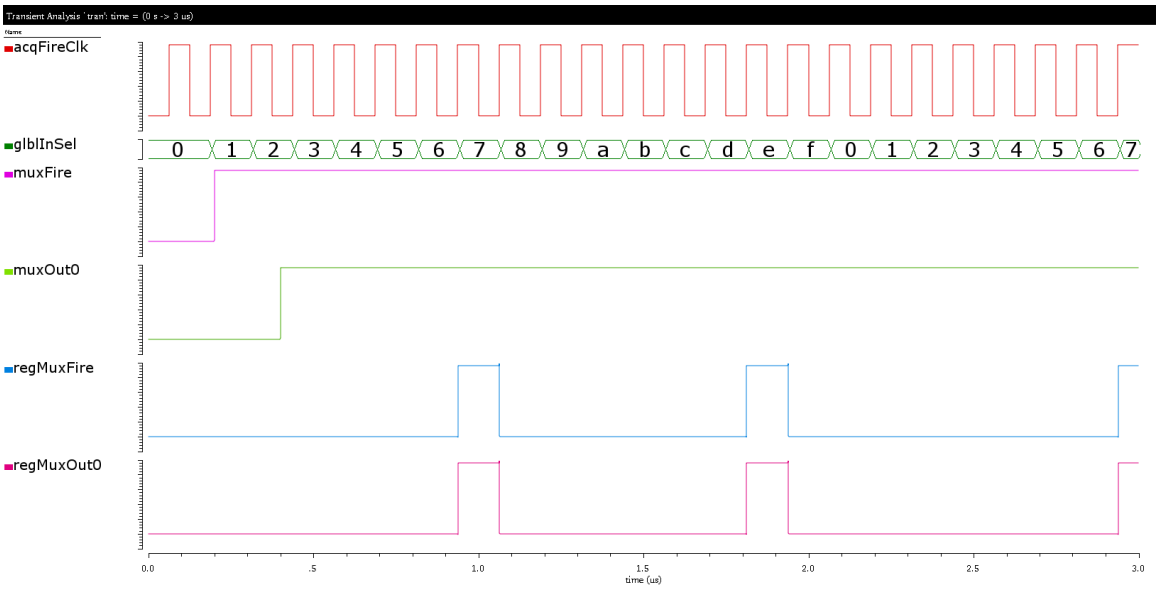


Figure 7.7: Input Multiplexer - Input on Multiple Enabled Inputs Test

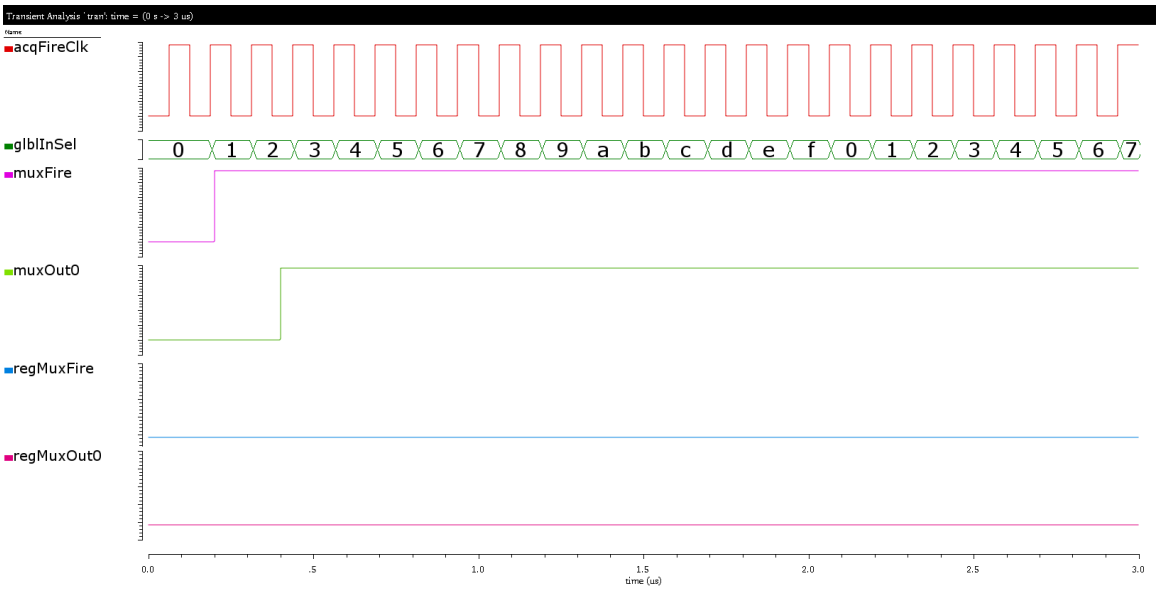


Figure 7.8: Input Multiplexer - Synapse Operation Test

7.3 LTP/LTD State Machine Testing

In the first test run on the LTP/LTD State Machine, the inputs are manipulated to create a situation where the conditions of potentiation are met. In Figure 7.9, Output Select is set to seven. As the figure shows, the element starts firing at the beginning of the Global Network cycle and `outputPortFireBack`, which is the fire signal of the element signified by Output Select, begins firing one cycle after input seven is sampled, as shown by the value of `glblInSel`. Because of this, `incWeight` goes high for a single Acquire/Fire Clock cycle. When this test was run on the extracted version of the component, it also behaved as expected and met timing requirements.

In the second test, the conditions for depression are met. As shown in Figure 7.10, the setup for this test is similar to the first one, with the element starting to fire at the beginning of the Global Network Cycle. In this test, `outputPortFireBack` begins to fire one cycle earlier, which is earlier than it could start firing if the element being tested caused it to fire, so `decWeight` is pulsed for a single Acquire/Fire Clock cycle. As before, this test was run on the extracted version of the component and was passed.

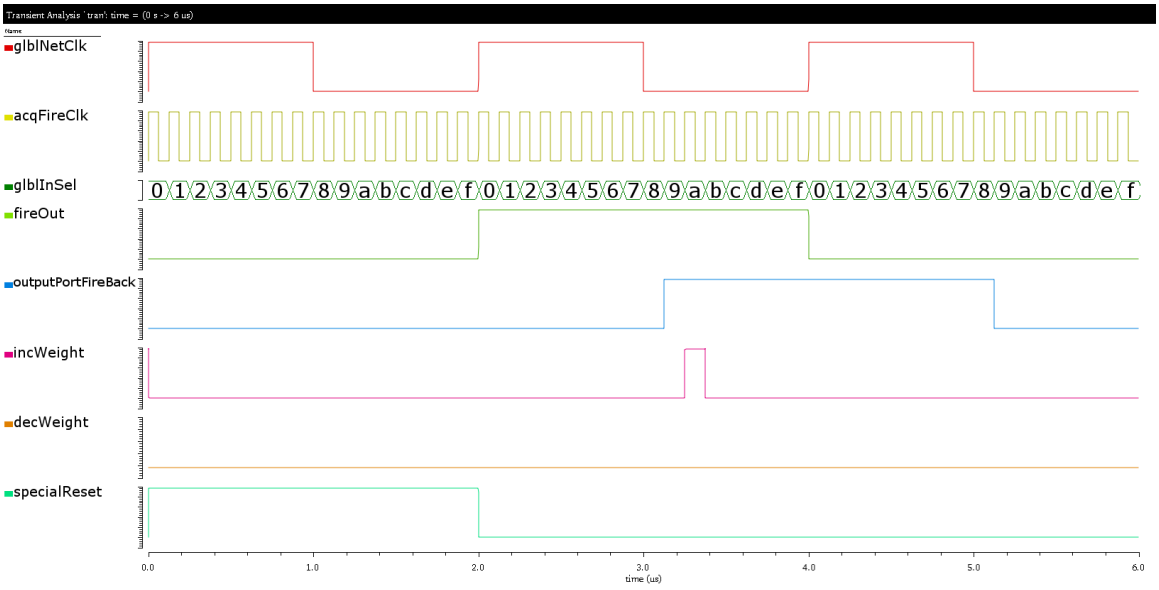


Figure 7.9: LTP/LTD State Machine - Potentiation Test

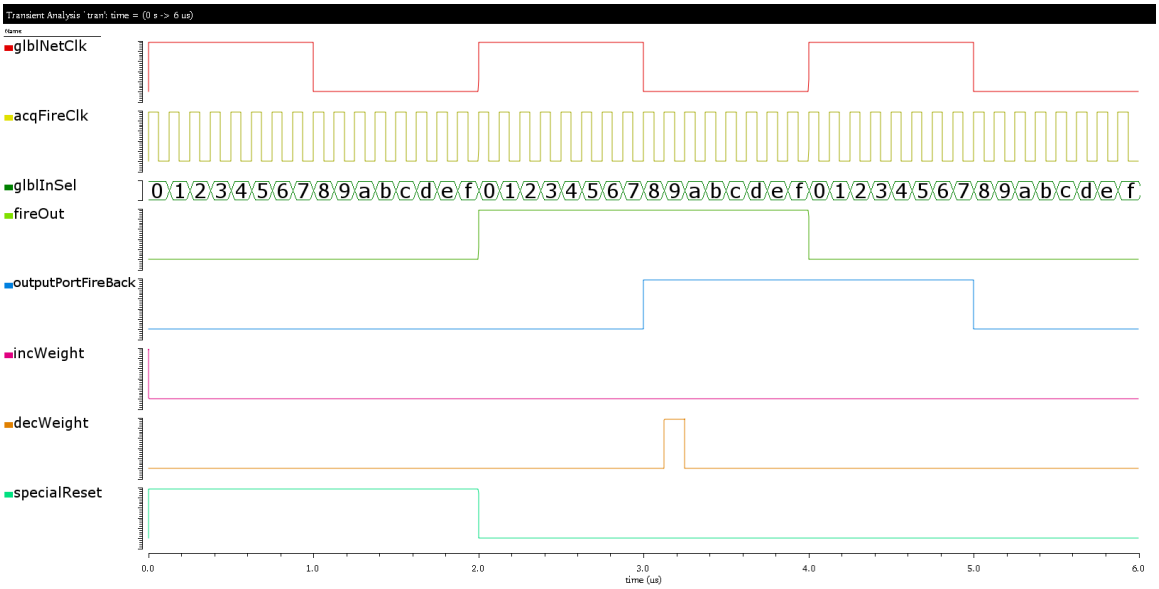


Figure 7.10: LTP/LTD State Machine - Depression Test

7.4 Monitoring Testing

In the first test run on the monitoring component, the element is programmed as a synapse with a single fire already having been sent out by the element and another fire currently stored in the distance register. As Figure 7.11 shows, both the `fireOut` and `noInputFire` signals are high on the cycle before the capture signal is pulsed. The `fireOut` signal is the actual output of the element and the `noInputFire` signal is the input to the synapse distance function. Based on this, there should be one fire counted in the output fire counter and one counted in the synapse distance register counter. On the next cycle after the fires are shown, the capture signal is raised and the strobe signal is pulsed, loading the internal shift register with the values and reseting the values stored in the counters. After that, shift is raised and the strobe signal is repeatedly pulsed. The following bitstream is outputted by the `serialOutput`: "01111101000000000000000100000001". After being split into the values stored in the register, this represents the value in the accumulator being "01111101", the fire counter being "0000000000000001", and the synapse distance

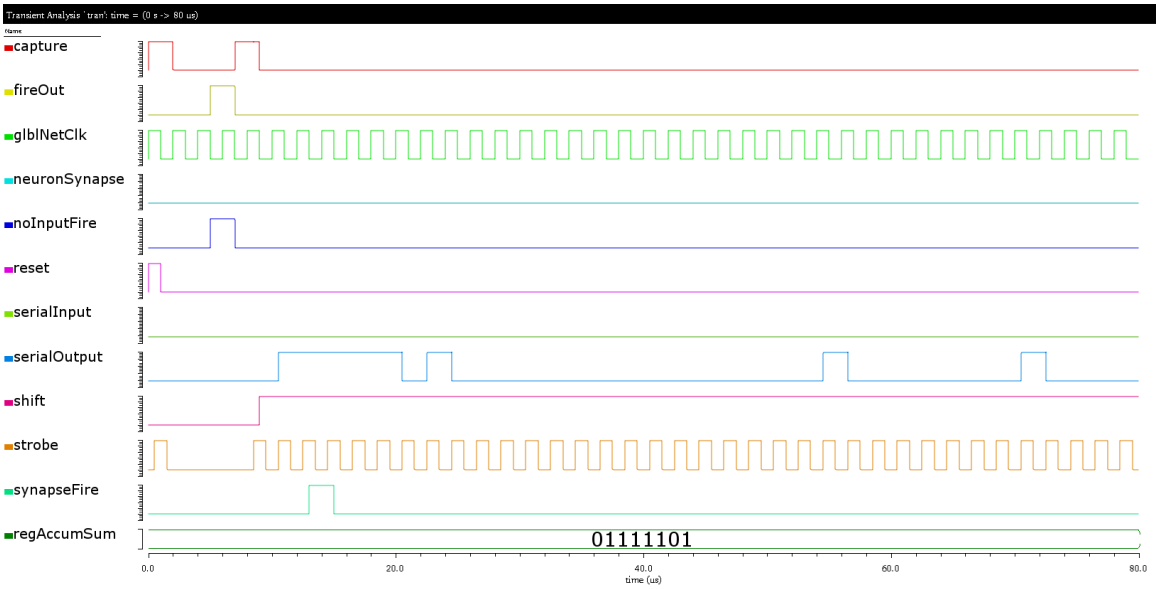


Figure 7.11: Monitoring - Synapse, Single Fire Sent, Single Fire in Distance Register Test

counter being "00000001". All of these values are correct and the results were also verified on the extracted version of the component, so it passed this test.

In the second test run on the monitoring component, shown in Figure 7.12, capture is pulsed again after the shift output has already started to reload the shift register before all of the existing data has been shifted out. In this test, the `serialOutput` begins with "011" before being interrupted with another capture signal. After the second capture signal, the `serialOutput` continues with "01111101000000000000000000000000". This represents the correct value for the accumulator and shows that both the fire counter and the synapse distance counter were reset to zero after the first capture signal. This same test was also run on the extracted version of the component and the results were also correct.

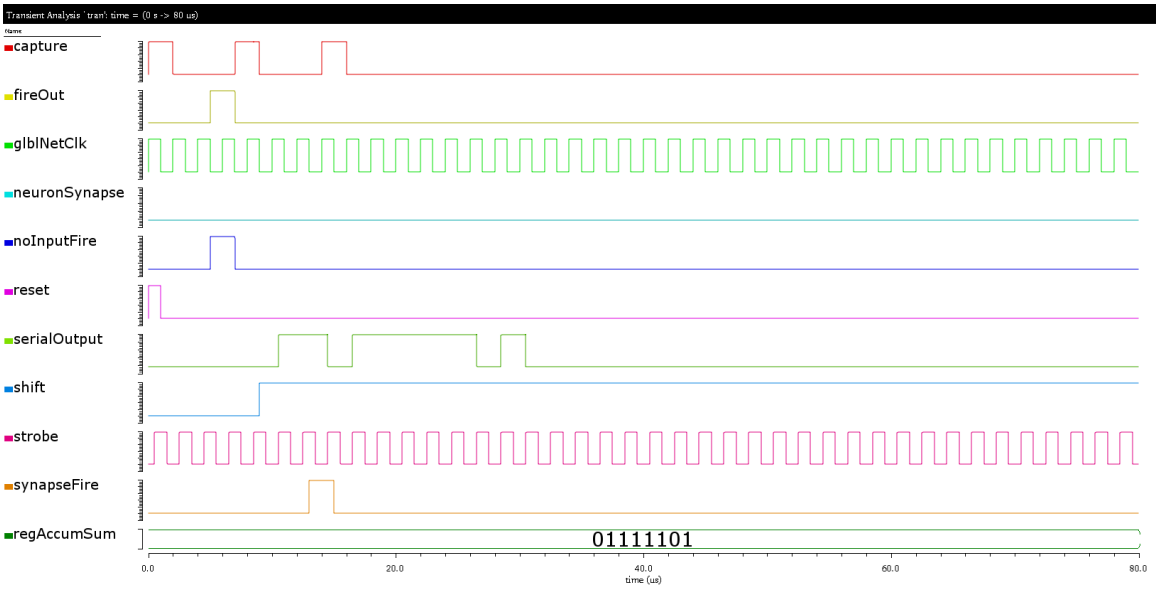


Figure 7.12: Monitoring - Reset After Partial Output Test

7.5 Output Control Testing

For the Output Control component, two sets of conditions were tested. The first was when the element is programmed as a neuron and the accumulator indicates that it is time for the element to fire. As Figure 7.13 shows, the `accumClr` signal gets pulsed for a single Acquire/Fire Clock cycle, which triggers the Output Control component to create an output fire. As the output value, the component drives out the value stored in the threshold register, which is 8'b0111111 (127) in this case. After the appropriate number of Acquire/Fire Cycles (15), the `fireComplete` signal pulses, causing the element to stop firing on the next cycle. This results in the overall time spent firing being 16 Acquire/Fire Clock cycles, or one Global Network Clock cycle, which is the correct amount of firing time. After running this test on the schematic of the component, the extracted version of the component was also tested and passed the test.

Next, the Output Control component was tested to verify its operation when the element is programmed as a synapse. In this configuration, the component simply passes through the fire signal received from the Synapse Distance component and

drives the current value in the accumulator out on the output lines. In Figure 7.14, the third signal from the top is the fire signal received from the Synapse Distance component. When that signal goes high, the appropriate signal is driven out on the output lines and the fire is passed through as the `fireOut` signal.

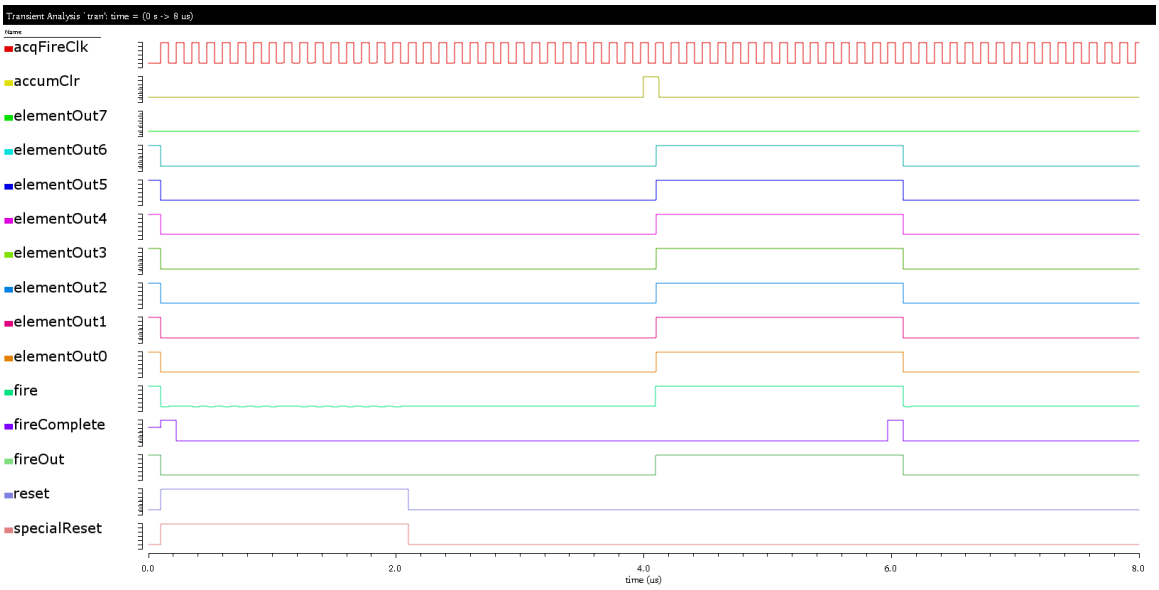


Figure 7.13: Output Control - Neuron Fire Test

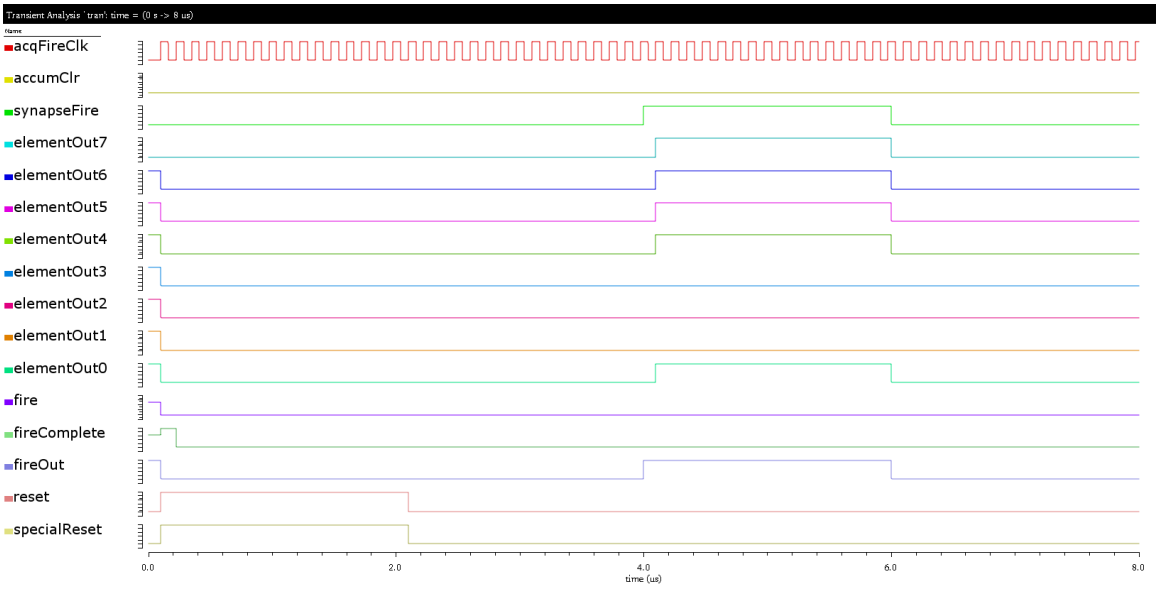


Figure 7.14: Output Control - Synapse Fire Test

7.6 Registers Testing

In the first test run on the Registers component, all of the registers are programmed using two strobe pulses. As Figure 7.15 shows, all of the elements were programmed at the appropriate time, with the `neuronSynapse`, `outputSel`, `synapseDistance`, `threshold`, and `wtRefracPer` signals programmed with the first strobe pulse and the `inputEnable` signal programmed with the second strobe. Additoinally, the `elementSelect` signal was driven high to signify that the address being driven out to the array matches the address of the element. Upon testing the extracted version of the component, it was found to operate properly as well.

In the second test, an address different than that of the element address is driven out on the lines and the same programming data and strobes are sent. As Figure 7.16 shows, none of the registers are programmed and the `elementSelect` signal is driven low, which is the correct behavior.

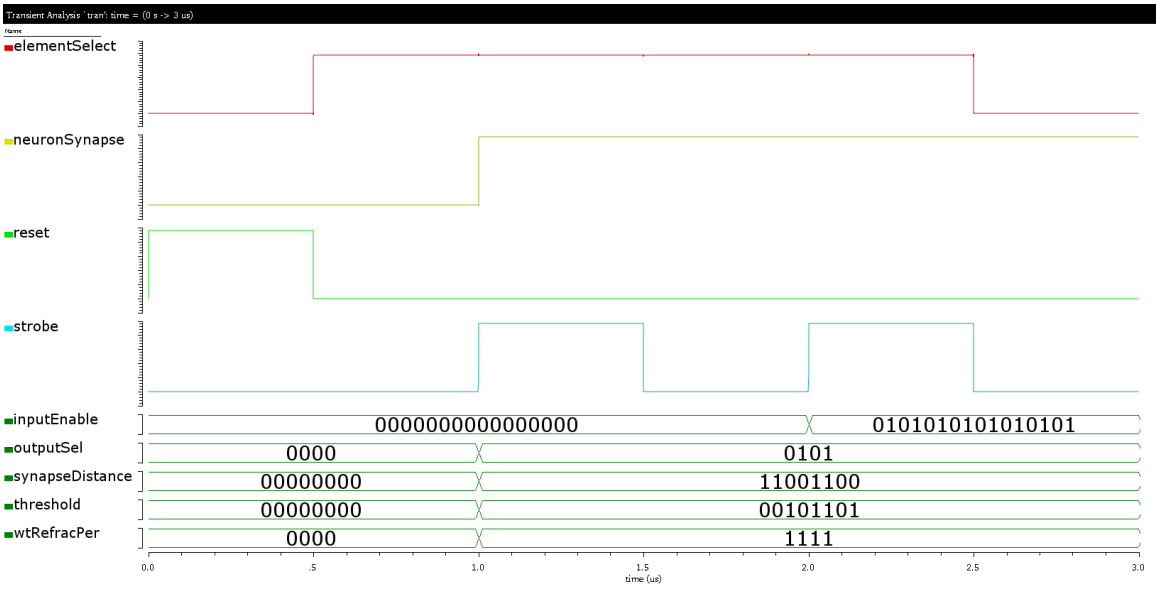


Figure 7.15: Registers - Normal Programming Test

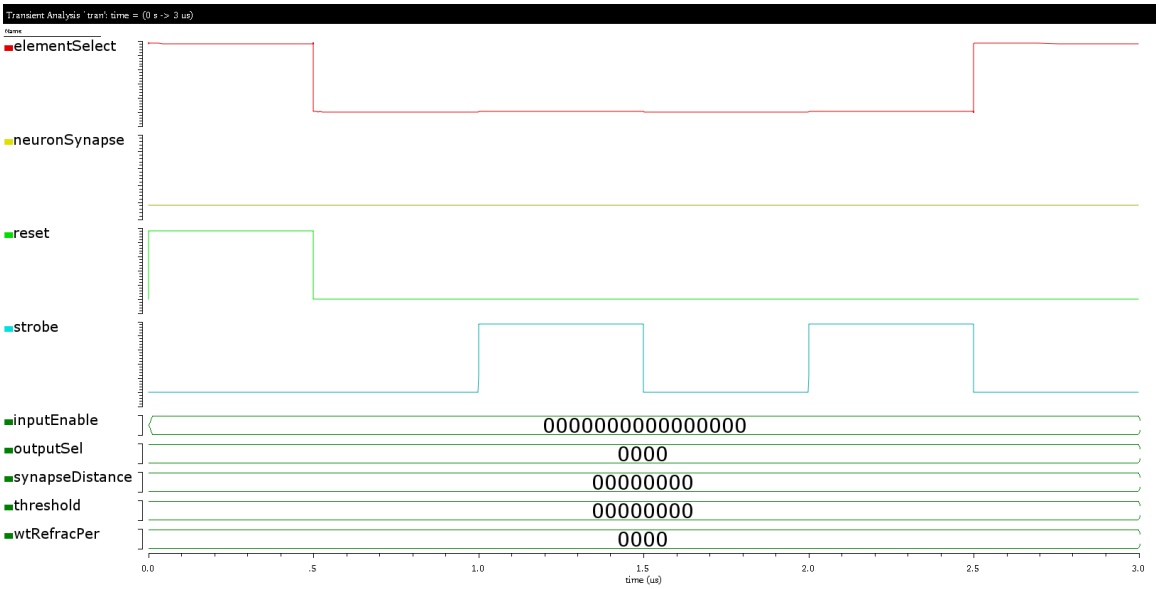


Figure 7.16: Registers - Wrong Address Programming Test

7.7 Synapse Distance Testing

For the Synapse Distance component, several tests were run due to the higher number of subcomponents and higher complexity of the component. The first test run was to receive an input on the enabled input with a Synapse Distance of zero. As shown in Figure 7.17, the input gets sampled on the first rising edge of the Global Network Clock (the top signal) and is then driven out on the next rising edge of the Global Network Clock. Testing on the extracted version of the component found that it mirrored the behavior of the schematic test.

In the second test, shown in Figure 7.18, the input is received and sampled at the same time, but the Synapse Distance is set to one so the output of the component is delayed by one cycle compared to the previous test. Based on the figure, the component behaved correctly during this test, as did the extracted version of the component.

In the third test run on the component, the input received is on an input port that is not enabled. Because of this, no output should be seen from the component. As Figure 7.19 shows, the component behaved correctly and did not drive out anything.

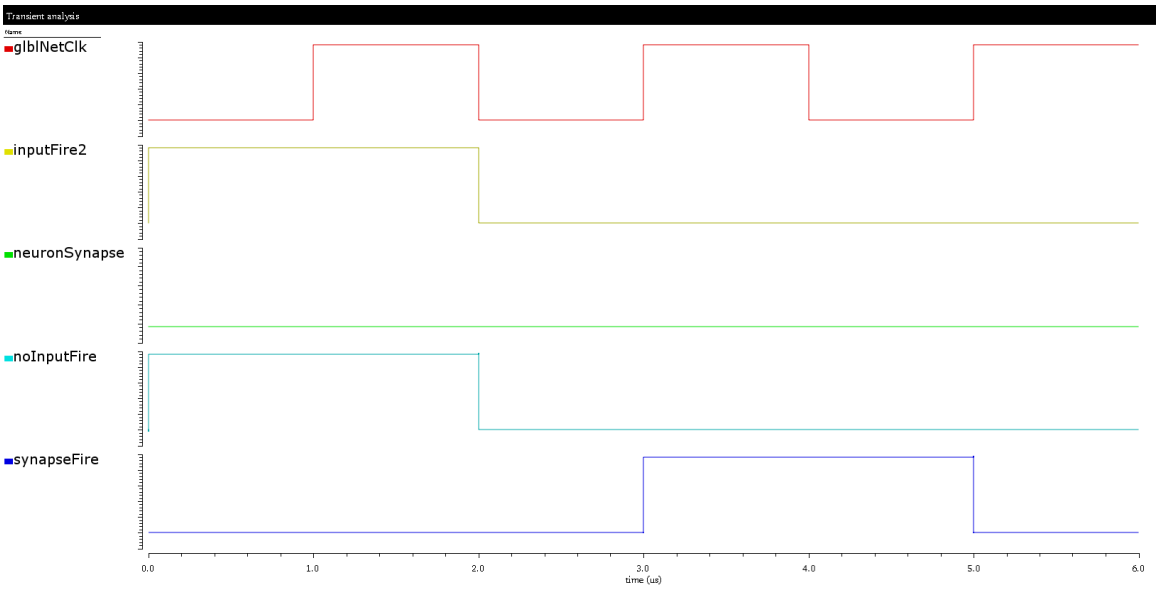


Figure 7.17: Synapse Distance - Single Input with Zero Distance Test

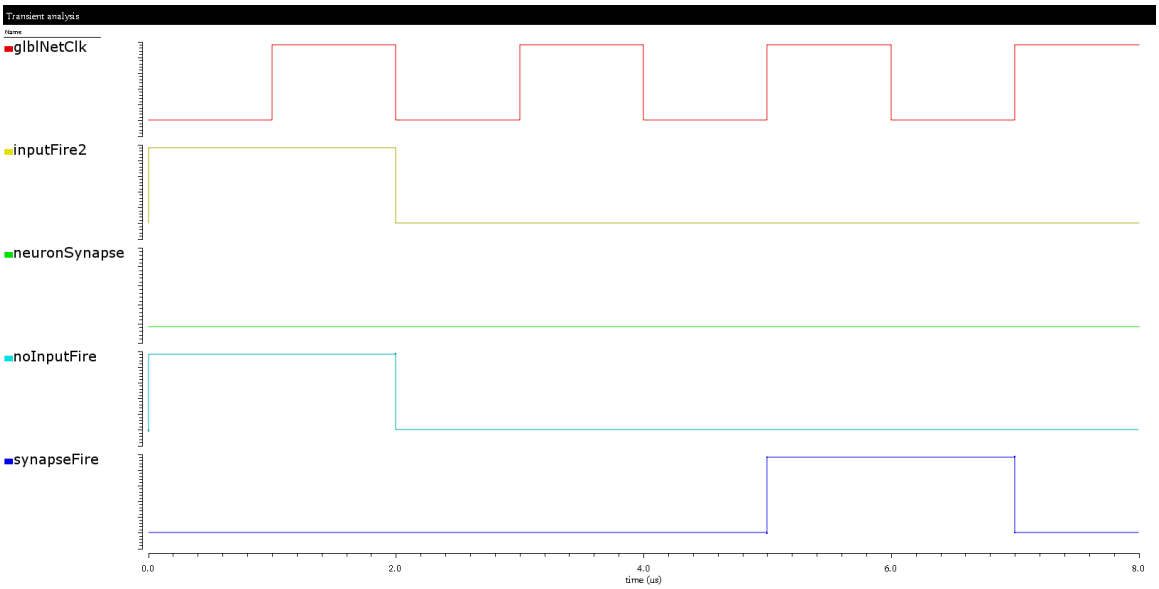


Figure 7.18: Synapse Distance - Single Input with One Distance Test

Though several spikes are visible in the schematic, the scale of the spikes is μV so they remain a logic '0'.

The fourth test run on the component tests the ability to have multiple fires in the shift register at the same time. As Figure 7.20 shows, multiple fires are received with the Synapse Distance set to two. After the appropriate number of cycles have passed, the output starts pulsing with multiple pulses in the shift register at the same time.

Finally, in the fifth test, multiple fires are in the shift register, but the value is reset before all of them come out, clearing the internal state of the shift register. Figure 7.21 shows the details of this test. As before, a train of input fires is sent into the component, and the first one does make its way out, but then `neuronSynapse` is changed from a logic '0' to a logic '1'. This resets the shift register and the rest of the fires do not pass through it.

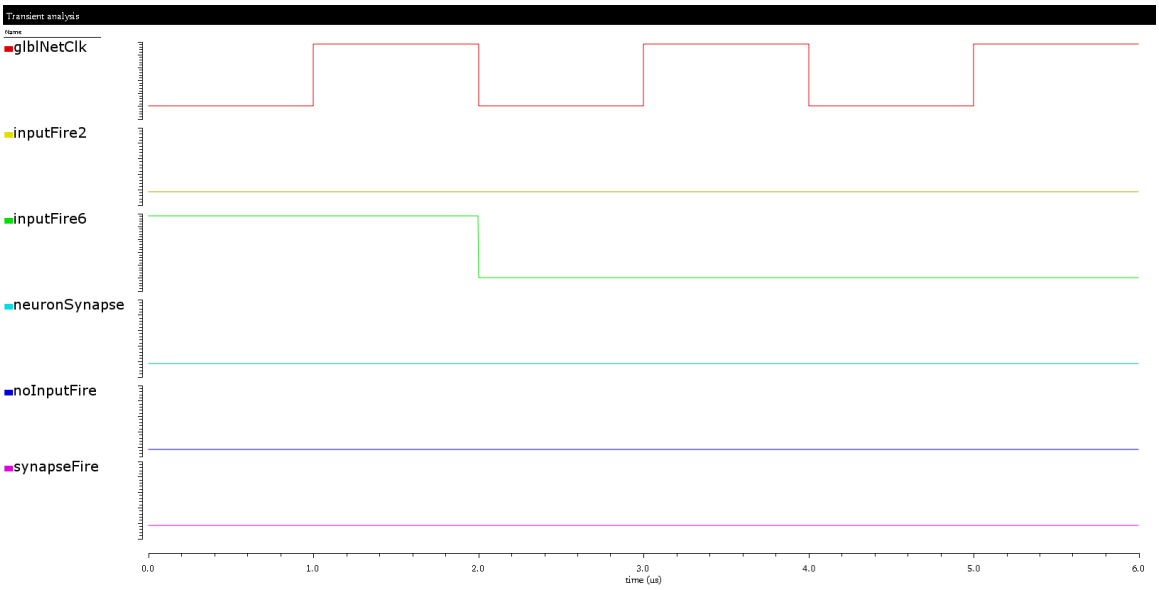


Figure 7.19: Synapse Distance - Disabled Input Test

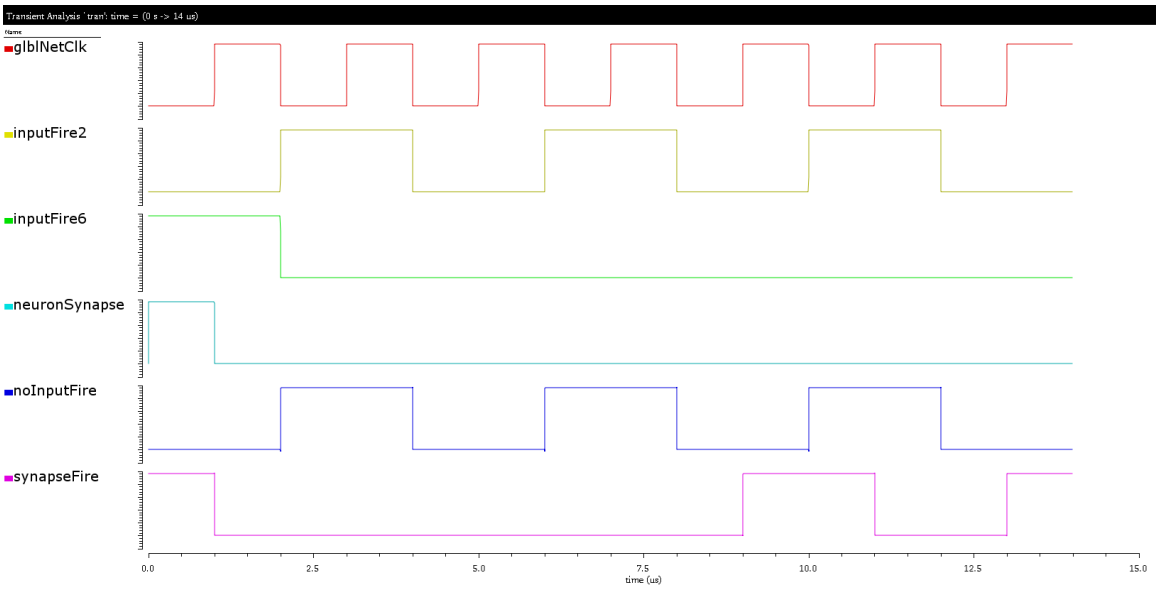


Figure 7.20: Synapse Distance - Multiple Stored Fires Test

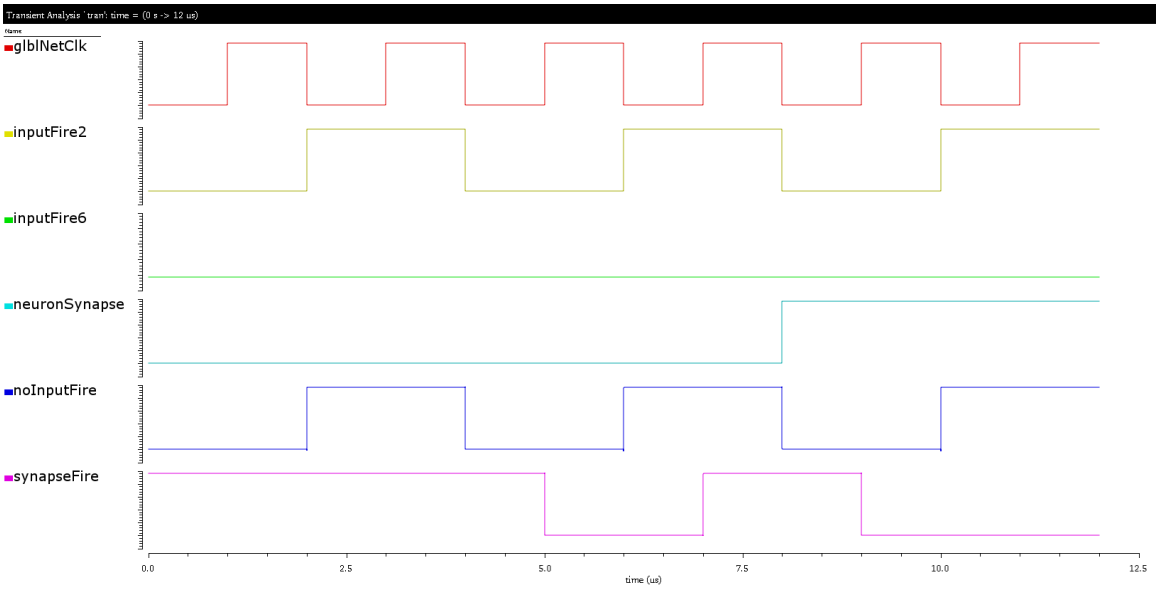


Figure 7.21: Synapse Distance - Reset Stored Fires Test

7.8 Combined Element Testing

Many tests were run on the full element to ensure that all of the features work properly and meet the requirements of the design. In the first test, the element is programmed as a neuron and a single input fire causes an output fire. As shown in Figure 7.22, input two of the element is driven with weight of seven after four microseconds of testing. The neuron is programmed to have a threshold of three, so this input causes the element to fire. Looking at the value of `glblInSel`, the element begins firing at the end of when its value is three, which is the correct time for an input received on input two.

In the next test, the element is still programmed as a neuron with a threshold of three, but multiple inputs are received on the same input. The total value of these is enough to cause the element to fire. As shown in Figure 7.23, input two is driven with a weight of one after four microseconds and then a weight of seven two microseconds later. After both of these weights are received, the neuron fires.

The third test on the element tests the ability of a neuron to have a zero threshold. In this test, shown in Figure 7.24, the input fire signal for input two is driven high

with a zero weight after four microseconds and this does cause the element to fire out.

The next test on the element checks the ability of a neuron to receive negative input fire weights. In this test, the element is programmed as a neuron with a threshold of zero. The waveform shown in Figure 7.25 shows that a weight of 0xFB (-5) is received

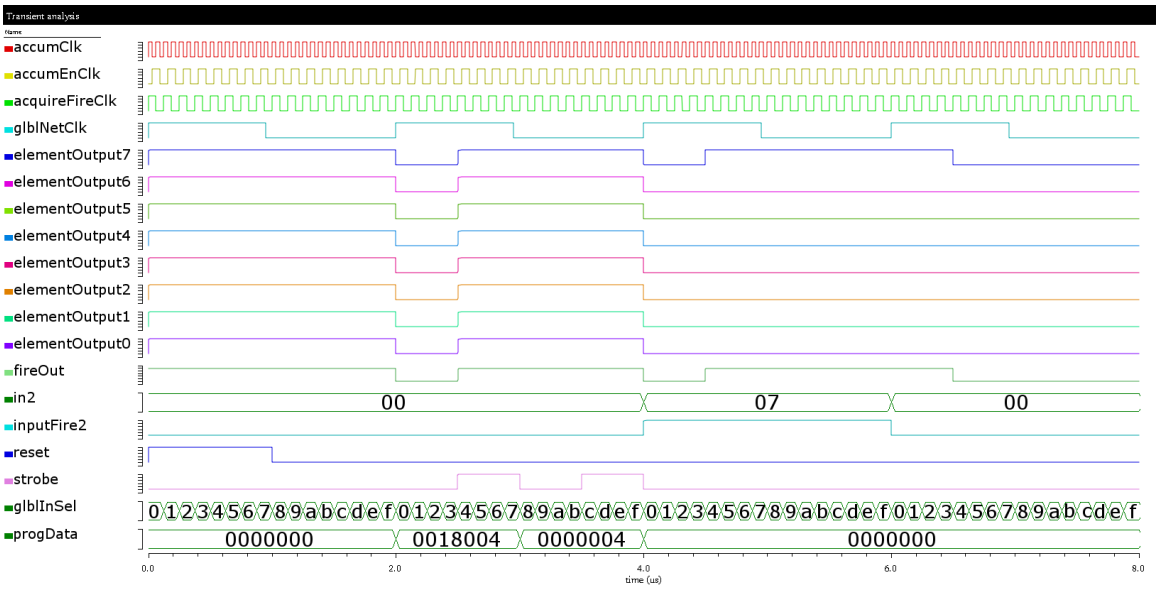


Figure 7.22: Combined Element - Neuron, Fire Causes Output Test

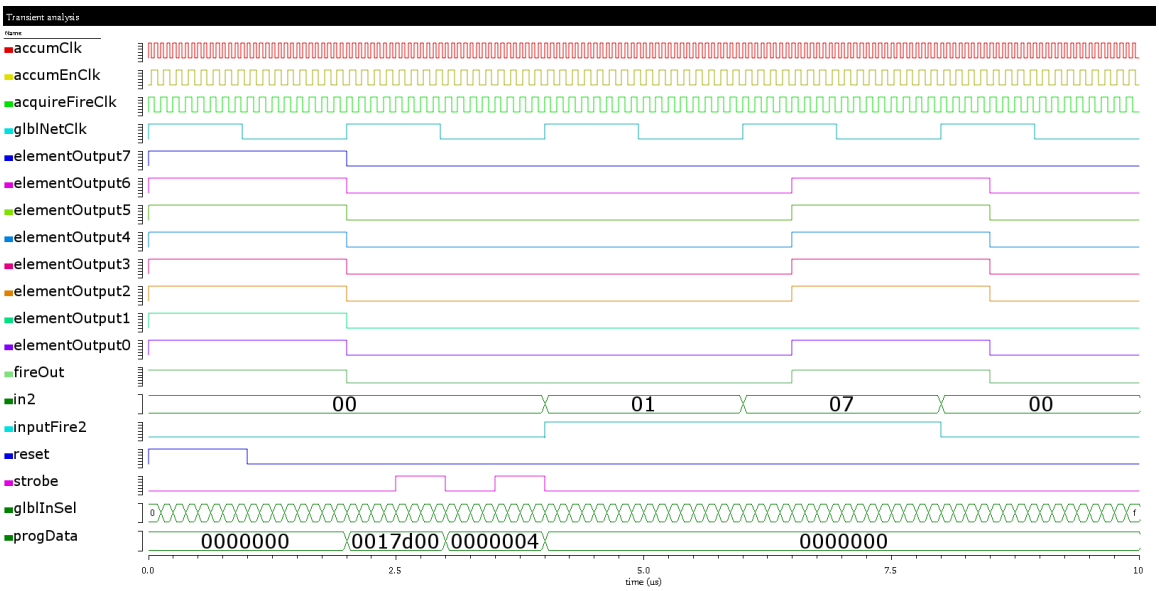


Figure 7.23: Combined Element - Neuron, Two Inputs Cause Fire Test

on input at time $4\mu s$ and then a weight of eight is received at time $6\mu s$ which causes the neuron to fire. This shows that the neuron handles the input correctly and interprets the first input as a negative weight.

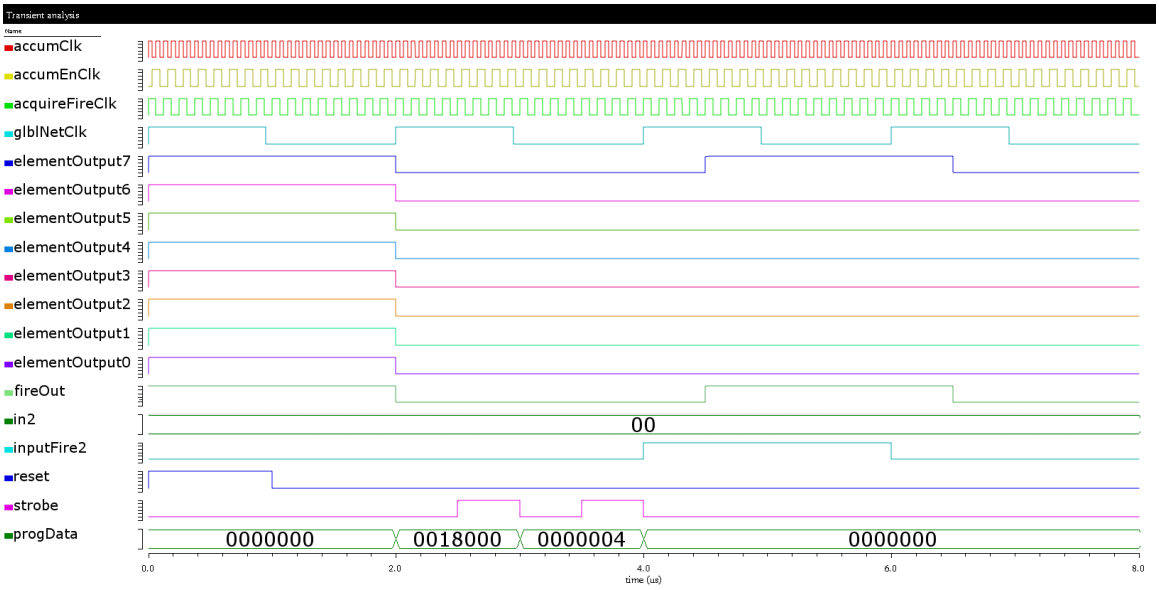


Figure 7.24: Combined Element - Neuron, Zero Threshold, Zero Weight Input Causes Fire

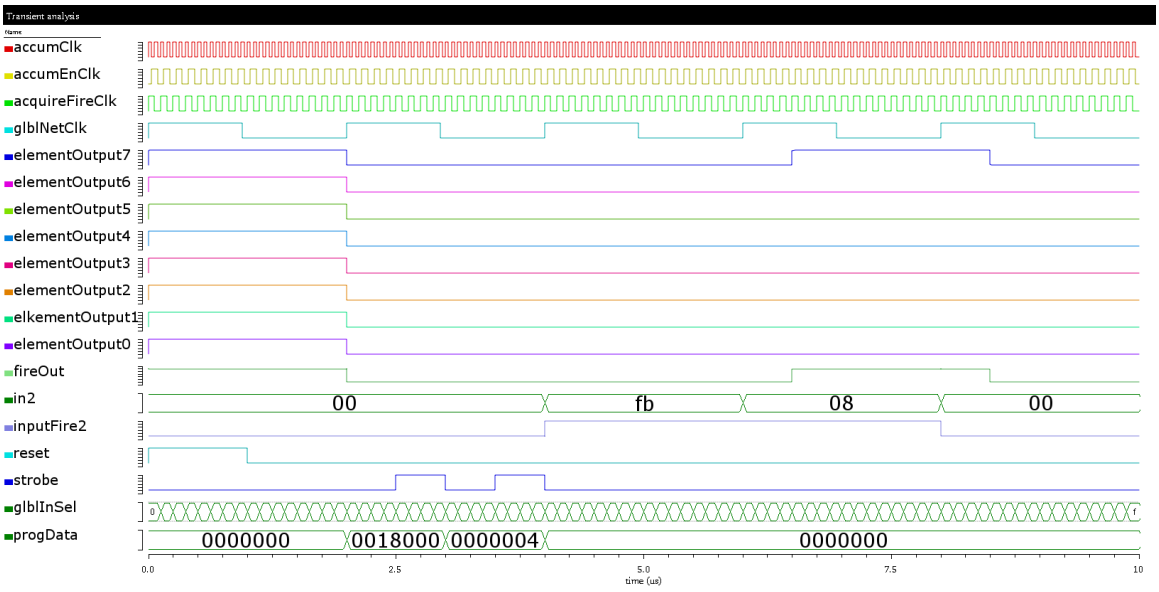


Figure 7.25: Combined Element - Neuron, Zero Threshold, Negative Input Test

In the fifth test performed on the element, it is programmed as a synapse with a distance of zero. Six microseconds into the test, an input fire with a weight of 127 is driven into input two of the element. Because this input starts at the same time as a rising edge of the Global Network Clock, it is not sampled until the next rising edge. When it is sampled $8\mu s$ into the simulation, it enters the synapse distance function shift register and is then driven at the next Global Network Clock cycle due to the distance being zero. The waveform for this simulation showing the inputs and outputs is in Figure 7.26.

In the next test, the element is also programmed as synapse and the test is designed to test the Long Term Depression feature of the synapse. The waveform showing the details of this test are shown in Figure 7.27. First, an input weight of 127 is driven into input two of the element. This causes the element to fire $4\mu s$ later with an output weight of seven. At the same time as the element begins to fire, a weight of 127 is driven into input three, which is the monitored input specified by the Output Select register. Because it starts firing at the same time as the element begins firing, it could not have been caused by the synapse, so the synapse should depress. To view the

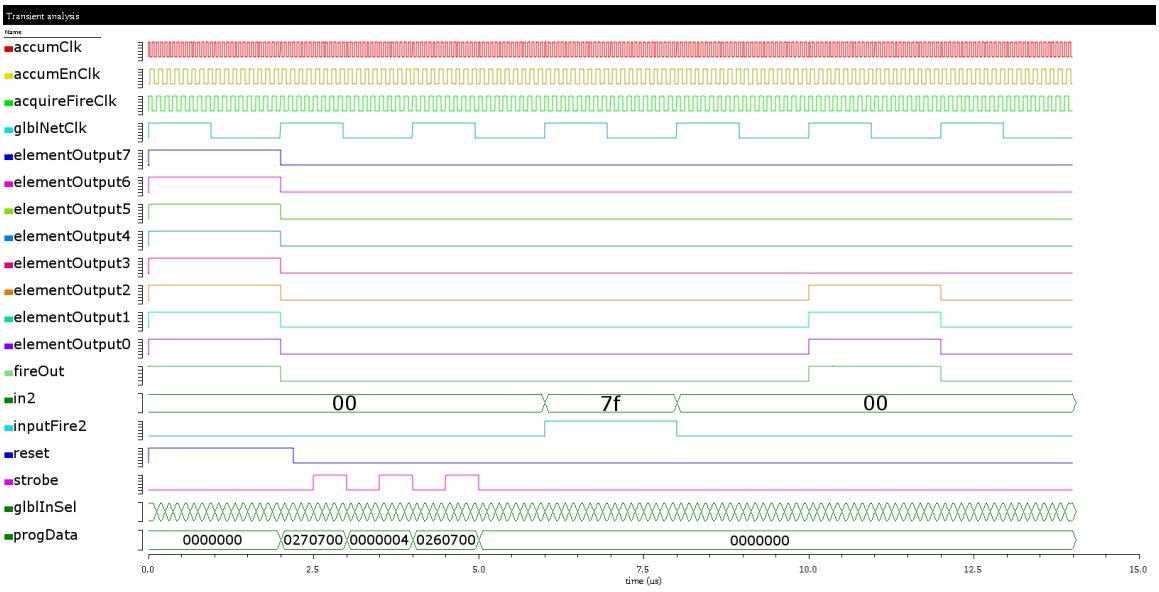


Figure 7.26: Combined Element - Synapse, Zero Distance Test

change in the internal weight, the synapse is made to fire again by driving another input into input two. When the element fires again, it has a weight of six instead of seven, showing that it depressed correctly.

Next, a test was run on the element to test the Long Term Potentiation feature of the synapse. This test is shown in Figure 7.28 and is very similar to the previous test. The only difference between this test and the other test is the timing of the fire sent into input three when the synapse is firing. The timing of the input fire is a specific amount of time after the synapse begins firing and is based on the value of the Global Input Select. Because of the time that input three begins firing, the fire could only have been caused by the synapse so it potentiates. This is shown by the output of the synapse, which is seven during the first fire and eight during the second fire.

The next test run on the element tests the situation where the element is programmed as a neuron and an input is received during the refractory period of the neuron. The waveform for this test is shown in Figure 7.29. As in the previous neuron tests, the threshold is set at three. Four microseconds into the test, inputs

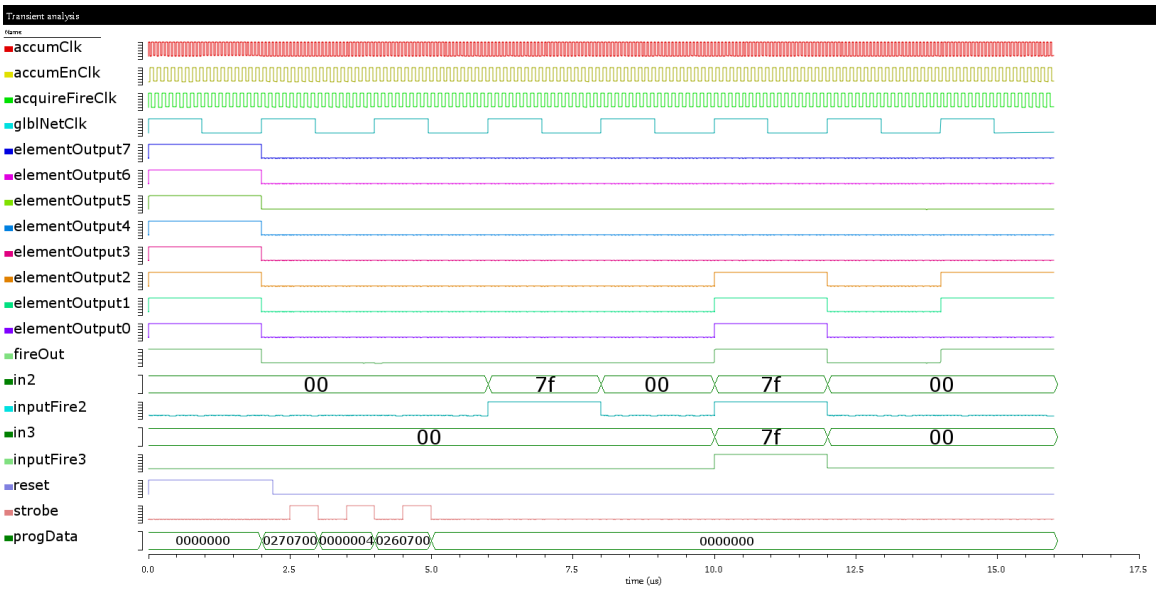


Figure 7.27: Combined Element - Synapse Depression Test

with a weight of seven are driven into inputs two and twelve. Because the Global Input Select in this simulation is set to start a zero at each Global Network Clock cycle and count up, input two is sampled first and causes the element to begin firing. Later in the same Global Network Clock cycle, input 12 is sampled and added into the accumulator, but the element is already firing and in its refractory period so it does not affect the firing of the element. Later, $8\mu s$ into the simulation, another input with a weight of one is received on input two. This input causes the neuron to fire and shows that the earlier weight of seven received during the refractory period was added into the accumulator because a weight of one would not cause the neuron with a threshold of three to fire on its own.

In the next test run on the element, all of the inputs are the same as those of the first test that programmed the element as a neuron and a single input caused it to fire, except that the address driven into the element is incorrect. This tests the functionality of the programming interface and registers of the element and is shown in Figure 7.30. As the figure shows, when the input is received four microseconds

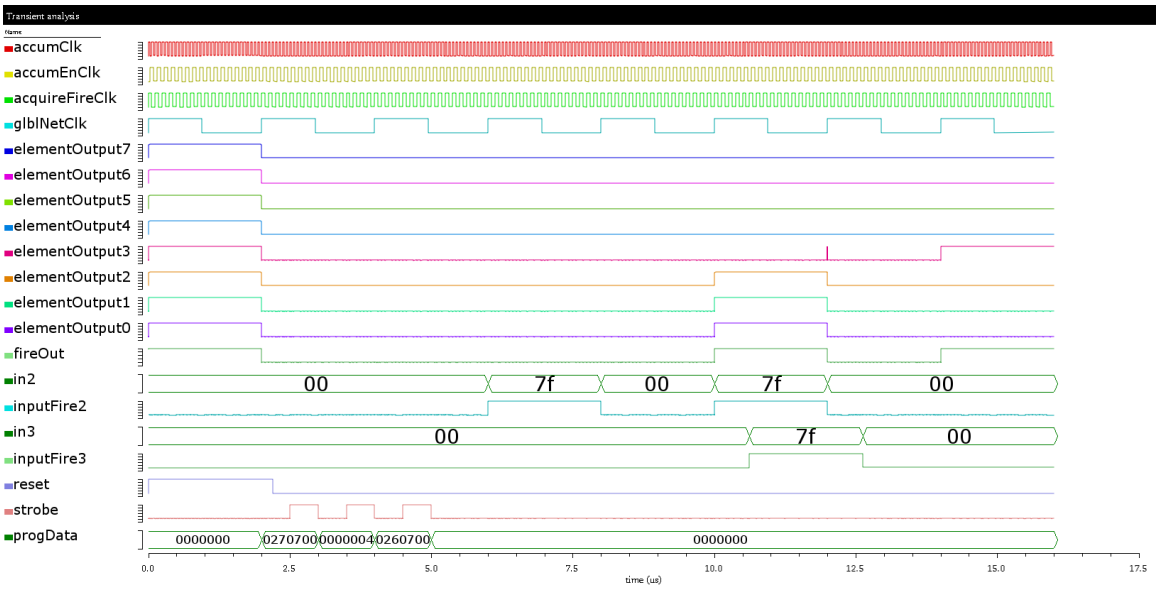


Figure 7.28: Combined Element - Synapse Potentiation Test

into the simulation, it does not cause an output fire as it would if the element was programmed.

Next, the functionality of the element when programmed as a neuron with multiple inputs enabled is tested and is shown in Figure 7.31. When inputs with a weight of two are received on inputs two and three, they cause the element to fire. This shows that

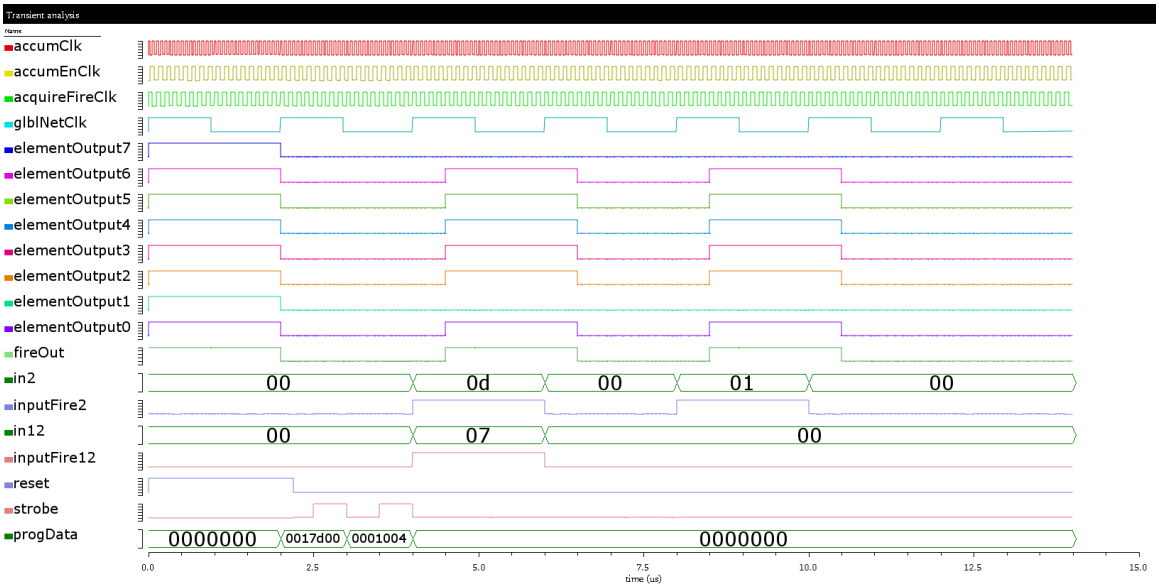


Figure 7.29: Combined Element - Neuron Refractory Period Test

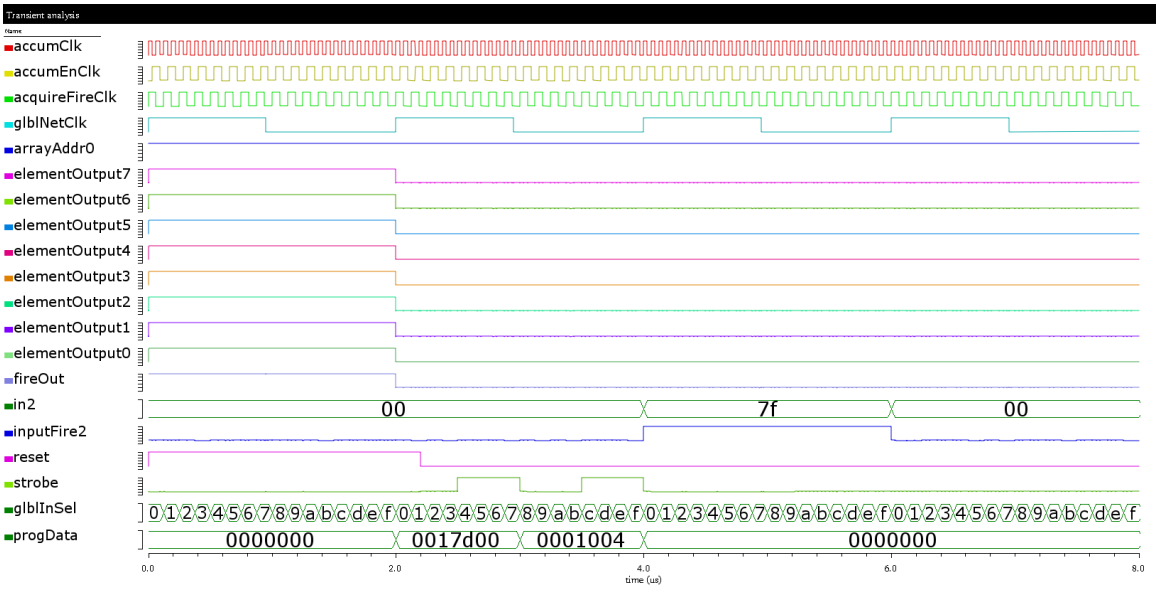


Figure 7.30: Combined Element - Wrong Programming Address Test

the values of all enabled inputs were added into the accumulator because the threshold is three and each of the inputs could not cause the element to fire individually but together they do.

The final test run on the full element tests the functionality of the monitoring feature of the element and is shown in Figure 7.32. After the element is programmed as a synapse, a fire is sent into input two and the synapse distance is very large so it is stored in the synapse distance register but does not come out during the simulation time. Then, capture is pulsed and the data is sent out serially. After reviewing the data on `serialOutput`, it was found to be correct. Note that, while this testing does verify the features of the element, additional exhaustive testing is necessary before implementing the design in silicon.

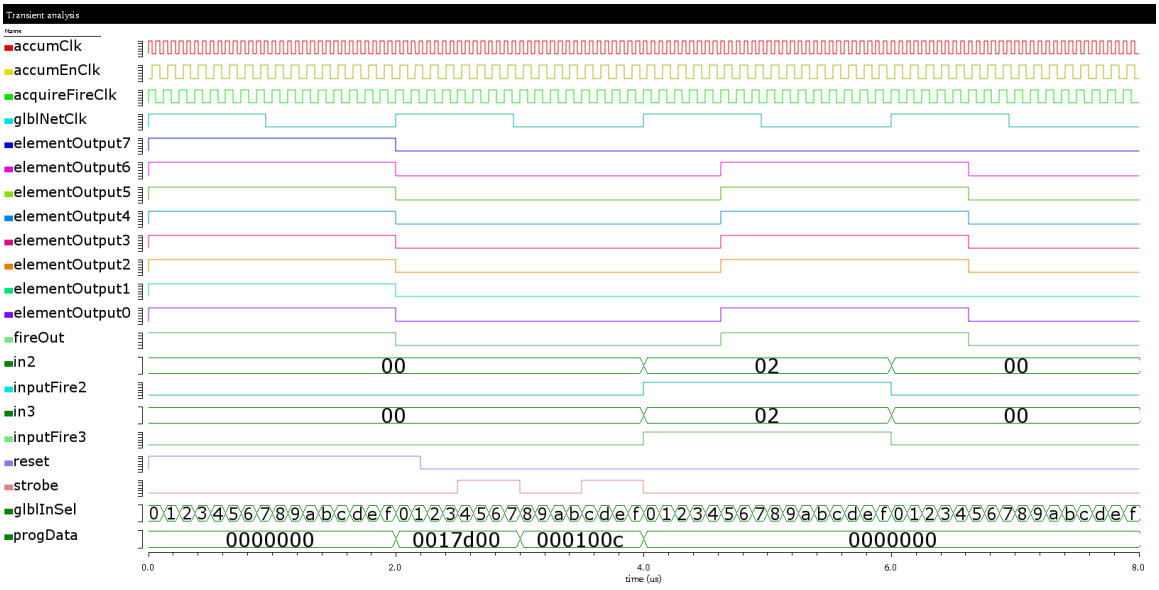


Figure 7.31: Combined Element - Neuron with Multiple Inputs Test

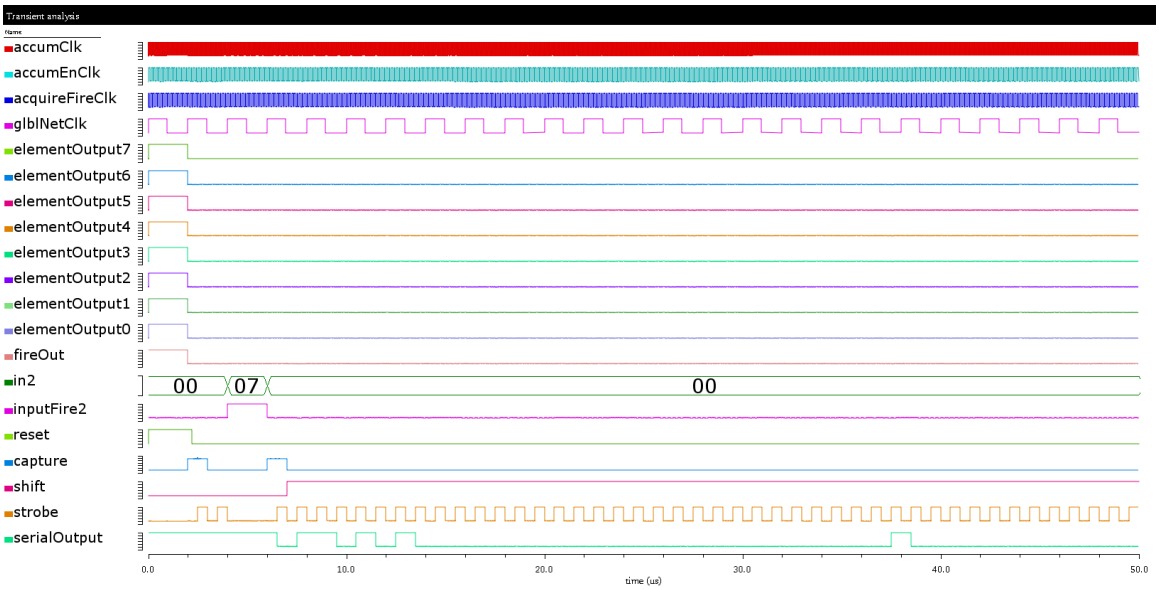


Figure 7.32: Combined Element - Monitoring Test

Chapter 8

VLSI Array Implementation

8.1 Preparation for Encounter

Before the layout of the array could be started, several preparation steps had to be completed. The first of those was to learn about Cadence Encounter. This software tool implements the placement and routing steps of the ASIC design flow. As its input, it takes a Verilog netlist describing how standard cells are connected and a library of standard cells. To improve the quality of the resulting design, Encounter also allows the user to import Liberty timing libraries, which define the power, timing, and delay characteristics of the standard cells, Synopsys Design Constraints (SDC) files, which define all of the constraints on the final design such as clocks, loads, and delays, and capacitance tables and technology files, which assist with the parasitic extraction of the final design. After these files are provided to Encounter, floorplanning is performed to define the design size and the spacing between cells. Next, power net connections are defined and the power rings are placed. Then, placement is run on the design. This step places all of the standard cells, taking into account their connectivity and timing. This step also runs Trial Route after placement. Trial Route is a basic routing routine that does a very poor job but provides a starting point for the other steps. Next, the fanout correction tool is run. This tool analyzes all of the

nets in the design and ensures that their drivers can drive all of the loads while still meeting the timing requirements of the design. If they cannot, buffers and inverters are added to reduce the load on the driver. After that, the Special Route tool is used to connect all of the power nets of the cells to the power rings. Finally, NanoRoute is used to perform the routing on the design. NanoRoute is a much more powerful routing tool than Trial Route as it takes into account all of the metal layers to avoid shorts and follows the Design Rules. It also takes timing into account to further optimize the design. After all of these steps have been performed, Encounter can also be used to perform checking on the design.

After learning about how to use Encounter, the next step was to attempt to import the element. As mentioned in the previous paragraph, Encounter is designed to take premade cells and place and route them. Normally, this means that small standard cells for basic gates and flops would be combined, but a different approach was taken for this design. Instead of a small cell, the element is the basic building block of the design, though inverters and buffers are also used for the connections between elements and the trees for global signals. The first attempt to import the element was to use an OpenAccess (OA) database. This database is a relatively new data format developed to make exchanging data between separate programs easier. It is the file format that both Virtuoso, the custom layout tool, and Encounter use internally to store the design. Upon importing the design using this method, Encounter was able to place all of the cells, but NanoRoute gave error messages about no vias begin available. After investigating the issue, the problem was found to be an incompatibility in the IBM CMOS-8RF technology library. To get around this problem, the alternate method of importing the element as a Library Exchange Format (LEF) file was pursued. LEF files are what OA is replacing, but they are a much more mature format that has been used for many years and they are much more stable. In order to create the LEF file, another Cadence program had to be learned about. This program is called the Abstract Generator and it reads in a layout and creates an abstract view. This view includes the outlines of all the metal layers

(called blockages), the shapes and layers of all the pins, and antenna information about the pins. After the abstract is created, the Abstract Generator is also used to write out the LEF file, including information about the technology library. In addition to the element, the created LEF file included information about all of the buffers and inverters present in the standard cell library. Finally, after the LEF was created, it was imported into Encounter and the placement and routing steps were run again. During NanoRoute, the same issue with the missing vias was encountered. Because the LEF file is self-contained and has all of the information about both the design and technology, it could be modified to include the appropriate vias without possibly messing up all of the other designs associated with the technology library. After some investigation, it was found that there are two ways to specify vias in a LEF file, `via` statements and `viagenerate` statements. The technology library used all `viagenerate` statements, but Encounter expected `via` statements. After converting the statements manually, NanoRoute began functioning properly.

The next step after successfully importing the element into Encounter was to create the necessary timing data for the cells. Because they had come from a standard cell library, all of the buffers and inverters already had a timing library associated with them, but because it was a custom design, the element did not. The Cadence tool that characterizes cells and creates a timing library for them is called Encounter Library Characterizer (ELC). As its input it takes a Spectre netlist describing the element and all of its parasitics and it then devises the appropriate tests and uses the Spectre Simulator to test and characterize the cell. The first needed piece of information about the element was the extracted Spectre netlist. For the IBM 8RF process, Mentor Graphics Calibre PEX is used to determine all of the parasitics. When the PEX netlist was run through the ELC flow, it did not behave as expected. The first step performed by ELC is to set up a database to contain the information about the cells and their test results. This database is simply a folder structure present in the working directory. Because some of the subcell names contained slash characters and they were not caught by the program, the names were interpreted as

additional directory structure and caused ELC to crash. After fixing the problem with the slashes, the design was run through ELC and the next problem was discovered. The output timing library of ELC contained many cells instead of a single one and there was not any timing data for the cells. After further investigation, it was found that the output of PEX is organized into many individual cells that are then wired together to create the overall design. Each of the cells is the RC network for a single net. Unfortunately, ELC expects the design to be in a single cell and characterizes each cell individually. Because each of the parasitic cells contains only resistors and capacitors and no logic and the logical cells contain no resistors or capacitors to introduce delay, no timing data was produced for any cells. To fix this problem, a python script was written that both changes any slash characters to legal characters and rewires the netlist to combine all of the individual cells into a single cell. After performing all of these steps, a Spectre netlist that was compatible with ELC was created and the characterization process was begun. After running for many hours, ELC crashed unexpectedly even though all of the input was valid and in the correct format. After investigation, it was found that ELC had used up all of the memory resources of the computer (RAM and swap) and had been killed by the operating system. Because the computer being used had enormous resources, 192 GB total RAM and swap of which ELC used 166 GB, it was unlikely that ELC would work properly for this design on any available computer. The most likely reason for ELC using so many resources was because it is designed to characterize small standard cell libraries that contain gates and flops but it was being run on a large design. Because ELC would not work, a limited library was created by hand. This library did not contain all of the timing data that the output of ELC would have, but it did contain the capacitance and fanout load of all of the input pins and the maximum capacitance and fanout load that all of the output pins could drive. Because all of the elements in the design are identical, the internal timing data for a single element is not as important as the loads.

The final preparation step that had to be completed before attempting an array layout was the creation of the supporting files. First, the Synopsys Design Constraints (SDC) file was written. This file contains definitions for all of the clocks, including their period, duty cycle, and source pin, the maximum load that the input pins can drive, and the load that the drivers of the output pins must be able to drive. After creating this file, the capacitance table and technology file were needed. These files are used in the RC extraction process and were provided as part of the 8RF design kit.

8.2 Array Layout

8.2.1 Layout Planning

Before work on creating the layout of the array could begin, several planning steps had to be completed. The first of those steps was to determine how much of the die space the array could occupy. To determine this value, the amount of space occupied by the padframe had to be determined. As shown in Figure 8.1, each pad is $101.33\mu m \times 184.825\mu m$. Because of the orientation of the pads, $184.825\mu m$ was subtracted from all four sides of the chip. The total size of the chip is $20.76mm \times 18.76mm$, so the total space remaining for the layout after subtracting the pad area is $20.39035mm \times 18.39035mm$. Of this space, all but a small strip at the bottom was allocated for the array. Based on this, the area for the array is $20mm \times 18mm$ and the remaining area for the supporting components is $20mm \times 375\mu m$. This floorplan is shown in Figure 8.2, though the diagram is not drawn to scale.

The final planning step was to decide where the pins should be placed. To give the placement and routing algorithms as much freedom as possible, the pins are only constrained to a certain edge of the array. Because the supporting components will be placed in a row below the array, all control pins were constrained to the bottom of the

array. Because all inputs are to elements in the first column of the array, two-thirds of those pins were constrained to the left edge of the array and one-third of those pins were constrained to the top of the array. The reason for placing one-third of the pins at the top of the array was to assist with the eventual wiring of the pins to the padframe. For the same reason and because all outputs are connected to elements on the rightmost column of the array, one-third of the outputs were constrained to the

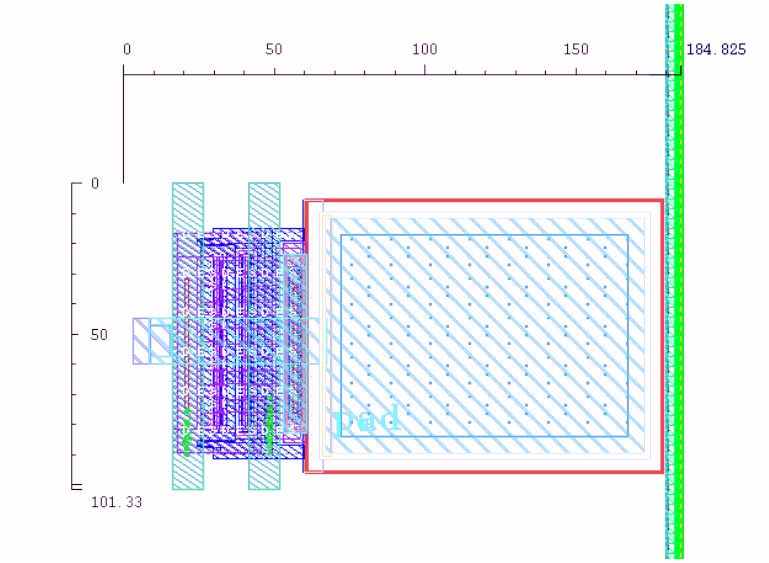


Figure 8.1: Single pad measurement

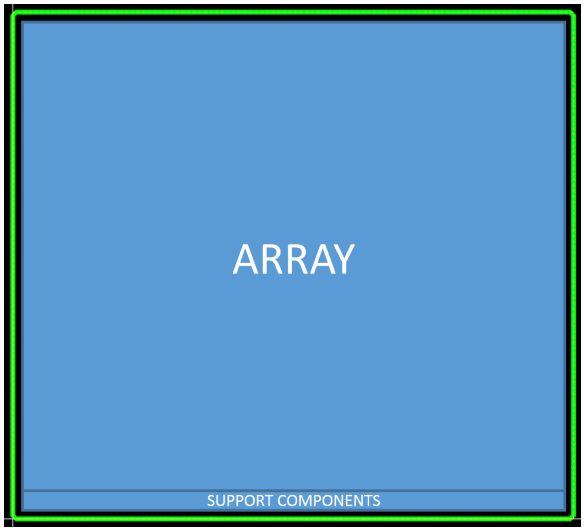


Figure 8.2: Floorplan for the full chip (Not to Scale)

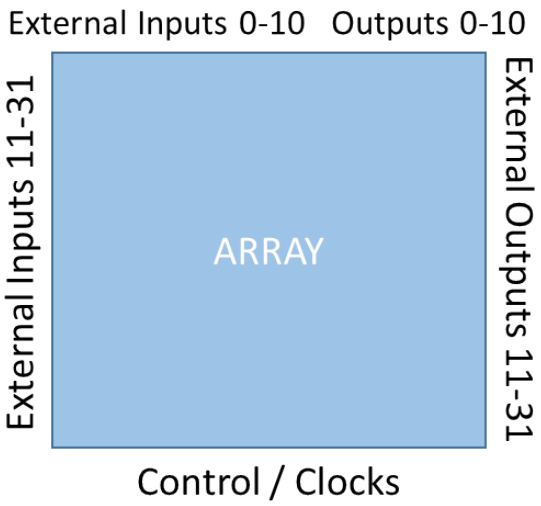


Figure 8.3: Pin placement guide for the array

top of the array and two-thirds of the outputs were constrained to the right side of the array. A diagram showing where all of the pins were constrained to is shown in Figure 8.3.

8.2.2 Initial Attempt

The initial attempt to create a full array in Encounter was on a 100×100 array. After placement of the elements in the array, the density of the design was over 95% without any of the additional buffers and inverters required for fanout violation correction and for creating the clock trees. Next, the process for identifying and fixing max fanout violations was run. Though the tool identified over 10,000 max fanout violations, no buffers were inserted due to the lack of available space. Next, even though fanout correction had failed, clock tree synthesis (CTS) was run to see how it would react to the large array. Soon after starting the process, CTS failed with a segmentation fault. Further investigation showed that this was likely related to the number of elements present in the array instead of the lack of available space for the buffers. This conclusion was reached by placing and running CTS on several smaller sizes of arrays in the same space allowed for the 100×100 array. Segmentation faults were

encountered on arrays as small as 25×25. Because of this, alternate strategies had to be developed to get around this issue.

8.2.3 Alternate Strategies

To get around the segmentation fault issues that were encountered when attempting to run clock tree synthesis on the 100×100 array, three strategies were developed. The first alternate strategy developed and the most preferred one was to create a single row of the array and run fanout correction and CTS on that single row. Then, that row would be replicated and connected to create the full array. This strategy gets around the segmentation fault issue by limiting the array size that CTS is run on to a small number of elements. Because CTS is run on the individual rows and they are then connected together at the top level, the resulting clock tree is still optimized to meet the timing of the design. The second strategy developed was to create the full array at once and use the fanout correction process for all of the signals including clocks. This would get around the segmentation fault issue because the CTS flow would never be run, but would have the drawback of the resulting clock tree not being based on the desired timing of the design. This may work due to the relatively slow clock speeds driven out to the array, but additional investigation and testing would be required to ensure that the design constraints are met. Finally, the third strategy that was developed is a hybrid of the first two strategies. As in the first strategy, a single row would be created, but the fanout correction process would be run on all nets except for the Global Network Clock, which would have CTS run on it. This strategy was mainly a contingency plan in case a segmentation fault was still encountered with the first strategy, with the reduced number of clock loads (1 clock instead of 4) hopefully eliminating the problem.

Before any of the strategies could be attempted, the maximum possible array size needed to be determined. To do this, a single row of various sized arrays was placed in the correct proportion of the total array space. An 80×80 array was chosen as

the starting point for this test. Because the placement is required to be in tracks, the available area for placing a single row cannot simply be the full width and $1/80^{th}$ of the available height, as that would create a single track and not actually allow components to be placed in all available space. Instead, the area was specified as being a single track height tall, which is $169\mu m$, and expanding the width so that the area equals $1/80^{th}$ of the total area. In the case of the single row of the 80×80 array, this resulted in the row area being $26628\mu m$ wide and $169\mu m$ tall. After the single row was placed in this area, the full fanout correction and CTS flows for all clocks were run on it. While there were not segmentation faults in the CTS flow, it did run out of area to place buffers and inverters. The placement that was created is shown in Figure 8.4. In the figure, the black Xs mark places where a valid placement for an instance was not able to be found so the instance was placed where it overlaps with another instance. Because of these overlaps, the 80×80 array was determined to be too large.

Next, a 75×75 array was tested to see if it would fit in the allowable area. For this array, $1/75^{th}$ of the total area equated to an area that is $28402\mu m$ wide and $169\mu m$ tall. The results of the placement of the array in this area after running the fanout correction and CTS flows is shown in Figure 8.5. Unlike the placement for the single row of the 80×80 array, no white Xs are present in the design. This indicates that valid placements were found for all necessary instances. Because of this, all future array layout attempts were for 75×75 arrays.



Figure 8.4: Placement for a single row of an 80×80 array



Figure 8.5: Placement for a single row of a 75×75 array

8.2.4 Strategy 2

Due to the simplicity of the strategy, the second strategy option was run first. The elements of the design were first placed using a netlist for a full 75×75 array. After all of these elements were placed, the fanout flow was run on all of the signals in the array including the clocking signals. The resulting placement is shown in Figure 8.6. This placement has an overall density of 53.84% and contains 172,977 instances. The number of instances is especially impressive considering that there are 5,625 elements in the design so the remaining 167,352 instances are buffers and inverters. This placement has several desirable features. First, it is very compact. This seems to be beneficial as it would allow for an increase in the size of the array. After running the placement and fanout correction routines, all of the signals in the array were routed using NanoRoute. The results of this process are shown in Figure 8.7.

After the routing design flow was run, the Design Rule Checking (DRC) violations were viewed and are shown in Figure 8.8. All of the white Xs in the figure represent a single DRC violation. As the figure shows, the number of DRC violations is much

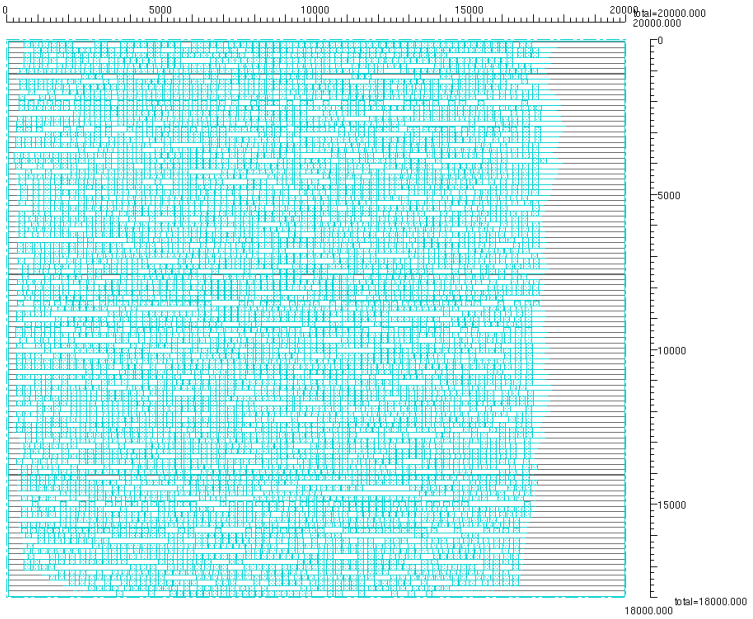


Figure 8.6: Placement for a 75×75 array using Strategy 2

too high. The routing procedure created 480,074 DRC violations of which 198,509 are shorts. Because of the shorts, there is no possibility of this design functioning correctly and 480,074 DRC violations are much too many to fix in a reasonable amount of time. There are likely several reasons for the high number of DRC violations. First, the NanoRoute tool was configured to attempt to identify and fix antenna violations, which further complicated the routing procedure. Second, the placement identified as very compact earlier in this section also increased the difficulty of successfully routing the design. Additionally, many of the buffers that were inserted to fix the fanout violations of the nets and to create the trees for the global signals are very far away from all of their connected cells. This is likely because of the way that the design was created, with all of the elements placed first and then the buffers were added and the element placement was moved slightly. This meant that the easiest place for the tool to put the buffers was near the edges of the array where there was either empty space or the elements were not as dense. Because of the failure of this strategy, the first strategy was attempted next.

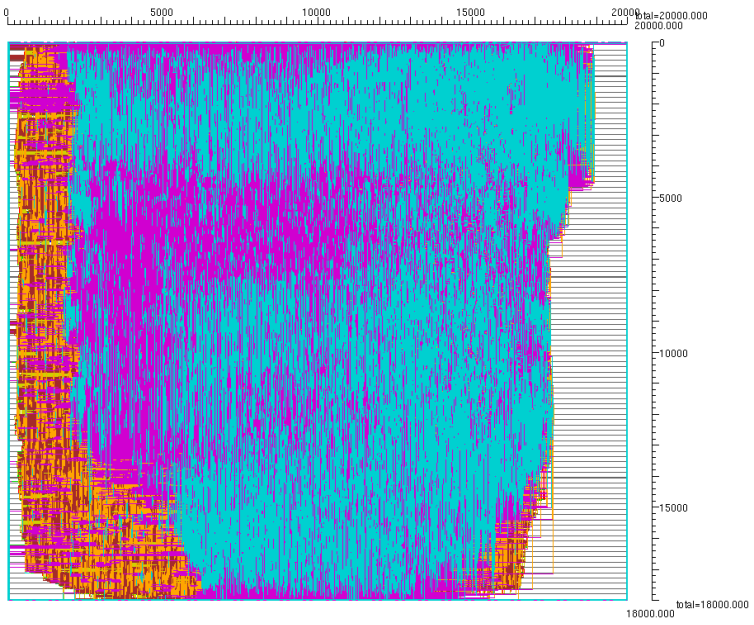


Figure 8.7: Routing of a 75 × 75 array using Strategy 2

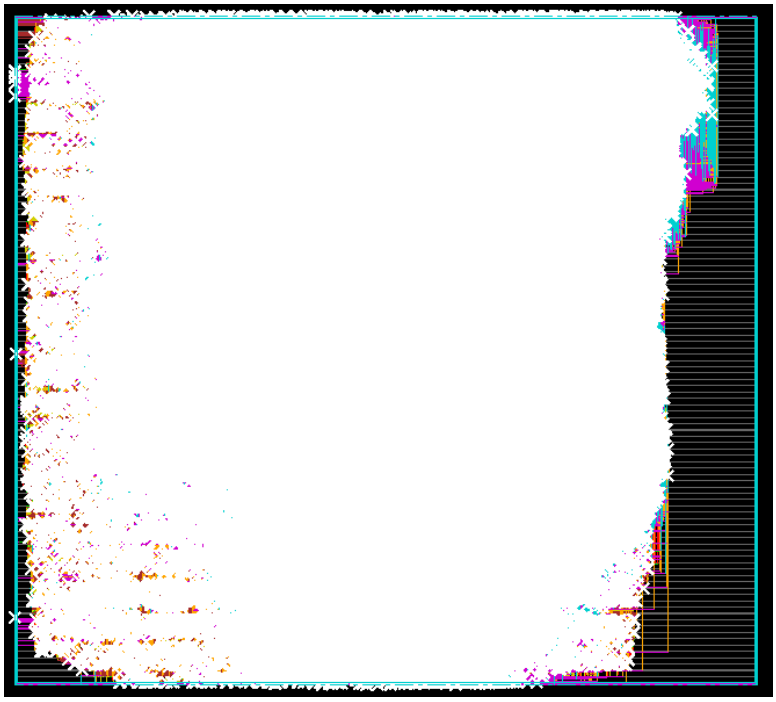


Figure 8.8: DRC violations of a 75 × 75 array using Strategy 2

8.2.5 Strategy 1

After attempting the second strategy in the alternate strategies list, the first and highest priority strategy was attempted. The original description of the strategy describes a single row of the array being run through the full fanout correction and CTS flow and then combined to create the fully array. This description does not take one important factor into account, the orientations of the elements. Because of this, the first four rows of the array were run through the process and then combined. Doing this covers all of the possible combinations of orientations. Another issue with running the rows through the fanout correction process individually is maintaining the correct fanout correction on the outputs of the elements that connect to other rows. Because of this, the SDC file was modified to add a load equivalent to 12 elements to all of the outputs pins of the elements going to other rows. After the SDC file was modified, the full flow was run on each of the four rows and then the Verilog netlist describing that row including all of the buffers and inverters was exported.

To connect all of the rows together, a python script was written to read through the netlist for each of the rows and produce a top level Verilog netlist tying the pins of the row modules together correctly. Also, because the addresses are hardcoded into each element, a separate script was written to change the address pins of the elements in a row to correctly reflect the addresses of the row they are in. Before placement could be run on the full array netlist including each of the rows, a clock tree needed to be created to connect the rows together. This clock tree would have the pin where the signal enters the array as its root and the top of each of the row clock trees as its loads. By analyzing the clock trees in the rows, the clock tree shown in Figure 8.9 was created. In the figure, the node at the top of the tree is the pin where the signal enters the array and the small nodes at the bottom of the tree connect to the individual row loads. This clock tree was incorporated into the top level netlist and is used for all of the global signals in the array including clocks, addresses, and programming signals.

Once all of the support tasks for this array were complete, the script was run and the top-level netlist was created. This netlist was then imported into Encounter and placement was run on it. The first time this process was run, the script incorrectly

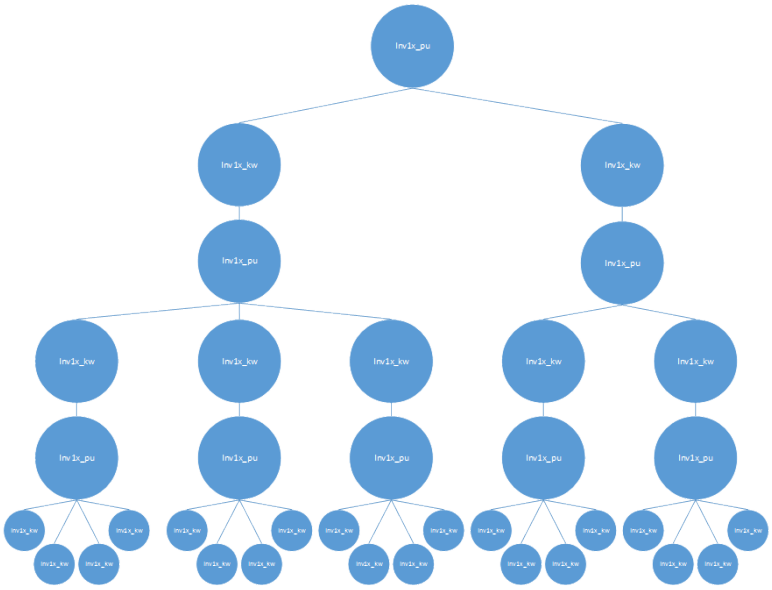


Figure 8.9: Manually created clock tree to drive the array rows

connected some of the global signals and the array shown in Figure 8.10 was created. After the script was fixed to correctly connect the global signals, the placement shown in Figure 8.11 was produced. This layout has a very unique shape, with the elements in the shape of the letter N and the global signal buffers placed in the gaps, but was actually found to be very logical after being analyzed. The details of this analysis can be found in Section 10.1. Also, this layout is much more spread out than the layout that was produced in Section 8.2.4 where all of the elements and buffers were bunched up against the left side of the chip area. The likely reason for the different shape and the buffers and elements being much more mixed together is because they were all placed at the same time. In the layout produced by Strategy 2, the elements were first placed and then the buffers were added, but in the layout produced by this strategy, all of the elements and buffers were already present in the netlist and were known from the beginning of the placement process so they were all placed at the same time.

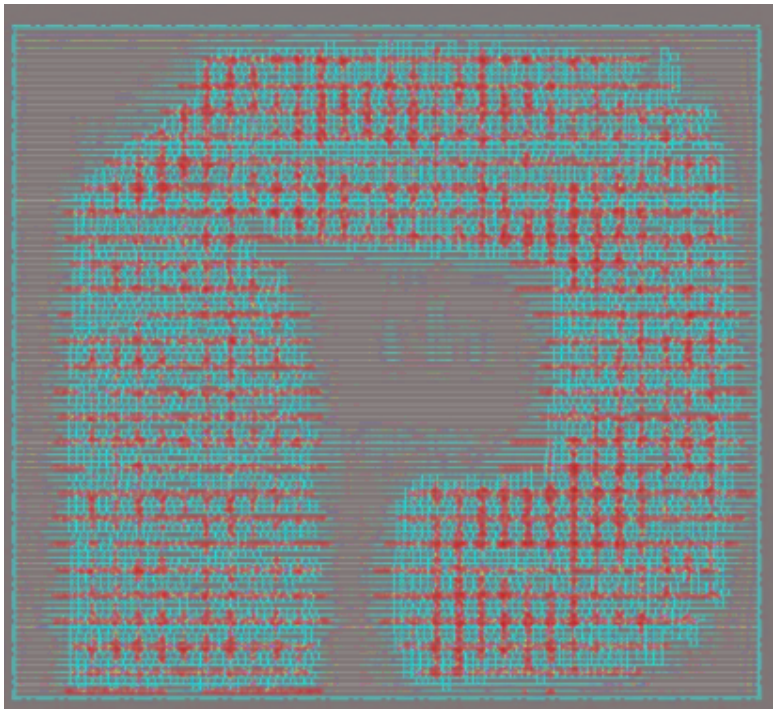


Figure 8.10: Placement of a 75×75 array using Strategy 1 with missing global signals

After placement was completed, routing was run on the array. Because all of the buffers for both fanout correction and the clock trees were already present in the netlist, the fanout correction and CTS flows were not repeated on this array. The results of the NanoRoute process are shown in Figure 8.12. As the figure shows, there was only one DRC violation present in this design. It was a minimum spacing violation that did not affect functionality and is easily fixable.

The overall density of this layout is 53.75% and contains 177,183 instances. Upon investigation of the layout, an issue was found that explains the low density numbers being achieved and is shown in Figure 8.13. The figure shows two elements with two buffers placed between them. Even though the buffers are much shorter than the elements, they block all of the space above them in the track from having additional buffers placed there. They block the space because of the way that Encounter placement works. Encounter is designed to work with cells that are all the same height and, if multiple height cells are present, the track height is defined by the height of the tallest cell. Then, cells can be placed beside each other but not stacked

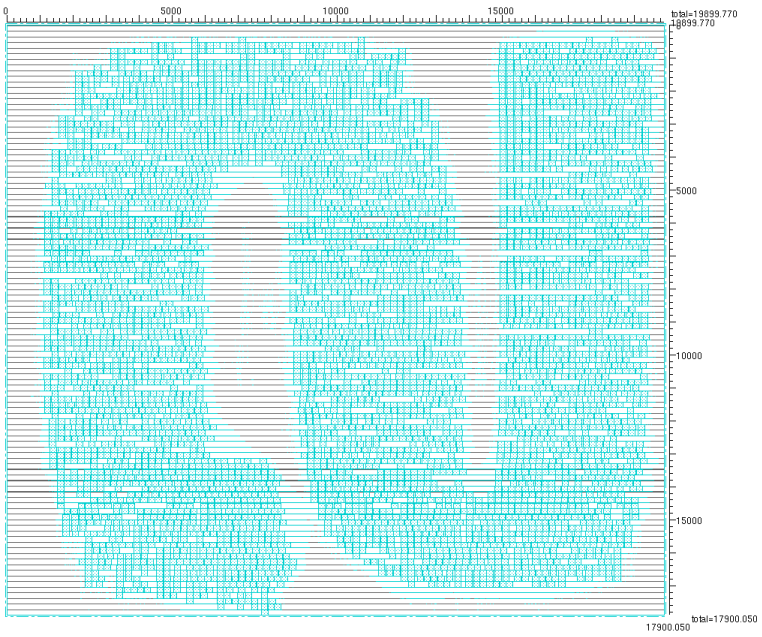


Figure 8.11: Placement of a 75×75 array using Strategy 1

vertically in a single track. There are several methods that could be employed to fix this issue. First, the element cell could be defined as a black box with the placement being independent of the tracks. Then, the tracks would be the height of the buffers and the space would not be wasted. Alternately, if multiple track heights were allowed, some of the tracks could be the height of the element and others could be the height of the buffers and much of the wasted space would be eliminated.

Finally, achieving this density and array size at the 130 nm process allows for extrapolation to other densities and process sizes. This extrapolation assumes quadratic scaling as the process size decreases and is shown in Table 8.1. The increase in density shown in the first column would be achieved using the methods described in the previous paragraph.

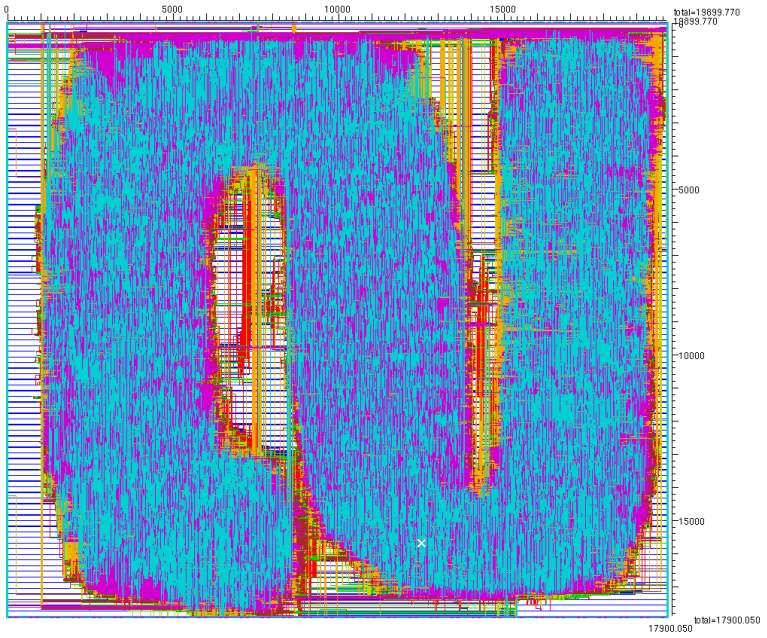


Figure 8.12: Routing of a 75 × 75 array using Strategy 1

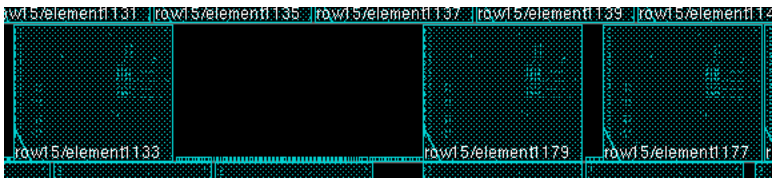


Figure 8.13: Wasted space by inverters and buffers

Table 8.1: Density/Process Extrapolation

Density	Process	Die Size	Elements	Array Size
53.75%	130 nm	390 mm^2	5,625	75×75
53.75%	28 nm	600 mm^2	186,543	432×432
53.75%	14 nm	600 mm^2	746,173	864×864
90%	130 nm	390 mm^2	9,419	97×97
90%	65 nm	390 mm^2	37,674	194×194
90%	28 nm	390 mm^2	203,028	450×450
90%	14 nm	390 mm^2	812,114	901×901
90%	12 nm	390 mm^2	1,156,656	1075×1075

Chapter 9

VLSI Supporting Component Implementation

The supporting components for the array include the clock generator, the FIFO logic, the random number generator, and the programming logic. The clock generator receives the four array clocks from the external clock generation chip and outputs modified versions of each of those clocks. The FIFO logic unloads the FIFO, puts the pieces received from the FIFO together, and sends them to the programming logic. It also receives data from the programming logic, splits it up into pieces, and loads those pieces into the FIFO. The random number generator uses a linear feedback shift register (LFSR) to create random numbers. Those numbers are then loaded as the starting point of a counter that provide the Global Input Select signal that is used by the elements to determine what input is sampled on a given cycle. Finally, the programming logic interprets the commands that are received through the FIFOs and sends the appropriate signals out to the array. It also monitors the output of selected elements in the array and receives the serial output from each column and packages them in a packet to be sent off chip. When creating these components, a very different design flow than the one used when making the elements and array was used. Like the element, the design flow begins with the FPGA VHDL code. Any

Xilinx IP blocks or FPGA primitives are then replaced with VHDL. Next, that code is sent through the Synopsys Design Compiler to create a Verilog netlist describing the connection of standard cells to create the component. That Verilog netlist is then analyzed to ensure that it is structured properly and that all VHDL functionality was able to be synthesized to standard cells. Finally, that netlist is imported into Cadence Encounter and the layout including placement and routing is created.

9.1 Clocking Element

The clocking element in the VLSI implementation serves a more limited purpose than it does in the FPGA implementation. In the FPGA implementation, it both generates the clocks and gates them, but in the VLSI implementation, the clocks are generated by an external chip, so the clocking element only gates the clocks. It receives all four of the array clocks directly from the external chip and it outputs two versions of each of those clocks to the array and other supporting components. The first version is an always on version of the clock that simply passes the input received from the external chip. The second version is a gated version that intelligently decides when to stop and start the clocks. Based on a signal received from the programming logic, the clocks are started and stopped at the boundary of Global Network Clock cycles so that no partial cycles are executed. Based on the VHDL code from Appendix A.8, the layout shown in Figure 9.1 was created. The layout is very small at $36.48\mu m \times 31.2\mu m$ not including the power rings.

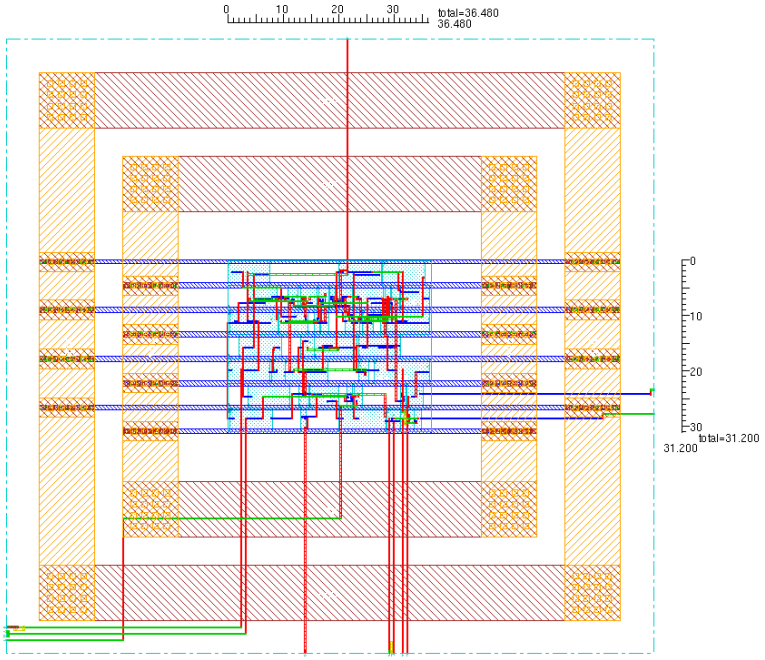


Figure 9.1: Clock Element layout from Cadence Encounter

9.2 FIFO Logic

The FIFO logic handles the data flow in both directions between the external FIFO chip and the programming logic. For data going from the FIFO to the programming logic, it unloads multiple four byte entries from the FIFO and then combines them to create the 36 byte packet that the programming logic expects. For the data going from the programming logic to the FIFO, this component splits the 64 byte packet into many four byte pieces and then loads them into the FIFO. Immediately before being implemented in the VLSI design, the FIFO logic underwent a major redesign that expanded the size of both the input and output packets and increased the internal clock speed to accommodate the increased packet size. The FIFO logic also required several minor changes to be compatible with the external FIFO chip that was chosen to interface with the chip. The primary change was that the read enable, write enable, full, and empty FIFO control signals all are active low in the VLSI implementation but were active high in the FPGA implementation. Using the VHDL code from Appendix



Figure 9.2: FIFO Logic layout from Cadence Encounter

A.10, the layout shown in Figure 9.2 was created. The layout is the largest of all the support components at $519.84\mu m \times 294.48\mu m$. The large size of the component is primarily due to the large number of flops that are required.

9.3 Random Number Generator

The random number generator component uses a linear feedback shift register (LFSR) to create a pseudo-random number which is used to initialize a counter. The output of this counter is then driven out to the array as the Global Input Select value that determines which input is sampled by each element on a given cycle. When implementing this component, the shift register had to be changed because, unlike in the FPGA, there is not a way to guarantee that the flops in the shift register power on in a set state. Instead, the shift register is implemented with a parallel load that is triggered by a reset signal. The pins of the parallel load are tied to constant values that initialize the shift register. After fixing the initialization problem with the shift register, the VHDL code from Appendix A.9 was used to produce the layout shown in Figure 9.3. Like the clocking element, this layout is relatively small at $170\mu m \times 170\mu m$. The layout has a very low density and could be much smaller, but more compact designs resulted in DRC violations in Encounter. Because much more

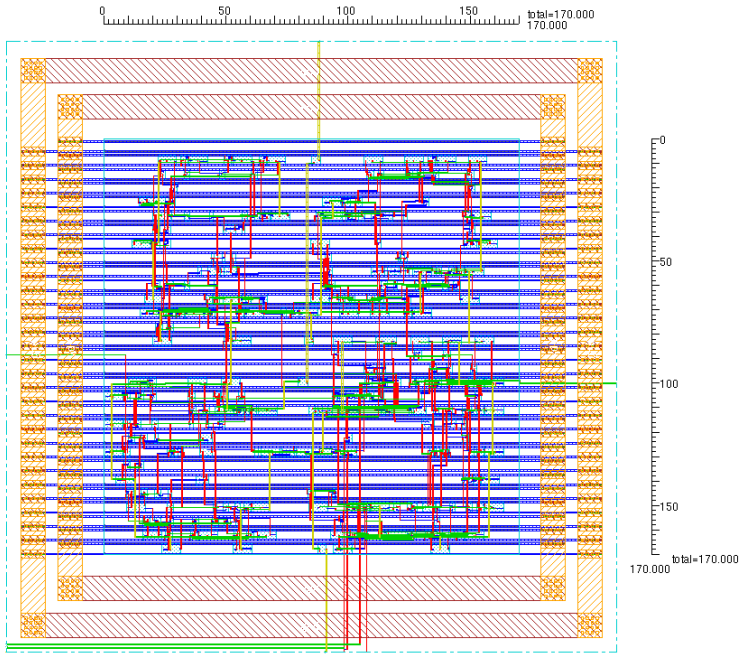


Figure 9.3: Random Number Generator layout from Cadence Encounter

space is available for the supporting components than they actually occupy, the larger size of this component was not an issue.

9.4 Programming Logic

The programming logic receives data from the FIFO logic, interprets that data, and then drives the appropriate signals out to the array. It also monitors the state of the output elements and receives the serial output signals for the monitoring of each column and then packages these values into a packet and sends it to the FIFO logic. Immediately before being implemented in the VLSI design, the programming logic underwent a major redesign. This design expanded the number of allowable external inputs from 16 to 32, which required an increase in the size of the input packet from 20 bytes to 36 bytes. It also expanded the allowable number of external outputs from 16 to 32. In addition to the expanded number of external inputs and outputs, it adds support for the monitoring functionality present in the array. It receives the serial

output signal from each column of the array and puts them into a packet to send off chip. Including the monitoring control functionality in the programming logic also required the addition of two commands to the existing command structure, capture and shift. When a capture command is received, the capture signal that is sent out to the array is raised and then the strobe signal is pulsed. When the shift command is received, the shift signal is raised and then the strobe signal is pulsed. Additionally, when the shift command is received, it triggers an output packet with the monitoring data that has been received from the array. The other commands that are available when interacting with the programming logic are run, halt, run N, fire, reset, and load. The run and halt commands start and stop the clocks going to the array immediately when received. The run N command tells the programming interface to run for a certain number of cycles before stopping. Another of the new features involves the run N command. Previously, the cycle counter for the array would be reset when a halt was implemented, but a change was made so the counter is only reset when a reset command is received. Because of this, an adder had to be implemented for the run N command so that the comparison value used to stop the counter would be equal to N cycles after the current count instead of the previous implementation which simply looked for when the counter equaled N. The fire command sends out weights to the array and was expanded to support 32 inputs. The reset signal resets all of the elements in the array and also resets the cycle counter. Finally, the load command was modified from its previous version. In the previous version, two load commands were required to program a single element and the values stored in the command payload changed each time. In the new version, a single load command triggers two commands to be sent out to the array. To support USB communication, a flag was added to output packets to indicate when a halt has occurred. A change was also made so that an output packet is triggered whenever a halt is implemented. An additional flag was added to the output packet to signify when the data in the monitoring output part of the packet is valid. Finally, an array ID was added at the end of each packet. This ID is used by the host machine to verify which array it is talking to and is hardcoded

into each array. For the programming interface implemented in the VLSI design, the array ID is 0x4344. When interpreted as ASCII, this array ID is "CD". Using the VHDL code from Appendix A.11, the layout shown in Figure 9.4 was created. This component is the second largest of the supporting components at $420\mu m \times 295\mu m$ but does not approach the allowed space for supporting components.

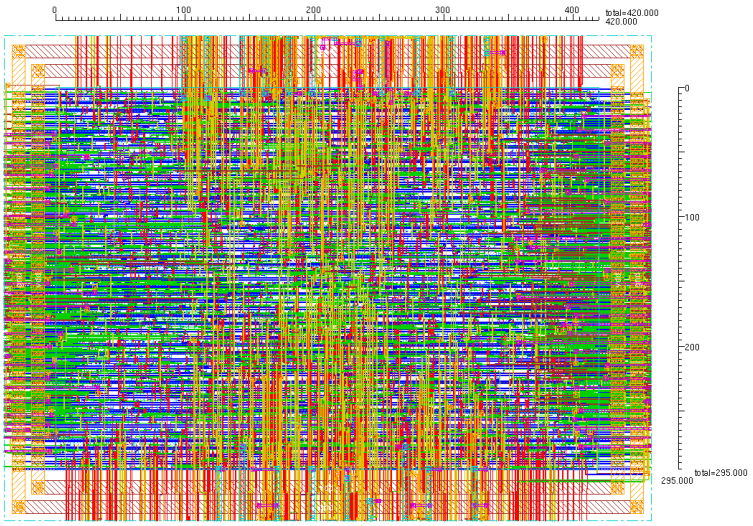


Figure 9.4: Programming Logic layout from Cadence Encounter

Chapter 10

Analysis

After completing the implementation of both the array and the supporting components, several analysis tasks were completed. These tasks were the analysis of the layout that was produced for the array, including if the placement of elements is logic, tracing through the clock tree, and determining buffer delays for multiple signals. Next, a model was created to implement the delay present through the element.

10.1 Layout Analysis

The first analysis that was performed as part of the layout analysis was to determine why the array was placed in the unique shape that it was and if that shape could be further improved. Figure 10.1 shows where some of the rows were placed within the array. The placement of rows starts at the bottom left corner of the shape and then follows the shape around until the last row is placed at the top right of the shape. The corner where row 50 is placed is tighter than the other corners, so row 50 and a few other nearby rows aren't as neatly placed as the others. This placement of the supporting components is very logical. All of the elements that are neighbors of the row are very close by, even closer than if the array was laid out in a standard square shape that took up the entire space. Also, this placement would be very conducive to creating the torus shape that has been discussed. Because the rows are placed

very compactly, the connections between the left and right side of the array would not have to be routed very far. The connection between the top and bottom of the array would be a little more complex, but the layout could possibly be modified to form the shape of a W instead of an N and would make this connection much easier.

After analyzing the placement of the elements and rows of the array, the placement and shape of the clock tree was analyzed. Figure 10.2 traces the Global Network Clock from the pin where it enters the array to the point where it enters each of the rows. As the figure shows, the clock enters the array at the bottom towards the right side and then is immediately routed to the middle of the array. The clock tree is not connected to itself at the point where it crosses itself immediately after entering the array. After being routed to the middle of the array, it branches out to the left and right sides and then further branches from there to reach all of the rows and elements of the array. After analyzing the shape of the clock tree, the layout shape of the elements makes even more sense. By laying the elements out in the shape of the letter N, the

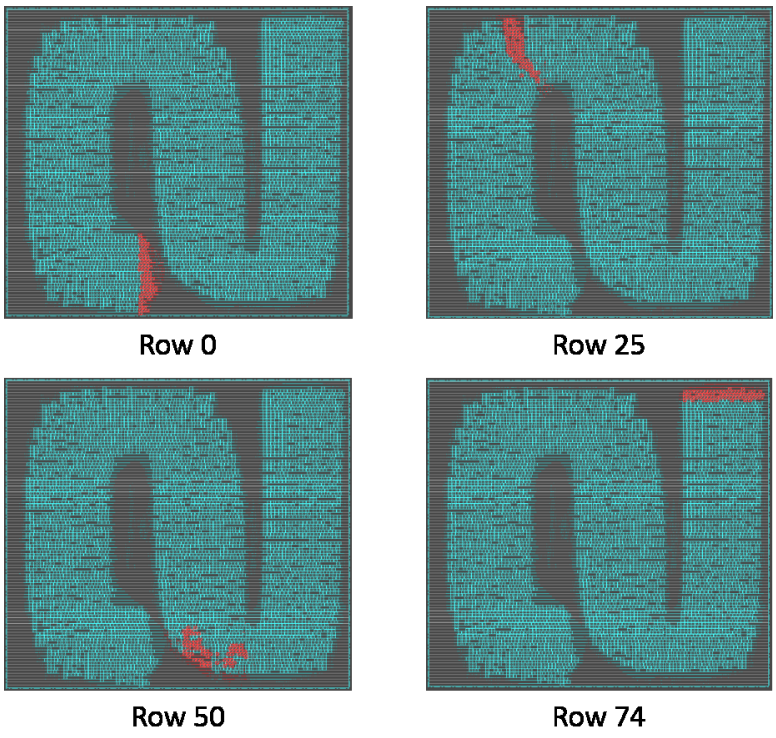


Figure 10.1: Position of several rows within the array, shown in red

spaces between the shape can be used to hold many buffers while keeping the buffers as close as possible to the loads being driven.

Finally, an analysis was performed to check the number of buffers between the root of a clock tree and several far away elements. This analysis was performed for several global signals including both clocks and normal global signals. Until the tree enters a single row, all of the clocks and global signals use the same layout for their tree. The layout of this tree was discussed in Section 8.2.5 and was shown in Figure 8.9. Based on this tree structure, each global signal passes through six inverters before entering the tree. After entering the tree, the number of buffers passed through varies slightly. For the `progData0` signal, which is the least significant bit of the programming data driven by the programming interface, the signal passes through a single buffer. For the clocks, each of the four row variations has a slightly different structure for the clock tree. In three of the four row variations, the clock signal passes through two inverters and one buffer between the top of the row tree and the loads. In the other

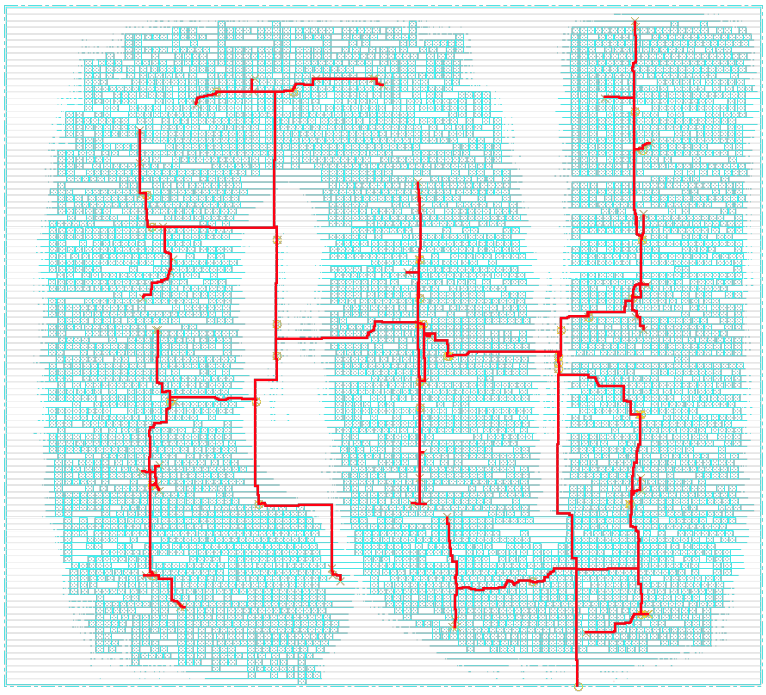


Figure 10.2: Global Network Clock tree from pin to each row

row variation, the clock passes through two inverters and two buffers between the top of the row tree and the loads. Due to the relatively slow clock speeds and the fact that the clock trees were designed using Clock Tree Synthesis that utilizes timing data, the slight difference in the number of buffers should not be a problem. Finally, the difference between the number of buffers for the programming interface signaling and the clocks should not be a problem because the programming interface operates separately from the clocks.

10.2 Delay Modeling

After completing the analysis of the layout that was created for the array, work began on creating a Verilog wrapper for the element HDL that implements the actual delays present in the layout of the element. The purpose of creating this wrapper is to work toward faster simulations of the entire array by replacing the actual elements with the wrapper. Depending on the compatibility of the simulator being used, the wrapper can be wrapped around either a VHDL or Verilog model of the element and will add the necessary delays to the outputs. The delays for the model were calculated by running simulations on the extracted version of the element with all parasitics present when driving the equivalent load of 4 NAND gates. One aspect of the element which made the creation of the delay model simpler was that all of the outputs of the element are triggered by a clock, so the only pin to pin delay that needed to be measured was from the clock to the output. When developing this delay model, Mentor Graphics ModelSim was used to simulate the Verilog. In Figure 10.3, a simulation is shown where a synapse fires. At the zoom level shown, the output looks like it all happens at the same time. Figure 10.4 shows a zoomed in version of the exact moment when the element begins firing and the varying delays of the pins are visible. Table 10.1 shows the delays that were measured and implemented in the delay model.

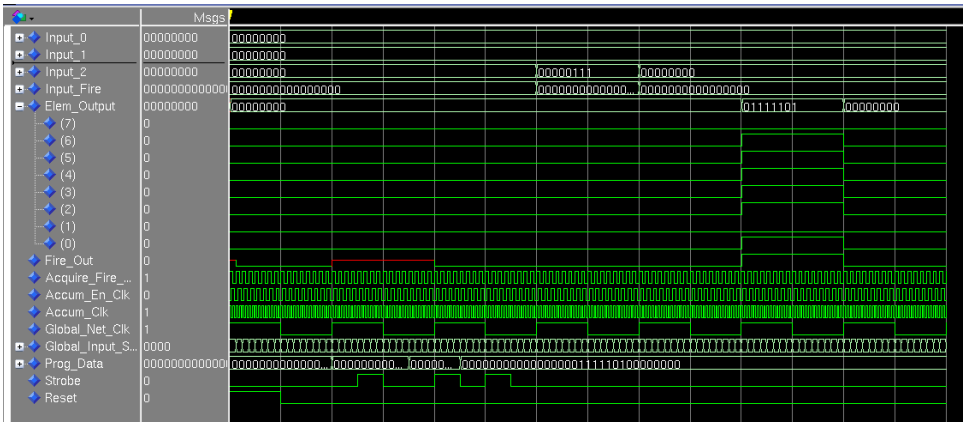


Figure 10.3: ModelSim simulation of the delay model

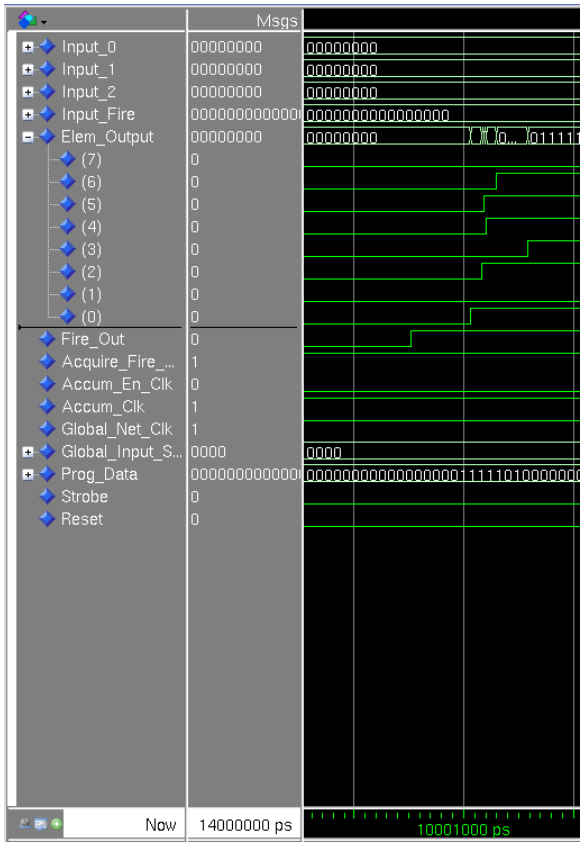


Figure 10.4: ModelSim simulation showing the difference in signal delays

Table 10.1: Element Output Delays

Signal	Rising Delay	Falling Delay
fireOut	0.976386 ns	1.226089 ns
elementOutput0	1.003336 ns	1.22333 ns
elementOutput1	0.999366 ns	1.219268 ns
elementOutput2	1.008296 ns	1.228938 ns
elementOutput3	1.029126 ns	1.253781 ns
elementOutput4	1.009946 ns	1.231169 ns
elementOutput5	1.008626 ns	1.22983 ns
elementOutput6	1.015236 ns	1.236674 ns
elementOutput7	1.029456 ns	1.251847 ns
serialOutput	0.5939928 ns	0.5111956 ns

Chapter 11

System Structure

All of the pieces discussed in this chapter are shown in the system diagram in Figure 11.1. The chip itself which is the focus of this thesis is marked as "CHIP" in the diagram, the external support chips discussed in Section 11.2 are labeled as "SiLabs Clock" and "FIFO Chip", the communication chips discussed in Section 11.3 are labeled as "FPGA or Microcontroller" and "FX3", and the external processes discussed in Section 11.4 are labeled as "External Process."

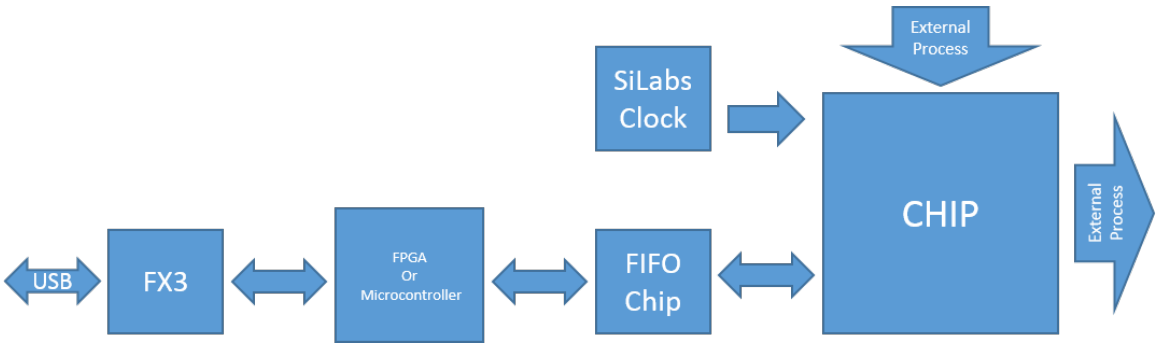


Figure 11.1: System diagram showing all components

11.1 Chip Floorplan

After the layout for all of the pieces of the chip was complete, they were combined into a single layout to create the chip floorplan shown in Figure 11.2. While the various layouts that make up the floorplan are not wired together, the floorplan does show how they all fit into the chip and their relative size compared to each other. Upon viewing the floorplan, it becomes very clear that the array dominates the area of the chip, as it should because it is doing the majority of the work on the chip. The floorplan also shows how tiny the supporting components are by comparison. Even though a strip across the bottom was left open for the supporting components, all four of them take up a very small total space. Therefore, if non-rectangular shapes were allowed, the array could be made even larger.

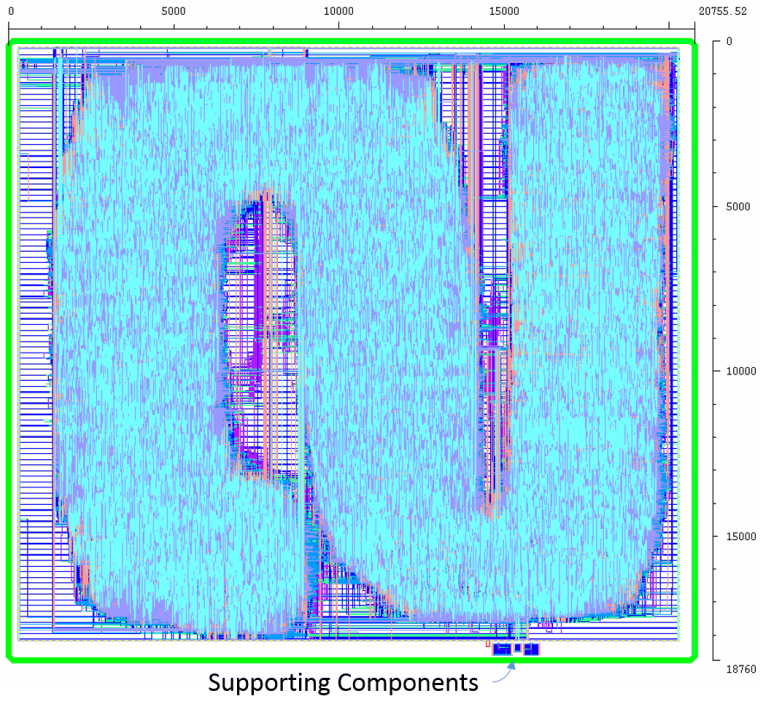


Figure 11.2: Full chip floorplan showing all components

11.2 External Support Chips

Several external support chips will be used in the full system that includes the VLSI chip. The first of those chips provides all of the clocks for the array. A Silicon Labs Si5334K-B04797-GM chip generates all of the necessary clocks, shown in Table 11.1, from a single 100 MHz differential oscillator input. This chip is a semi-custom chip in that it is generic but comes factory programmed to produce the needed clocks. One benefit of using an external chip to generate the clocks for the array instead of an internal generator is the ability to change the clock speeds later if desired. This can be accomplished by reprogramming the internal registers of the Silicon Labs chip.

The other external support chip used by the VLSI chip is the external FIFO chip. Two of these chips will be used to provide two-way off-chip communication. One side of each FIFO chip will interface with the VLSI chip and the other side will interface with the external communication chip discussed in the next section. For this chip, the Texas Instruments SN74V3670-7PEU chip has been chosen. This FIFO chip is 36 bits wide by 8192 entries deep. The chip is 36 bits wide instead of 32 bits wide because it leaves room for a parity bit. This bit could either be implemented or simply tied to ground. This specific chip was chosen because it is more than deep enough to support our current operation (the FPGA FIFOs are only 512 entries deep), it is relatively inexpensive, and it supports the desired clock speed. The maximum clock speed supported by the FIFO is 133.3 MHz but the VLSI chip will only use an 80 MHz clock to interface with it.

Table 11.1: Required Clocks

Name	Frequency	Phase Shift
Global Network Clock	5 MHz	0 deg
Acquire/Fire Clock	80 MHz	0 deg
Accumulator Enable Clock	80 MHz	90 deg
Accumulator Clock	80 MHz	0 deg

11.3 Communication Chips

There are several communication chips which would be present on the circuit board with the VLSI chip and the support chips. The first of these chips is a small FPGA or microcontroller. This chip interfaces with the opposite side of the FIFO chips than the VLSI chip and helps to implement the communication protocol. In the case of the system diagram shown in Figure 11.1, this small chip serves as the interface between the FIFO chips and the second communication chip, the FX3. This chip implements the USB communication logic for the design. One note about this design is that, while the current plan is to implement USB communication, the VLSI chip simply interfaces with a FIFO and then any communication protocol could be implemented to load and unload those FIFOs.

11.4 External Processes

The final aspect of the system diagram shown in Figure 11.1 is the external process communication. As part of the design of the VLSI chip, a subset of the external input and output elements are directly accessible through the pins of the chip. This allows the chip to interact directly with external processes instead of having to go through a host machine using whatever communication protocol is implemented.

Chapter 12

Future Work

There are many exciting possibilities for future work involving the hardware implementations of DANNA. As the next steps in the creation of the VLSI chip, the Verilog model could be perfected and it could be beneficial to create a Verilog-A model to accurately model the individual transistors in the element. In addition to the additional steps that would be beneficial for the Verilog model, the ability to simulate the clock trees individually is needed. This could be accomplished through the creation of a SKILL script that would start where the clock signal enters the array through the pin and would then trace through the layout to identify the clock tree. The script could then either copy the clock tree to another layout or could delete everything except the clock tree in the current layout. The script would also need to be able to identify the loads on the clock tree and replace them with appropriately sized dummy loads. To assist with testing, the random number generator shift register could be replaced with a parallel load shift register with the seed to be loaded provided through the programming interface.

In the near term future, a device with even greater density can likely be produced. Using the current method, the maximum array size is somewhere between 75×75 and 80×80, so a few dozen more elements could be added to the array with very little modification necessary. With the addition of multi-height tracks to

eliminate the buffer height issue, the maximum array size would likely approach 95×95. Another way to improve the number of elements in an array would be to reduce the size of the element. Several components inside the element could be further optimized. Examples of possible changes include replacing shift registers with counters, optimizing the logic equations of certain functions, replacing the decoder and gate in the LTP/LTD component with an XNOR comparator, and moving the reset for the monitoring component into the flops themselves. The density could be further improved by replacing the flops in the design with SRAM cells. Additionally, the connectivity of the design could be improved and network routing issues potentially resolved by adding connections to organize the array into a torus. This would allow for connections across edges and would reduce the maximum necessary routing distance within a network.

A change which would enable considerably larger arrays to be created would be access to improved processes. For example, with access to a 65 nm process, the number of elements could likely be nearly quadrupled, possibly approaching 38,000 elements. With access to further improved processes such as the 14 nm process currently available from MOSIS, the density could be improved even more significantly to as many as 812,000 elements. Another change which would greatly increase the density of the array would be to implement a full-custom layout. This would allow for higher densities because the constraints on element placement and density would be removed. In the longer term, a version of the DANNA chip could be made with incorporates analog devices to further improve the density of the elements. This would primarily be done by using memristors to replace the charge accumulation function of the element. Through the inclusion of memristors and other analog devices, the density of elements on the chip could be improved by five to ten times. Finally, a fourth element type in addition to the neuron, synapse, and pass-through could be included, the Central Pattern Generator (CPG). The CPG emulates the behavior of some biological processes by behaving like a neuron in the way it accumulates charge, but it sends out a train of multiple fires for some period of time when it fires instead

of a single pulse. The frequency of the pulses and the number of pulses sent out before stopping would be programmable parameters in the same way that the threshold and weight are in current elements. In addition to this, there are many other important possible improvements and revisions that can be made to the chip such as including the ability to connect multiple DANNA chips together to create a single array.

Chapter 13

Conclusion

In this thesis, I have designed and implemented a full DANNA array. This design includes a custom VLSI implementation of the DANNA element which includes many optimizations, was created to be as small and dense as possible, and was tested extensively to ensure that it functions correctly and meets the requirements for the design. The element design was then replicated to create a DANNA area using ASIC tools. The resulting array contains 5,625 elements in a 75×75 configuration and also includes all of the necessary components to support the distribution of internal signals including clocking trees and fanout trees. This array is a successor to the FPGA implementations which have been used for DANNA hardare up until now. This VLSI design improves upon the FPGA implementation in density and speed and includes improved array monitoring capabilities. The density of the array is improved by incorporating 5,625 elements on a chip compared to approximately 4,900 elements on the FPGA. Additionally, the VLSI design is in the IBM-8RF 130 nm process with a die size of $390mm^2$ and the FPGA design is in a 28 nm process with a die size of $600mm^2$. If scaling is taken into account, the VLSI design could theoretically contain 186,543 elements using the same process and die size as the FPGA. Additionally, the VLSI array is designed to support a Global Network Clock of 5 MHz, a ten times improvement over the FPGA design. In addition to the array itself, all of the

supporting components have been designed, the chip has been floorplanned, and a system including supporting chips has been specified. Overall, this thesis represents a significant step towards the goal of creating a DANNA chip.

Bibliography

[1] B.V. Benjamin, Peiran Gao, E. McQuinn, S. Choudhary, A.R. Chandrasekaran, J.M. Bussat, R. Alvarez-Icaza, J.V. Arthur, P.A Merolla, and K. Boahen. Neurogrid: A mixed-analog-digital multichip system for large-scale neural simulations. *Proceedings of the IEEE*, 102(5):699–716, May 2014. 4

[2] J. Douglas Birdwell, Mark E. Dean, Margaret Drouhard, and Catherine Schuman. Method and apparatus for constructing a neuroscience-inspired artificial neural network with visualization of neural pathways. U.S. Patent Application 14/513,497, filed October 14, 2014. 12

[3] J. Douglas Birdwell, Mark E. Dean, and Catherine Schuman. Method and apparatus for constructing a dynamic adaptive neural network array (DANNA). U.S. Patent Application 14/513,297, filed October 14, 2014. 8

[4] J. Douglas Birdwell, Mark E. Dean, and Catherine Schuman. Method and apparatus for constructing, using and reusing components and structures of an artificial neural network. U.S. Patent Application 14/513,388, filed October 14, 2014. 12

[5] J. Douglas Birdwell, Mark E. Dean, and Catherine Schuman. Method and apparatus for providing random selection and long-term potentiation and depression in an artificial network. U.S. Patent Application 14/513,334, filed October 14, 2014. 9

[6] J. Douglas Birdwell, Mark E. Dean, and Catherine Schuman. Method and apparatus for providing real-time monitoring of an artificial neural network. U.S. Patent Application 14/513,447, filed October 14, 2014. 23

[7] J. Douglas Birdwell and Catherine Schuman. Method and apparatus for constructing a neuroscience-inspired artificial neural network. U.S. Patent Application 14/513,280, filed October 14, 2014. 8

[8] Jason Chan. Danna element connection diagram. 10

[9] Mark E. Dean, Catherine D. Schuman, and J. Douglas Birdwell. Dynamic adaptive neural network array. In Oscar H. Ibarra, Lila Kari, and Steffen Kopechi, editors, *Unconventional Computation and Natural Computation*, Lecture Notes in Computer Science, pages 129–141. Springer International Publishing, 2014. 8, 35

[10] Ethan Farquhar, C. Gordon, and P. Hasler. A field programmable neural array. In *Circuits and Systems, 2006. ISCAS 2006. Proceedings. 2006 IEEE International Symposium on*, pages 4 pp.–4117, May 2006. 5

[11] S.B. Furber, D.R. Lester, L.A Plana, J.D. Garside, E. Painkras, S. Temple, and AD. Brown. Overview of the SpiNNaker system architecture. *Computers, IEEE Transactions on*, 62(12):2454–2467, Dec 2013. 3

[12] G. Indiveri, Elisabetta Chicca, and R. Douglas. A vlsi reconfigurable network of integrate-and-fire neurons with spike-based learning synapses. In *Proceedings of the 12th European Symposium on Artificial Neural Networks (ESANN)*, pages 405–410, 2004. 6

[13] Bryan L. Jackson, Bipin Rajendran, Gregory S. Corrado, Matthew Breitwisch, Geoffrey W. Burr, Roger Cheek, Kailash Gopalakrishnan, Simone Raoux, Charles T. Rettner, Alvaro Padilla, Alex G. Schrott, Rohit S. Shenoy, Bülent N. Kurdi, Chung H. Lam, and Dharmendra S. Modha. Nanoscale electronic synapses using phase change devices. *J. Emerg. Technol. Comput. Syst.*, 9(2):12:1–12:20, May 2013. 6

[14] M.M. Khan, D.R. Lester, L.A. Plana, A. Rast, X. Jin, E. Painkras, and S.B. Furber. Spinnaker: Mapping neural networks onto a massively-parallel chip multiprocessor. In *Neural Networks, 2008. IJCNN 2008. (IEEE World Congress on Computational Intelligence). IEEE International Joint Conference on*, pages 2849–2856, June 2008. 6

[15] B.J. Leiner, V.Q. Lorena, T.M. Cesar, and M.V. Lorenzo. Hardware architecture for fpga implementation of a neural network and its application in images processing. In *Electronics, Robotics and Automotive Mechanics Conference, 2008. CERMA '08*, pages 405–410, Sept 2008. 5

[16] P. Merolla, J. Arthur, F. Akopyan, N. Imam, R. Manohar, and D.S. Modha. A digital neurosynaptic core using embedded crossbar memory with 45pj per spike in 45nm. In *Custom Integrated Circuits Conference (CICC), 2011 IEEE*, pages 1–4, Sept 2011. 4, 6

[17] Paul A. Merolla, John V. Arthur, Rodrigo Alvarez-Icaza, Andrew S. Cassidy, Jun Sawada, Filipp Akopyan, Bryan L. Jackson, Nabil Imam, Chen Guo, Yutaka Nakamura, Bernard Brezzo, Ivan Vo, Steven K. Esser, Rathinakumar Appuswamy, Brian Taba, Arnon Amir, Myron D. Flickner, William P. Risk, Rajit Manohar, and Dharmendra S. Modha. A million spiking-neuron integrated circuit with a scalable communication network and interface. *Science*, 345(6197):668–673, 2014. 4

[18] Sebastian Millner, Andreas Grübl, Karlheinz Meier, Johannes Schemmel, and Marc olivier Schwartz. A vlsi implementation of the adaptive exponential integrate-and-fire neuron model. In J.D. Lafferty, C.K.I. Williams, J. Shawe-Taylor, R.S. Zemel, and A. Culotta, editors, *Advances in Neural Information Processing Systems 23*, pages 1642–1650. Curran Associates, Inc., 2010. 4

[19] Javier Navaridas, Mikel Luján, Jose Miguel-Alonso, Luis A. Plana, and Steve Furber. Understanding the interconnection network of spinnaker. In *Proceedings*

of the 23rd International Conference on Supercomputing, ICS '09, pages 286–295, New York, NY, USA, 2009. ACM. 6

[20] Thomas Pfeil, Andreas Grübl, Sebastian Jeltsch, Eric Müller, Paul Müller, Mihai A Petrovici, Michael Schmuker, Daniel Brüderle, Johannes Schemmel, and Karlheinz Meier. Six networks on a universal neuromorphic computing substrate. *Frontiers in neuroscience*, 7, 2013. 7

[21] B. Rajendran, Yong Liu, Jae sun Seo, K. Gopalakrishnan, L. Chang, D.J. Friedman, and M.B. Ritter. Specifications of nanoscale devices and circuits for neuromorphic computational systems. *Electron Devices, IEEE Transactions on*, 60(1):246–253, Jan 2013. 6

[22] E. Ros, E.M. Ortigosa, R. Carrillo, and M. Arnold. Real-time computing platform for spiking neurons (rt-spike). *Neural Networks, IEEE Transactions on*, 17(4):1050–1063, July 2006. 7

[23] J. Schemmel, J. Fieres, and K. Meier. Wafer-scale integration of analog neural networks. In *Neural Networks, 2008. IJCNN 2008. (IEEE World Congress on Computational Intelligence). IEEE International Joint Conference on*, pages 431–438, June 2008. 4

[24] J. Schemmel, A. Grubl, K. Meier, and E. Mueller. Implementing synaptic plasticity in a vlsi spiking neural network model. In *Neural Networks, 2006. IJCNN '06. International Joint Conference on*, pages 1–6, 2006. 4

[25] Catherine Dorothy Schuman. *Neuroscience-Inspired Dynamic Architectures*. PhD thesis, University of Tennessee at Knoxville, 5 2015. 8

[26] Thomas Sharp, Francesco Galluppi, Alexander Rast, and Steve Furber. Power-efficient simulation of detailed cortical microcircuits on spinnaker. *Journal of Neuroscience Methods*, 210(1):110 – 118, 2012. Special Issue on Computational Neuroscience. 5

[27] M. Soltiz, D. Kudithipudi, C. Merkel, G.S. Rose, and R.E. Pino. Memristor-based neural logic blocks for nonlinearly separable functions. *Computers, IEEE Transactions on*, 62(8):1597–1606, Aug 2013. 5

[28] Maria Stepanova, Feng Lin, and ValerieC.-L. Lin. A hopfield neural classifier and its fpga implementation for identification of symmetrically structured dna motifs. *The Journal of VLSI Signal Processing Systems for Signal, Image, and Video Technology*, 48(3):239–254, 2007. 7

[29] University of Wisconsin. University of Wisconsin VLSI Group. http://uwvlsi.wikispaces.com. 16, 31

[30] Xilinx. Xilinx 2000t parts promotional picture. http://www.s2cinc.com/resource-library/technical-articles/what-i-think-about-xilinx-virtex-7-2000t-vivado-design-kit. ix, 20

[31] Xilinx. *Large FPGA Methodology Guide: Including Stacked Silicon Interconnect (SSI) Technology*, 13.4 edition, January 2012. ix, 22

[32] C. Yakopcic, T.M. Taha, G. Subramanyam, and S. Rogers. Multiple memristor read and write circuit for neuromorphic applications. In *Neural Networks (IJCNN), The 2011 International Joint Conference on*, pages 2676–2682, July 2011. 5

Appendix

Appendix A

VHDL Code

A.1 Accumulator

```
----------------------------------------------------------------------------------
-- Company: UTK EECS
-- Engineer: Christopher Daffron
--
-- Create Date: 12/03/2014 02:38:15 PM
-- Design Name:
-- Module Name: accumulator - Behavioral
-- Project Name:
-- Target Devices:
-- Tool Versions:
-- Description:
--
-- Dependencies:
--
-- Revision:
-- Revision 0.01 - File Created
-- Additional Comments:
--
----------------------------------------------------------------------------------


library IEEE;
use IEEE.STD_LOGIC_1164.ALL;

-- Uncomment the following library declaration if using
-- arithmetic functions with Signed or Unsigned values
--use IEEE.NUMERIC_STD.ALL;

-- Uncomment the following library declaration if instantiating
-- any Xilinx leaf cells in this code.
--library UNISIM;
--use UNISIM.VComponents.all;

entity accumulator is
```

```
    Port (
        Accum_Clk           : in  std_logic;
        Accum_En_Clk        : in  std_logic;
        Dec_Weight          : in  std_logic;
        Element_Select      : in  std_logic;
        Fire                : in  std_logic;
        Fire_Complete       : in  std_logic;
        Inc_Weight          : in  std_logic;
        Neuron_Synapse      : in  std_logic;
        Reg_Mux_Fire        : in  std_logic;
        Reg_Mux_Out         : in  std_logic_vector (7 downto 0);
        Reset               : in  std_logic;
        Strobe              : in  std_logic;
        Threshold           : in  std_logic_vector (7 downto 0);
        Accum_Clr           : out std_logic;
        Reg_Accum_Sum       : out std_logic_vector (7 downto 0);
        Special_Reset       : out std_logic
    );
end accumulator;

architecture Behavioral of accumulator is

    signal Accum_Clr_sig : std_logic;
    signal Accum_Cout : std_logic_vector (7 downto 0);
    signal Accum_Reset_Underrun : std_logic;
    signal Accum_Reset_Overrun : std_logic;
    signal Accum_Sum : std_logic_vector (7 downto 0);
    signal Reg_Accum_Sum_sig : std_logic_vector (7 downto 0);
    signal Special_Reset_sig : std_logic;
    signal Accum_En : std_logic;

begin

    Accum_Clr <= Accum_Clr_sig;
    Reg_Accum_Sum <= Reg_Accum_Sum_sig;
    Special_Reset <= Special_Reset_sig;

--  Accumulator function of element...adjust charge for neuron or weight for synapse
--  The Accumulator implementation uses a "carry-chain adder".
--  This implementation minimizes the number of Slices required for the target function.
--  The performance should be sufficient to support the target operating frequencies.

Accumulator : process ( Accum_Clr_sig, Dec_Weight, Accum_Reset_Underrun, Accum_Reset_Overrun )
    begin
        if  (Accum_Clr_sig = '0'  and   Dec_Weight = '0')  then  --  Add registered output of
            input mux to accumulated value, if synapse...Reg_Mux_Out = 0 and inc_weight ctrls LTP

            Accum_Sum(0)  <=  (((Reg_Mux_Out(0) xor Reg_Accum_Sum_sig(0)) xor Inc_Weight) and not
                Accum_Reset_Underrun) or Accum_Reset_Overrun;
            Accum_Cout(0) <=  ((Reg_Mux_Out(0) and Reg_Accum_Sum_sig(0)) or (Inc_Weight and
                Reg_Accum_Sum_sig(0))) and not (Accum_Reset_Underrun or Accum_Reset_Overrun);

            Accum_Sum(1)  <=  (((Reg_Mux_Out(1) xor Reg_Accum_Sum_sig(1)) xor Accum_Cout(0)) and
                not Accum_Reset_Underrun) or Accum_Reset_Overrun;
            Accum_Cout(1) <=  (((Reg_Mux_Out(1) xor Reg_Accum_Sum_sig(1)) and Accum_Cout(0)) or (
                Reg_Mux_Out(1) and Reg_Accum_Sum_sig(1))) and not (Accum_Reset_Underrun or
                Accum_Reset_Overrun);
```

```
        Accum_Sum(2) <= (((Reg_Mux_Out(2) xor Reg_Accum_Sum_sig(2)) xor Accum_Cout(1)) and
            not Accum_Reset_Underrun) or Accum_Reset_Overrun;
        Accum_Cout(2) <= (((Reg_Mux_Out(2) xor Reg_Accum_Sum_sig(2)) and Accum_Cout(1)) or (
            Reg_Mux_Out(2) and Reg_Accum_Sum_sig(2))) and not (Accum_Reset_Underrun or
            Accum_Reset_Overrun);

        Accum_Sum(3) <= (((Reg_Mux_Out(3) xor Reg_Accum_Sum_sig(3)) xor Accum_Cout(2)) and
            not Accum_Reset_Underrun) or Accum_Reset_Overrun;
        Accum_Cout(3) <= (((Reg_Mux_Out(3) xor Reg_Accum_Sum_sig(3)) and Accum_Cout(2)) or (
            Reg_Mux_Out(3) and Reg_Accum_Sum_sig(3))) and not (Accum_Reset_Underrun or
            Accum_Reset_Overrun);

        Accum_Sum(4) <= (((Reg_Mux_Out(4) xor Reg_Accum_Sum_sig(4)) xor Accum_Cout(3)) and
            not Accum_Reset_Underrun) or Accum_Reset_Overrun;
        Accum_Cout(4) <= (((Reg_Mux_Out(4) xor Reg_Accum_Sum_sig(4)) and Accum_Cout(3)) or (
            Reg_Mux_Out(4) and Reg_Accum_Sum_sig(4))) and not (Accum_Reset_Underrun or
            Accum_Reset_Overrun);

        Accum_Sum(5) <= (((Reg_Mux_Out(5) xor Reg_Accum_Sum_sig(5)) xor Accum_Cout(4)) and
            not Accum_Reset_Underrun) or Accum_Reset_Overrun;
        Accum_Cout(5) <= (((Reg_Mux_Out(5) xor Reg_Accum_Sum_sig(5)) and Accum_Cout(4)) or (
            Reg_Mux_Out(5) and Reg_Accum_Sum_sig(5))) and not (Accum_Reset_Underrun or
            Accum_Reset_Overrun);

        Accum_Sum(6) <= (((Reg_Mux_Out(6) xor Reg_Accum_Sum_sig(6)) xor Accum_Cout(5)) and
            not Accum_Reset_Underrun) or Accum_Reset_Overrun;
        Accum_Cout(6) <= (((Reg_Mux_Out(6) xor Reg_Accum_Sum_sig(6)) and Accum_Cout(5)) or (
            Reg_Mux_Out(6) and Reg_Accum_Sum_sig(6))) and not (Accum_Reset_Underrun or
            Accum_Reset_Overrun);

        Accum_Sum(7) <= ((((((Reg_Mux_Out(7) xor Reg_Accum_Sum_sig(7)) xor Accum_Cout(6)) or
            (Reg_Mux_Out(7) and Reg_Accum_Sum_sig(7))) and (not Neuron_Synapse))
                        or (((Reg_Mux_Out(7) xor Reg_Accum_Sum_sig(7)) xor Accum_Cout(6)) and
                            (Neuron_Synapse))) and not Accum_Reset_Underrun) or
                           Accum_Reset_Overrun;

        Accum_Cout(7) <= ((((not Reg_Mux_Out(7)) and (not Reg_Accum_Sum_sig(7)) and Accum_Cout
            (6)) and (not Neuron_Synapse))
                         or ((((Reg_Mux_Out(7) xor Reg_Accum_Sum_sig(7)) and Accum_Cout(6))
                             or (Reg_Mux_Out(7) and Reg_Accum_Sum_sig(7))) and (
                             Neuron_Synapse))) and not (Accum_Reset_Underrun or
                             Accum_Reset_Overrun);

    elsif (Accum_Clr_sig = '0' and Dec_Weight = '1') then -- Add '-1' to Reg_Accum_Sum
        , LTD
        Accum_Sum(0) <= (not Reg_Accum_Sum_sig(0) and not Accum_Reset_Underrun) or
            Accum_Reset_Overrun;
        Accum_Cout(0) <= Reg_Accum_Sum_sig(0) and not (Accum_Reset_Underrun or
            Accum_Reset_Overrun);

        Accum_Sum(1) <= (((not Reg_Accum_Sum_sig(1)) xor Accum_Cout(0)) and not
            Accum_Reset_Underrun) or Accum_Reset_Overrun;
        Accum_Cout(1) <= (((not Reg_Accum_Sum_sig(1)) and Accum_Cout(0)) or Reg_Accum_Sum_sig
            (1)) and not (Accum_Reset_Underrun or Accum_Reset_Overrun);

        Accum_Sum(2) <= (((not Reg_Accum_Sum_sig(2)) xor Accum_Cout(1)) and not
            Accum_Reset_Underrun) or Accum_Reset_Overrun;
        Accum_Cout(2) <= (((not Reg_Accum_Sum_sig(2)) and Accum_Cout(1)) or Reg_Accum_Sum_sig
            (2)) and not (Accum_Reset_Underrun or Accum_Reset_Overrun);
```

```
        Accum_Sum(3) <= (((not Reg_Accum_Sum_sig(3)) xor Accum_Cout(2)) and not
            Accum_Reset_Underrun) or Accum_Reset_Overrun;
        Accum_Cout(3) <= (((not Reg_Accum_Sum_sig(3)) and Accum_Cout(2)) or Reg_Accum_Sum_sig
            (3)) and not (Accum_Reset_Underrun or Accum_Reset_Overrun);

        Accum_Sum(4) <= (((not Reg_Accum_Sum_sig(4)) xor Accum_Cout(3)) and not
            Accum_Reset_Underrun) or Accum_Reset_Overrun;
        Accum_Cout(4) <= (((not Reg_Accum_Sum_sig(4)) and Accum_Cout(3)) or Reg_Accum_Sum_sig
            (4)) and not (Accum_Reset_Underrun or Accum_Reset_Overrun);

        Accum_Sum(5) <= (((not Reg_Accum_Sum_sig(5)) xor Accum_Cout(4)) and not
            Accum_Reset_Underrun) or Accum_Reset_Overrun;
        Accum_Cout(5) <= (((not Reg_Accum_Sum_sig(5)) and Accum_Cout(4)) or Reg_Accum_Sum_sig
            (5)) and not (Accum_Reset_Underrun or Accum_Reset_Overrun);

        Accum_Sum(6) <= (((not Reg_Accum_Sum_sig(6)) xor Accum_Cout(5)) and not
            Accum_Reset_Underrun) or Accum_Reset_Overrun;
        Accum_Cout(6) <= (((not Reg_Accum_Sum_sig(6)) and Accum_Cout(5)) or Reg_Accum_Sum_sig
            (6)) and not (Accum_Reset_Underrun or Accum_Reset_Overrun);

        Accum_Sum(7) <= ((((not Reg_Accum_Sum_sig(7)) xor Accum_Cout(6)) or
            Reg_Accum_Sum_sig(7)) and not Accum_Reset_Underrun) or Accum_Reset_Overrun;
        Accum_Cout(7) <= '0'; -- if decrementing for LTD, this disables forcing of
            Reg_Accum_Sum to "7f"

    elsif (Accum_Clr_sig = '1') then

        Accum_Sum <= Threshold ; -- set accumulator to the programmed threshold value if
             neuron fired .... also represents weight for a synapse during initialization
        Accum_Cout <= "00000000" ;

    end if;
end process Accumulator;

Accumulate_Enable : process (Accum_Clr_sig, Accum_En_Clk)
begin
    Accum_En <= (Accum_En_Clk or Accum_Clr_sig) or Accum_Reset_Underrun or
        Accum_Reset_Overrun;
end process Accumulate_Enable;

registered_accumulator : process (Accum_Clk, Element_Select, Strobe, Reset)
begin
    if (rising_edge (Accum_Clk)) then
        if (Accum_En = '1') then -- load Accum Register with either new accumulated charge
            value or threshold (if neuron fired)
            -- Do this even though its in its refractory period
            Reg_Accum_Sum_sig <= Accum_Sum;
        else
            Reg_Accum_Sum_sig <= Reg_Accum_Sum_sig; -- hold Accum Register output if no
                new input for neuron or no LTP_LTD for Synapse
        end if;
    end if;
    -- If accumulator has reached maximum ('7f') and overflowed, than indicate fire condition
        has been reached (neuron only)
    -- Set to "1' on Reset so that threshold register is loaded into accumulator. used to
        load weight into accumulator when element is a synapse.
    if (Reset = '1' or (Strobe = '1' and Element_Select = '1')) then
        Special_Reset_sig <= '1';
```

```
            Accum_Clr_sig <= '1';
        elsif(rising_edge(Accum_Clk)) then
            Accum_Reset_Overrun <= Accum_Cout(7) and (not Accum_Sum(7)) and (not Reg_Mux_Out(7))
                and Reg_Accum_Sum_sig(7) and Accum_En_Clk;
            Accum_Reset_Underrun <= (not Accum_Cout(7)) and Accum_Sum(7) and Reg_Mux_Out(7) and (
                not Reg_Accum_Sum_sig(7)) and Accum_En_Clk; -- Go HI when underruns, go from 0 to
                255 DOWN

            Accum_Clr_sig <= ((Accum_Sum(7) and Accum_Cout(7) and Reg_Accum_Sum_sig(7)) or
                              ( Accum_Sum(7) and (not Reg_Mux_Out(7)) ) or
                              ( Accum_Cout(7) and (not Reg_Mux_Out(7))))
                             and reg_mux_fire and Neuron_Synapse and Accum_En_Clk
                             and not ( Fire and (not Fire_Complete)  );
            Special_Reset_sig <= '0';
        end if;
    end process registered_accumulator;


end Behavioral;
```

A.2 Input Multiplexer Control

```
----------------------------------------------------------------------------------
-- Company:
-- Engineer:
--
-- Create Date: 12/04/2014 10:38:15 AM
-- Design Name:
-- Module Name: input_mux_control - Behavioral
-- Project Name:
-- Target Devices:
-- Tool Versions:
-- Description:
--
-- Dependencies:
--
-- Revision:
-- Revision 0.01 - File Created
-- Additional Comments:
--
----------------------------------------------------------------------------------


library IEEE;
use IEEE.STD_LOGIC_1164.ALL;

-- Uncomment the following library declaration if using
-- arithmetic functions with Signed or Unsigned values
--use IEEE.NUMERIC_STD.ALL;

-- Uncomment the following library declaration if instantiating
-- any Xilinx leaf cells in this code.
--library UNISIM;
--use UNISIM.VComponents.all;

entity input_mux_control is
    Port (
```

```
        Global_Input_Sel        : in  std_logic_vector (3 downto 0);
        Input_Enable            : in  std_logic_vector (15 downto 0);
        Acquire_Fire_Clk        : in  std_logic;
        Neuron_Synapse          : in  std_logic;
        Mux_Out                 : in  std_logic_vector (7 downto 0);
        Mux_Fire                : in  std_logic;
        Reg_Mux_Out             : out std_logic_vector (7 downto 0);
        Reg_Mux_Fire            : out std_logic
    );
end input_mux_control;

architecture Behavioral of input_mux_control is

    signal Input_Mux_Clr : std_logic;
    signal Reg_Mux_Out_sig : std_logic_vector (7 downto 0);
    signal Reg_Mux_Fire_sig : std_logic;

begin

    Reg_Mux_Out <= Reg_Mux_Out_sig;
    Reg_Mux_Fire <= Reg_Mux_Fire_sig;

    with  Global_Input_Sel  select
    Input_Mux_Clr  <=   Input_Enable(0)  when "0000",
                        Input_Enable(1)  when "0001",
                        Input_Enable(2)  when "0010",
                        Input_Enable(3)  when "0011",
                        Input_Enable(4)  when "0100",
                        Input_Enable(5)  when "0101",
                        Input_Enable(6)  when "0110",
                        Input_Enable(7)  when "0111",
                        Input_Enable(8)  when "1000",
                        Input_Enable(9)  when "1001",
                        Input_Enable(10) when "1010",
                        Input_Enable(11) when "1011",
                        Input_Enable(12) when "1100",
                        Input_Enable(13) when "1101",
                        Input_Enable(14) when "1110",
                        Input_Enable(15) when "1111",
                        ('0') when others;

   register_input_sel :  process (Acquire_Fire_Clk)
    begin
        if (rising_edge (Acquire_Fire_Clk)) then
            if (Input_Mux_Clr = '0') then
                Reg_Mux_Out_sig  <=  "00000000";
                Reg_Mux_Fire_sig  <=  '0';
            elsif (Input_Mux_Clr = '1') then
                if (Neuron_Synapse = '0') then
                    Reg_Mux_Out_sig  <=  Reg_Mux_Out_sig;  --  Neuron_Synapse connected to CE
                        signal of Latch
                    Reg_Mux_Fire_sig  <=  Reg_Mux_Fire_sig;  --  Synapse : connect to CE signal of
                         latch
                elsif (Neuron_Synapse = '1') then
                    Reg_Mux_Out_sig  <=  Mux_Out;  --  selected input enabled for this element,
                        element is a neuron
                    Reg_Mux_Fire_sig  <=  Mux_Fire;      --  Neuron  : selected input enabled for
                        this element, element is a neuron
                end if;
```

```
            end if;
        end if;
    end process  register_input_sel;


end Behavioral;
```

A.3 LTP/LTD State Machine

```
----------------------------------------------------------------------------------
-- Company:
-- Engineer:
--
-- Create Date: 12/03/2014 02:38:15 PM
-- Design Name:
-- Module Name: ltp_ltd_mux_state_machine - Behavioral
-- Project Name:
-- Target Devices:
-- Tool Versions:
-- Description:
--
-- Dependencies:
--
-- Revision:
-- Revision 0.01 - File Created
-- Additional Comments:
--
----------------------------------------------------------------------------------


library IEEE;
use IEEE.STD_LOGIC_1164.ALL;

-- Uncomment the following library declaration if using
-- arithmetic functions with Signed or Unsigned values
--use IEEE.NUMERIC_STD.ALL;

-- Uncomment the following library declaration if instantiating
-- any Xilinx leaf cells in this code.
library UNISIM;
use UNISIM.VComponents.all;

entity ltp_ltd_mux_state_machine is
    Port (
        Acquire_Fire_Clk        : in  std_logic;
        Fire_Out                : in  std_logic;
        Global_Input_Sel        : in  std_logic_vector (3 downto 0);
        Global_Net_Clk          : in  std_logic;
        Neuron_Synapse          : in  std_logic;
        Output_Sel              : in  std_logic_vector (3 downto 0);
        Output_Port_Fire_Back   : in  std_logic;
        Special_Reset           : in  std_logic;
        Reg_Accum_Sum           : in  std_logic_vector (7 downto 0);
        Wt_Refrac_Per           : in  std_logic_vector (3 downto 0);
        Dec_Weight              : out std_logic;
        Inc_Weight              : out std_logic
```

```
    );
end ltp_ltd_mux_state_machine;

architecture Behavioral of ltp_ltd_mux_state_machine is

    signal Gate_Accum_En : std_logic;
    signal Output_Port_Selected : std_logic;
    signal Output_Port_Selected_1 : std_logic;
    signal Output_Port_Selected_2 : std_logic;
    signal Inc_Disable : std_logic;
    signal Dec_Disable : std_logic;
    signal Gate_Accum_En_Pulse : std_logic;
    signal Gate_Accum_En_Ctr_In : std_logic;
    signal End_Wt_Refrac_Per : std_logic;
    signal LTP_LTD_Detect : std_logic;
    signal Dec_Weight_sig : std_logic;
    signal Inc_Weight_sig : std_logic;

begin

    Dec_Weight <= Dec_Weight_sig;
    Inc_Weight <= Inc_Weight_sig;

 --  Control to determine when to sample output feedback.
    Sample_Port_Fire_Back : process (Global_Input_Sel, Output_Sel, Acquire_Fire_Clk,
        Inc_Weight_sig, Dec_Weight_sig, Gate_Accum_En, Reg_Accum_Sum, Fire_Out)
    begin
--  Must determine when to sample the "input signal" from the neuron connected/monitoring the
    output if the element is a synapse
--  Used to determine if "weight" should be potentiated or depressed
--  Use the following if implemented using "if else" construct to create Output_Port_Selected ...
    provides "true-table" for LUT6 construct listed above
        if (Global_Input_Sel = "0000" and Output_Sel = "0000") then Output_Port_Selected <= '1';
        elsif (Global_Input_Sel = "0001" and Output_Sel = "0001") then Output_Port_Selected <=
            '1';
        elsif (Global_Input_Sel = "0010" and Output_Sel = "0010") then Output_Port_Selected <=
            '1';
        elsif (Global_Input_Sel = "0011" and Output_Sel = "0011") then Output_Port_Selected <=
            '1';
        elsif (Global_Input_Sel = "0100" and Output_Sel = "0100") then Output_Port_Selected <=
            '1';
        elsif (Global_Input_Sel = "0101" and Output_Sel = "0101") then Output_Port_Selected <=
            '1';
        elsif (Global_Input_Sel = "0110" and Output_Sel = "0110") then Output_Port_Selected <=
            '1';
        elsif (Global_Input_Sel = "0111" and Output_Sel = "0111") then Output_Port_Selected <=
            '1';
        elsif (Global_Input_Sel = "1000" and Output_Sel = "1000") then Output_Port_Selected <=
            '1';
        elsif (Global_Input_Sel = "1001" and Output_Sel = "1001") then Output_Port_Selected <=
            '1';
        elsif (Global_Input_Sel = "1010" and Output_Sel = "1010") then Output_Port_Selected <=
            '1';
        elsif (Global_Input_Sel = "1011" and Output_Sel = "1011") then Output_Port_Selected <=
            '1';
        elsif (Global_Input_Sel = "1100" and Output_Sel = "1100") then Output_Port_Selected <=
            '1';
        elsif (Global_Input_Sel = "1101" and Output_Sel = "1101") then Output_Port_Selected <=
            '1';
```

```
        elsif (Global_Input_Sel = "1110" and Output_Sel = "1110") then Output_Port_Selected <=
            '1';
        elsif (Global_Input_Sel = "1111" and Output_Sel = "1111") then Output_Port_Selected <=
            '1';
        else Output_Port_Selected <= '0';
        end if;

        if (Reg_Accum_Sum = "01111111")     then    Inc_Disable <= '1' ; -- Indicates when
            synapse weight reaches maximum, avoid rolling weight over to -128
        else  Inc_Disable <= '0';
        end if;

        if (Reg_Accum_Sum = "10000000")     then    Dec_Disable <= '1' ; -- Indicates when
            synapse weight reaches minimum, -128, avoids rolling weight over to +127
        else  Dec_Disable <= '0';
        end if;

        if (rising_edge   (Acquire_Fire_Clk)) then
            Output_Port_Selected_1 <= Output_Port_Selected and Fire_Out;
            Output_Port_Selected_2 <= Output_Port_Selected_1;
            Inc_Weight_sig <= (Output_Port_Fire_Back and Output_Port_Selected_2 and (not
                Neuron_Synapse) and (not Inc_Disable) and (not Gate_Accum_En) and (not
                Dec_Weight_sig));
            Dec_Weight_sig <= (Output_Port_Fire_Back and Output_Port_Selected_1 and (not
                Neuron_Synapse) and (not Dec_Disable) and (not Gate_Accum_En));
        end if;
    end process Sample_Port_Fire_Back;

--  Accumulator Enable Function to support accumulation only on the accumulation phase of the
    cycle ... update charge for neuron or LTP/LTD for synapse
    Accumulator_Control   :   process (Inc_Weight_sig, Dec_Weight_sig, Gate_Accum_En,
        Gate_Accum_En_Pulse, End_Wt_Refrac_Per)
    begin
        Gate_Accum_En_Ctr_In <= (Gate_Accum_En and (not Gate_Accum_En_Pulse));
        LTP_LTD_Detect <= (Inc_Weight_sig or Dec_Weight_sig or (Gate_Accum_En and (not
            End_Wt_Refrac_Per)));
    end process Accumulator_Control;

-- SRL16E: 16-bit shift register LUT with clock enable operating on posedge of clock (Mapped to
    SliceM LUT6)
-- Virtex-7
-- Xilinx HDL Language Template, version 2013.3
--  Set the LTP/LTD refractory period to the global value and execute after synapse weight has
    been changed

    LTP_LTD_Refrac_Period : SRL16E
    generic map (
      INIT => X"0000")
    port map (
      Q => End_Wt_Refrac_Per,       -- SRL data output
      A0 => Wt_Refrac_Per(0),       -- Select[0] input
      A1 => Wt_Refrac_Per(1),       -- Select[1] input
      A2 => Wt_Refrac_Per(2),       -- Select[2] input
      A3 => Wt_Refrac_Per(3),       -- Select[3] input
      CE => '1',                    -- Clock enable input
      CLK => Global_Net_Clk,        -- Clock input
      D => Gate_Accum_En_Ctr_In     -- SRL data input
    );
```

```
   -- FDRE: Single Data Rate D Flip-Flop with Synchronous Reset and
   --       Clock Enable (posedge clk).
   --       Virtex-7
   -- Xilinx HDL Language Template, version 2013.3
   -- This flip-flop captures the condition that the synapse weight has been potentiated or
        depressed

    Disable_LTP_LTD : FDRE
    generic map (
      INIT => '0') -- Initial value of register ('0' or '1')
    port map (
      Q => Gate_Accum_En,        -- Data output
      C => Acquire_Fire_Clk,     -- Clock input
      CE => '1',                 -- Clock enable input
      --R => '0',
      R => Special_Reset,                  -- Synchronous reset input // changed from '0'
      D => LTP_LTD_Detect        -- Data input
    );

   -- This flip-flop captures the condition that the synapse weight has been potentiated or
        depressed
   -- it provides a pulse to the LTP/LTD refractory counter

    Disable_LTP_LTD_Pulse : FDRE
    generic map (
      INIT => '0') -- Initial value of register ('0' or '1')
    port map (
      Q => Gate_Accum_En_Pulse,       -- Data output
      C => Global_Net_Clk,       -- Clock input
      CE => '1',     -- Clock enable input
      --R => '0',
      R => Special_Reset,      -- Synchronous reset input // changed from '0'
      D => Gate_Accum_En         -- Data input
    );


end Behavioral;
```

A.4 Monitoring

```
----------------------------------------------------------------------------------
-- Company:
-- Engineer:
--
-- Create Date: 03/11/2015 05:15:14 PM
-- Design Name:
-- Module Name: monitoring - Behavioral
-- Project Name:
-- Target Devices:
-- Tool Versions:
-- Description:
--
-- Dependencies:
--
-- Revision:
-- Revision 0.01 - File Created
-- Additional Comments:
```

```
--
----------------------------------------------------------------------------------


library IEEE;
use IEEE.STD_LOGIC_1164.ALL;

-- Uncomment the following library declaration if using
-- arithmetic functions with Signed or Unsigned values
use IEEE.NUMERIC_STD.ALL;
use IEEE.std_logic_unsigned.all;

-- Uncomment the following library declaration if instantiating
-- any Xilinx leaf cells in this code.
--library UNISIM;
--use UNISIM.VComponents.all;

entity monitoring is
    Port (
        Global_Net_Clk      : in  std_logic;
        Strobe              : in  std_logic;
        Capture             : in  std_logic;
        Shift               : in  std_logic;
        Fire_Out            : in  std_logic;
        No_Input_Fire       : in  std_logic;
        Synapse_Fire        : in  std_logic;
        Reset               : in  std_logic;
        Reg_Accum_Sum       : in  std_logic_vector (7 downto 0);
        Neuron_Synapse      : in  std_logic;
        Serial_Input        : in  std_logic;
        Serial_Output       : out std_logic
    );
end monitoring;

architecture Behavioral of monitoring is

    COMPONENT c_counter_binary_0
      PORT (
        CLK : IN STD_LOGIC;
        CE : IN STD_LOGIC;
        SCLR : IN STD_LOGIC;
        Q : OUT STD_LOGIC_VECTOR(15 DOWNTO 0)
      );
    END COMPONENT;
    ATTRIBUTE SYN_BLACK_BOX : BOOLEAN;
    ATTRIBUTE SYN_BLACK_BOX OF c_counter_binary_0 : COMPONENT IS TRUE;
    ATTRIBUTE BLACK_BOX_PAD_PIN : STRING;
    ATTRIBUTE BLACK_BOX_PAD_PIN OF c_counter_binary_0 : COMPONENT IS "CLK,CE,SCLR,Q[15:0]";

    component eight_b_counter is
        Port (
            clk         : in  std_logic;
            reset       : in  std_logic;
            increment   : in  std_logic;
            decrement   : in  std_logic;
            value       : out std_logic_vector (7 downto 0)
        );
    end component;
```

```
    signal capture_reg : std_logic;
    signal N : std_logic_vector (15 downto 0);
    signal SR_Content : std_logic_vector (31 downto 0);
    signal stored_fires : std_logic_vector (7 downto 0);

begin

--    capture_FF : process( Global_Net_Clk )
--    begin
--        if(rising_edge(Global_Net_Clk)) then
--            capture_reg <= capture;
--        end if;
--    end process;

    fire_counter : c_counter_binary_0
      PORT MAP (
        CLK => ( Global_Net_Clk or ( Strobe and Capture ) ),
        CE => ( Fire_Out and Global_Net_Clk ),
        SCLR => capture,
        Q => N
      );

--    fire_counter : process(Global_Net_Clk)
--    begin
--        if(rising_edge(Global_Net_Clk)) then
--            if(capture_reg = '0') then
--                if( Fire_Out = '1') then
--                    N <= N + 1;
--                else
--                    N <= N;
--                end if;
--            else
--                N <= "0000000000000000";
--            end if;
--        end if;
--    end process;

    Serial_Output <= SR_Content(31);

    monitoring_SR : process(Strobe)
    begin
        if(rising_edge(Strobe)) then
            if( reset = '0' ) then
                if( capture = '1' ) then
                    SR_Content <= Reg_Accum_Sum & N & stored_fires;
                elsif( capture = '0' and shift = '1' ) then
--                    Serial_Output <= SR_Content(31);
                    SR_Content <= SR_Content(30 downto 0) & Serial_Input;
                end if;
            else
                SR_Content <= "00000000000000000000000000000000";
            end if;
        end if;
    end process;

    dist_reg_mon : eight_b_counter
    Port Map (
        clk => Global_Net_Clk,
        reset => (neuron_synapse or reset),
```

```
        increment => No_Input_Fire,
        decrement => Synapse_Fire,
        value => stored_fires
    );

end Behavioral;
```

A.5 Output Control

```
----------------------------------------------------------------------------------
-- Company:
-- Engineer:
--
-- Create Date: 12/04/2014 10:19:42 AM
-- Design Name:
-- Module Name: output_control - Behavioral
-- Project Name:
-- Target Devices:
-- Tool Versions:
-- Description:
--
-- Dependencies:
--
-- Revision:
-- Revision 0.01 - File Created
-- Additional Comments:
--
----------------------------------------------------------------------------------


library IEEE;
use IEEE.STD_LOGIC_1164.ALL;

-- Uncomment the following library declaration if using
-- arithmetic functions with Signed or Unsigned values
--use IEEE.NUMERIC_STD.ALL;

-- Uncomment the following library declaration if instantiating
-- any Xilinx leaf cells in this code.
library UNISIM;
use UNISIM.VComponents.all;

entity output_control is
    Port (
        Accum_Clr               : in  std_logic;
        Acquire_Fire_Clk        : in  std_logic;
        Global_Net_Clk          : in  std_logic;
        Neuron_Synapse          : in  std_logic;
        Special_Reset           : in  std_logic;
        Synapse_Fire            : in  std_logic;
        Reset                   : in  std_logic;
        Reg_Accum_Sum           : in  std_logic_vector (7 downto 0);
        Threshold               : in  std_logic_vector (7 downto 0);
        Element_Output          : out std_logic_vector (7 downto 0);
        Fire                    : out std_logic;
        Fire_Complete           : out std_logic;
```

```
        Fire_Out                : out std_logic
    );
end output_control;

architecture Behavioral of output_control is

    signal Fire_Pulse : std_logic;
    signal Fire_sig : std_logic;
    signal Fire_Complete_sig : std_logic;
    signal Synapse_Weight : std_logic_vector (7 downto 0);

begin

    Fire <= Fire_sig;
    Fire_Complete <= Fire_Complete_sig;


--  Element Output Control Function

-- SRL16E: 16-bit shift register LUT with clock enable operating on posedge of clock (Mapped to
     SliceM LUT6)
--         Virtex-7
-- Xilinx HDL Language Template, version 2013.3

    Fire_Period : SRL16E  --  Set the fire period to one network cycle or 8 element cycles
    generic map (
      INIT => X"0000")
    port map (
      Q => Fire_Complete_sig,       -- SRL data output
      A0 => '0',                    -- Select[0] input
      A1 => '1',                    -- Select[1] input
      A2 => '1',                    -- Select[2] input
      A3 => '1',                    -- Select[3] input
      CE => '1',                    -- Clock enable input
      CLK => Acquire_Fire_Clk,      -- Clock input
      D => Fire_Pulse               -- SRL data input
    );

    output_port :   process (Global_Net_Clk, Fire_sig, Synapse_Fire, Threshold, Synapse_Weight) --
          Diives Synapse Weight or Neuron Charge to output during fire command
    begin
--  Latch the active Synapse Weight to hold constant during the present global network cycle
      if  (rising_edge (Global_Net_Clk))  then
          if (Reset = '1')  then  --  SRSet input on register..clear if set
                Synapse_Weight  <=  "00000000";  --  active synapse weight should be 0 if element
                     is not firing
          else
                Synapse_Weight  <=  Reg_Accum_Sum;
          end if;
      end if;

--  Mux the appropriate output value during a firing command
          if (Fire_sig = '1' and Synapse_Fire = '0') then
              Element_Output  <=  Threshold;
          elsif (Fire_sig = '0' and Synapse_Fire = '1') then
              Element_Output  <=  Synapse_Weight;
          else
              Element_Output  <=  "00000000";
          end if;
```

```
    end process  output_port;


--  Element Fire, Fire_Pulse and Fire_Out control Signals
    element_fire_control    :   process (Acquire_Fire_Clk, Synapse_Fire, Fire_sig, Special_Reset)
    begin

--  Fire element on charge >= threshold if a neuron
        if  (rising_edge  (Acquire_Fire_Clk))    then
            if  (Special_Reset  =  '1')  then  --  Set Fire and Fire_Pulse outputs to 0
                Fire_sig  <=  '0';
                Fire_Pulse  <=  '0';
            else
                Fire_sig    <=  (Neuron_Synapse and Accum_Clr and (not Fire_sig)          ) or --
                    Not firing
                        (Neuron_Synapse and Accum_Clr and Fire_Complete_sig       ) or -- Done
                            FIring
                        (Neuron_Synapse and Fire_sig      and (not Fire_Complete_sig) );   --
                            Firing and not done firing

--  Pulse if fire condition identified if a neuron, used as input to create firing time of 8-
    cycles
                Fire_Pulse  <=  (Neuron_Synapse and Accum_Clr and (not Fire_Pulse) and (not
                    Fire_sig)) or
                            (Neuron_Synapse and Accum_Clr and (not Fire_Pulse) and
                                Fire_Complete_sig);
            end if;
        end if;

--  Fire output signal for one network cycle when fire condition identified, neuron or synapse
        Fire_Out <= (Fire_sig or Synapse_Fire);
    end process element_fire_control;

end Behavioral;
```

A.6 Registers

```
----------------------------------------------------------------------------------
-- Company:
-- Engineer:
--
-- Create Date: 12/03/2014 02:57:42 PM
-- Design Name:
-- Module Name: Prog_Registers - Behavioral
-- Project Name:
-- Target Devices:
-- Tool Versions:
-- Description:
--
-- Dependencies:
--
-- Revision:
-- Revision 0.01 - File Created
-- Additional Comments:
--
----------------------------------------------------------------------------------
```

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;

-- Uncomment the following library declaration if using
-- arithmetic functions with Signed or Unsigned values
--use IEEE.NUMERIC_STD.ALL;

-- Uncomment the following library declaration if instantiating
-- any Xilinx leaf cells in this code.
--library UNISIM;
--use UNISIM.VComponents.all;

entity Prog_Registers is
    Generic ( Elem_Addr : std_logic_vector (15 downto 0) := "0000000000000000" );
    Port (
        Prog_Data               : in  std_logic_vector (31 downto 0);
        Array_Addr              : in  std_logic_vector (15 downto 0);
        Strobe                  : in  std_logic;
        Reset                   : in  std_logic;
        Threshold               : out std_logic_vector (7 downto 0);
        Synapse_Distance        : out std_logic_vector (7 downto 0);
        Input_Enable            : out std_logic_vector (15 downto 0);
        Neuron_Synapse          : out std_logic;
        Element_Select          : out std_logic;
        Output_Select           : out std_logic_vector (3 downto 0);
        Wt_Refrac_Per           : out std_logic_vector (3 downto 0)
    );
end Prog_Registers;

architecture Behavioral of Prog_Registers is

    signal threshold_sig : std_logic_vector (7 downto 0);
    signal synapse_distance_sig : std_logic_vector (7 downto 0);
    signal input_enable_sig : std_logic_vector (15 downto 0);
    signal neuron_synapse_sig : std_logic;
    signal output_select_sig : std_logic_vector (3 downto 0);
    signal wt_refrac_per_sig : std_logic_vector (3 downto 0);
    signal FF_STROBE : std_logic;
    signal Element_Select_sig : std_logic;
    signal Element_Control_sig : std_logic_vector (8 downto 0);

    constant Element_Address : std_logic_vector (15 downto 0) := Elem_Addr;

begin

    Threshold <= threshold_sig;
    Synapse_Distance <= synapse_distance_sig;
    Input_Enable <= input_enable_sig;
    Neuron_Synapse <= neuron_synapse_sig;
    Output_Select <= output_select_sig;
    Wt_Refrac_Per <= wt_refrac_per_sig;
    Element_Select <= Element_Select_sig;

    StrobeT : process(Strobe,Reset)
    begin
        if (Reset = '0') then --  normal operation
            if (rising_edge (Strobe)) then
```

```
                if( Element_Select_sig = '1' ) then
                    FF_STROBE <= not FF_STROBE;
                end if;
            end if;
        elsif (Reset = '1')  then  --Reset Param
            FF_STROBE <= '0';
        end if;
    end process StrobeT;

--  Initialize programmable registers via program interface
--  Decoder for Element Select
    Element_Sel  :  process  (Array_Addr)
    begin
        if (Array_Addr = Element_Address) then
            Element_Select_sig <= '1';
        else
            Element_Select_sig <= '0';
        end if;
    end process Element_Sel;

--  Threshold Register  <=  initially set to "all 0s" at reset;
    threshold_register  :  process  (Strobe,Reset)
    begin
        if (Reset = '0')  then --  normal operation
            if (rising_edge (Strobe))  then
                if (Element_Select_sig = '1' and FF_STROBE = '0')  then -- First Strobe Cycle
                    threshold_sig <= Prog_Data (15 downto 8);
                elsif (Element_Select_sig = '1' and FF_STROBE = '1')  then -- Second Strobe Cycle
                    threshold_sig <= threshold_sig;
                end if;
            end if;
        elsif (Reset = '1')  then  --Reset Param
            threshold_sig <= "00000000";
        end if;
    end process threshold_register;

--  Synapse_Distance Register  <=  initially set to "00000000" at reset;
    synapse_distance_register  :  process  (Strobe,Reset)
    begin
        if (Reset = '0')  then --  normal operation
            if (rising_edge (Strobe))  then
                if (Element_Select_sig = '1' and FF_STROBE = '0')  then -- First Strobe Cycle
                    synapse_distance_sig <= Prog_Data (7 downto 0);
                elsif (Element_Select_sig = '1' and FF_STROBE = '1')  then -- Second Strobe Cycle
                    synapse_distance_sig <= synapse_distance_sig;
                end if;
            end if;
        elsif (Reset = '1')  then  --Reset Param
            synapse_distance_sig <= "00000000";
        end if;
    end process synapse_distance_register;

--  Input_Enable Register  <=  initially set to "all 0s" at reset;
    input_enable_register  :  process  (Strobe, Reset)
    begin
         if (Reset = '0')  then --  normal operation
            if (rising_edge (Strobe))  then
                if (Element_Select_sig = '1' and FF_STROBE = '0')  then -- First Strobe Cycle
                    input_enable_sig (15 downto 0) <= input_enable_sig (15 downto 0);
```

```
                elsif (Element_Select_sig = '1' and FF_STROBE = '1')  then -- Second Strobe Cycle
                    input_enable_sig (15 downto 0) <= Prog_Data (15 downto 0);
                end if;
            end if;
        elsif (Reset = '1')  then  --Reset Param
            input_enable_sig  <=  "0000000000000000";
        end if;
    end process input_enable_register;

--    Neuron_Synapse and Output_Sel in Element Control Register
--    Neuron_Synapse    <=  '1';  Set all elements to "neuron" mode at reset
--    Output_Sel        <=  "000";  default value, only used when element is a "synapse"

    element_control_register  :   process  (Strobe, Reset, element_control_sig)
    begin
        if (Reset = '0')  then --  normal operation
            if (rising_edge (Strobe))  then
                if (Element_Select_sig = '1' and FF_STROBE = '0')  then -- First Strobe Cycle
                    element_control_sig <= Prog_Data (27 downto 24) & Prog_Data (20 downto 16);
                elsif (Element_Select_sig = '1' and FF_STROBE = '1')  then -- Second Strobe Cycle
                    element_control_sig <= element_control_sig;
                end if;
            end if;
        elsif (Reset = '1')  then  --Reset Param
            element_control_sig <=  "000000000";
        end if;
        neuron_synapse_sig   <=  element_control_sig(0);
        output_select_sig(0)    <=  element_control_sig(1);
        output_select_sig(1)    <=  element_control_sig(2);
        output_select_sig(2)    <=  element_control_sig(3);
        output_select_sig(3)    <=  element_control_sig(4);
        wt_refrac_per_sig(0) <=  element_control_sig(5);
        wt_refrac_per_sig(1) <=  element_control_sig(6);
        wt_refrac_per_sig(2) <=  element_control_sig(7);
        wt_refrac_per_sig(3) <=  element_control_sig(8);
    end process element_control_register;


end Behavioral;
```

A.7 Synapse Distance

```
----------------------------------------------------------------------------------
-- Company:
-- Engineer:
--
-- Create Date: 12/03/2014 02:38:15 PM
-- Design Name:
-- Module Name: synapse_distance - Behavioral
-- Project Name:
-- Target Devices:
-- Tool Versions:
-- Description:
--
-- Dependencies:
--
-- Revision:
```

```
-- Revision 0.01 - File Created
-- Additional Comments:
--
----------------------------------------------------------------------------------


library IEEE;
use IEEE.STD_LOGIC_1164.ALL;

-- Uncomment the following library declaration if using
-- arithmetic functions with Signed or Unsigned values
--use IEEE.NUMERIC_STD.ALL;

-- Uncomment the following library declaration if instantiating
-- any Xilinx leaf cells in this code.
--library UNISIM;
--use UNISIM.VComponents.all;

entity synapse_distance_comp is
    Port (
        Global_Net_Clk      : in  std_logic;
        Input_Fire_0        : in  std_logic;
        Input_Fire_1        : in  std_logic;
        Input_Fire_2        : in  std_logic;
        Input_Fire_3        : in  std_logic;
        Input_Fire_4        : in  std_logic;
        Input_Fire_5        : in  std_logic;
        Input_Fire_6        : in  std_logic;
        Input_Fire_7        : in  std_logic;
        Input_Fire_8        : in  std_logic;
        Input_Fire_9        : in  std_logic;
        Input_Fire_10       : in  std_logic;
        Input_Fire_11       : in  std_logic;
        Input_Fire_12       : in  std_logic;
        Input_Fire_13       : in  std_logic;
        Input_Fire_14       : in  std_logic;
        Input_Fire_15       : in  std_logic;
        Neuron_Synapse      : in  std_logic;
        Input_Enable        : in  std_logic_vector (15 downto 0);
        Synapse_Distance    : in  std_logic_vector (7 downto 0);
        No_Input_Fire       : out std_logic;
        Synapse_Fire        : out std_logic
    );
end synapse_distance_comp;

architecture Behavioral of synapse_distance_comp is

    COMPONENT c_shift_ram_0
    PORT (
        A : IN STD_LOGIC_VECTOR(7 DOWNTO 0);
        D : IN STD_LOGIC_VECTOR(0 DOWNTO 0);
        CLK : IN STD_LOGIC;
        Q : OUT STD_LOGIC_VECTOR(0 DOWNTO 0);
        SCLR : IN STD_LOGIC
        );
    END COMPONENT;

    signal No_Input_Fire_sig : std_logic;
    signal No_Input_Fire_Vector : std_logic_vector (0 downto 0);
```

```
    signal Synapse_Fire_Vector : std_logic_vector (0 downto 0);

begin

    No_Input_Fire <= No_Input_Fire_sig;
    Synapse_Fire <= Synapse_Fire_Vector(0);
    No_Input_Fire_Vector(0) <= No_Input_Fire_sig;

    with  Input_Enable  select
    No_Input_Fire_sig  <= Input_Fire_0  when "0000000000000001",
                          Input_Fire_1  when "0000000000000010",
                          Input_Fire_2  when "0000000000000100",
                          Input_Fire_3  when "0000000000001000",
                          Input_Fire_4  when "0000000000010000",
                          Input_Fire_5  when "0000000000100000",
                          Input_Fire_6  when "0000000001000000",
                          Input_Fire_7  when "0000000010000000",
                          Input_Fire_8  when "0000000100000000",
                          Input_Fire_9  when "0000001000000000",
                          Input_Fire_10 when "0000010000000000",
                          Input_Fire_11 when "0000100000000000",
                          Input_Fire_12 when "0001000000000000",
                          Input_Fire_13 when "0010000000000000",
                          Input_Fire_14 when "0100000000000000",
                          Input_Fire_15 when "1000000000000000",
                                  ('0') when others;

    Synapse_Distance_Function : c_shift_ram_0
    PORT MAP (
        A => Synapse_Distance,          -- address input
        D => No_Input_Fire_Vector,      -- data input
        CLK => Global_Net_Clk,          -- clock
        Q => Synapse_Fire_Vector,       -- data output
        SCLR => Neuron_Synapse          -- synchronous clear
    );

end Behavioral;
```

A.8 Clocking Element

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;

entity clocking_element is
    Port (
        reset                   : in  std_logic;    -- Reset the clock generator
        Accum_Clk               : out std_logic;    -- Accumulator Clock (16 MHz)
        Accum_Clk_On            : out std_logic;    -- Accumulator Clock that always outputs
        Acquire_Fire_Clk        : out std_logic;    -- Acquire/Fire Clock (8 MHz)
        Acquire_Fire_Clk_On     : out std_logic;    -- Acquire/Fire Clock that always outputs
        Accum_En_Clk            : out std_logic;    -- Accumulator Enable Clock (8 MHz / 90 deg
            shift)
        Global_Net_Clk          : out std_logic;    -- Global Network Clock (1 MHz)
        Global_Net_Clk_On       : out std_logic;    -- Global Network Clock that always outputs
        Input_Sel_Ctr_Load      : out std_logic;    -- Load signal to the Global Input Select
            counter
        Global_CE               : in  std_logic;     -- Global Clock Enable (synchronous)
```

```
        Accum_Clk_Pin           : in std_logic;
        Acquire_Fire_Clk_Pin    : in std_logic;
        Accum_En_Clk_Pin        : in std_logic;
        Global_Net_Clk_Pin      : in std_logic

    );
end clocking_element;

architecture Behavioral of clocking_element is

    COMPONENT c_counter_binary_2
        PORT (
            CLK : IN STD_LOGIC;
            SCLR : IN STD_LOGIC;
            Q : OUT STD_LOGIC_VECTOR(3 DOWNTO 0)
        );
    END COMPONENT;

    signal ac           : std_logic := '0';
    signal ac_inv       : std_logic := '0';
    signal ac_output    : std_logic := '0';
    signal afc          : std_logic := '0';
    signal aec          : std_logic := '0';
    signal gnc          : std_logic := '0';
    signal gnc_en       : std_logic := '0';
    signal islc         : std_logic := '0';
    signal islc_counter : std_logic_vector (3 downto 0) := "0000";
    signal islc_latched : std_logic_vector (3 downto 0) := "0000";
    signal latched2     : std_logic_vector (3 downto 0) := "0000";
    signal internal_ce  : std_logic := '1';

begin

    ac <= Accum_Clk_Pin;
    ac_inv <= not Accum_Clk_Pin;
    afc <= Acquire_Fire_Clk_Pin;
    aec <= Accum_En_Clk_Pin;
    gnc <= Global_Net_Clk_Pin;

    Accum_Clk <= (ac AND internal_ce);
    Accum_Clk_On <= ac;
    Acquire_Fire_Clk <= (afc AND internal_ce);
    Acquire_Fire_Clk_On <= afc;
    Accum_En_Clk <= (aec AND internal_ce);
    Global_Net_Clk <= (gnc AND internal_ce);
    Global_Net_Clk_On <= gnc;
    Input_Sel_Ctr_Load <= (islc OR NOT internal_ce);

    counter : c_counter_binary_2
    PORT MAP (
        CLK => afc,
        SCLR => reset,
        Q => islc_counter
    );

    counter_latch : process (afc)
    begin
        if(rising_edge(afc)) then
```

```
            islc_latched <= islc_counter;
        end if;
    end process;

    islc_latch : process (ac)
    begin
        if(rising_edge(ac)) then
            latched2 <= islc_counter;
        end if;
    end process;

    islc_divider : process (reset, ac)
    begin
        if(reset = '1') then
            islc <= '0';
        elsif(falling_edge(afc)) then
            if(islc_latched = "0000") then
                islc <= '1';
            else
                islc <= '0';
            end if;
        end if;
    end process;

    globalce : process (Global_CE, ac_inv)
    begin
        if(latched2 = "0001") then
            if(rising_edge(ac_inv)) then
                internal_ce <= Global_CE;
            end if;
        end if;
    end process globalce;

end Behavioral;
```

A.9 Random Number Generator

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;

entity gis_gen is
    Port (
        Global_Net_Clk          : in std_logic;
        Reset                   : in std_logic;
        Acquire_Fire_Clk        : in std_logic;
        Input_Sel_Ctr_Load      : in std_logic;
        Global_Input_Sel        : out std_logic_vector (3 downto 0)
    );
end gis_gen;

architecture behavioral of gis_gen is

    COMPONENT protoann_PRNG_4bit
        PORT (
            Global_Net_Clk      : IN  STD_LOGIC;
            Reset               : IN  STD_LOGIC;
            LFSR_out            : OUT STD_LOGIC_VECTOR(3 DOWNTO 0)
```

```
        );
    END COMPONENT;

    COMPONENT prng_counter_comp
        PORT (
            Acquire_Fire_Clk    : IN  STD_LOGIC;
            Reset               : IN  STD_LOGIC;
            Input_Sel_Ctr_Load  : IN  STD_LOGIC;
            LFSR_value          : IN  STD_LOGIC_VECTOR(3 downto 0);
            Global_Input_Sel    : OUT STD_LOGIC_VECTOR(3 DOWNTO 0)
        );
    END COMPONENT;

    signal LFSR_out : std_logic_vector ( 3 downto 0);

begin

    PRNG : protoann_PRNG_4bit
    Port Map (
        Global_Net_Clk => Global_Net_Clk,
        Reset => Reset,
        LFSR_out => LFSR_out
    );

    counter : prng_counter_comp
    Port Map (
        Acquire_Fire_Clk => Acquire_Fire_Clk,
        Reset => Reset,
        Input_Sel_Ctr_Load => Input_Sel_Ctr_Load,
        LFSR_value => LFSR_out,
        Global_Input_Sel => Global_Input_Sel
    );

end behavioral;
```

A.10 FIFO Logic

```
-- Component to interface with FIFO and assemble data pieces
-- Copyright (C) 2014 Christopher Daffron
--
-- Permission is hereby granted, free of charge, to any person obtaining a copy
-- of this software and associated documentation files (the "Software"), to deal
-- in the Software without restriction, including without limitation the rights
-- to use, copy, modify, merge, publish, distribute, sublicense, and/or sell
-- copies of the Software, and to permit persons to whom the Software is
-- furnished to do so, subject to the following conditions:
--
-- The above copyright notice and this permission notice shall be included in
-- all copies or substantial portions of the Software.
--
-- THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, EXPRESS OR
-- IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY,
-- FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT. IN NO EVENT SHALL THE
-- AUTHORS OR COPYRIGHT HOLDERS BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER
-- LIABILITY, WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING FROM,
-- OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER DEALINGS IN
-- THE SOFTWARE.
```

```
----------------------------------------------------------------------------------
-- Company:
-- Engineer:
--
-- Create Date: 04/06/2014 07:48:50 PM
-- Design Name:
-- Module Name: fifo_logic - Behavioral
-- Project Name:
-- Target Devices:
-- Tool Versions:
-- Description:
--
-- Dependencies:
--
-- Revision:
-- Revision 0.01 - File Created
-- Revision 1.00 - Initial Version
-- Revision 2.00 - Updated for new 512 bit output
-- Additional Comments:
--
----------------------------------------------------------------------------------


library IEEE;
use IEEE.STD_LOGIC_1164.ALL;

-- Uncomment the following library declaration if using
-- arithmetic functions with Signed or Unsigned values
--use IEEE.NUMERIC_STD.ALL;

-- Uncomment the following library declaration if instantiating
-- any Xilinx leaf cells in this code.
--library UNISIM;
--use UNISIM.VComponents.all;

entity fifo_logic is
    Port (
        -- Clocks
        Global_Net_Clk_On       : in  std_logic;
        Accum_Clk_On         : in  std_logic;
        -- FIFO signals
        din                     : out std_logic_vector (31 downto 0);
        wr_en_bar               : out std_logic;
        rd_en_bar               : out std_logic;
        dout                    : in  std_logic_vector (31 downto 0);
        full_bar                : in  std_logic;
        empty_bar               : in  std_logic;
        -- To next logic stage
        Data                    : out std_logic_vector (287 downto 0);
        Data_Enable             : out std_logic;
        External_Data           : in  std_logic_vector (511 downto 0);
        Ext_Data_Present        : in  std_logic
    );
end fifo_logic;

architecture Behavioral of fifo_logic is

    COMPONENT fifo_slow_counter
```

```
        PORT (
            CLK : IN STD_LOGIC;
            CE : IN STD_LOGIC;
            Q : OUT STD_LOGIC_VECTOR (4 DOWNTO 0)
        );
    END COMPONENT;

    type fsm1_type is (S0, S1);
    signal fsm1 : fsm1_type := S0;
    type fsm2_type is (S0, S1, S2, S3, S4, S5, S6, S7, S8, S9, S10);
    signal fsm2 : fsm2_type := S0;
    type fsm3_type is (S0, S1, S2, S3, S4, S5, S6, S7, S8, S9, S10, S11, S12, S13, S14, S15, S16);
    signal fsm3 : fsm3_type := S0;
    signal medCounterEnable : std_logic := '0';
    signal Ext_Data_local : std_logic_vector (511 downto 0);
    signal medCounterLatched : std_logic_vector (4 downto 0);
    signal medCounterOut : std_logic_vector (4 downto 0);
    signal bigCounterLatched : std_logic_vector (4 downto 0);
    signal bigCounterOut : std_logic_vector (4 downto 0);
    signal data_en : std_logic;
    signal tempOut : std_logic_vector (287 downto 0);
    signal din_temp : std_logic_vector (31 downto 0);
    signal wr_en_temp : std_logic;
    signal full : std_logic;
    signal empty : std_logic;
    signal wr_en : std_logic;
    signal rd_en : std_logic;

begin

    rd_en_bar <= not rd_en;
    wr_en_bar <= not wr_en;
    full <= not full_bar;
    empty <= not empty_bar;

    medCounter : fifo_slow_counter
    PORT MAP (
        CLK => Accum_Clk_On,
        CE => medCounterEnable,
        Q => medCounterOut
    );

    bigCounter : fifo_slow_counter
    PORT MAP (
        CLK => Accum_Clk_On,
        CE => medCounterEnable,
        Q => bigCounterOut
    );

    counter_latch : process (Accum_Clk_On)
    begin
        if(rising_edge(Accum_Clk_On)) then
            medCounterLatched <= medCounterOut;
        end if;
    end process;

    big_latch : process (Accum_Clk_On)
    begin
        if(rising_edge(Accum_Clk_On)) then
```

```
            bigCounterLatched <= bigCounterOut;
        end if;
    end process;

    process(Global_Net_Clk_On)
    begin
        if(rising_edge(Global_Net_Clk_On)) then
            case fsm1 is
                when S0 =>
                    medCounterEnable <= '1';
                    fsm1 <= S1;
                when S1 =>
                    fsm1 <= S1;
            end case;
        end if;
    end process;

    -- Receive data for the board
    process(Accum_Clk_On, medCounterLatched, empty)
    begin
        if(rising_edge(Accum_Clk_On)) then
            case fsm2 is
                when S0 =>
                    if(medCounterLatched = "11111") then
                        Data <= tempOut;
                        Data_Enable <= data_en;
                        if(empty = '0') then
                            fsm2 <= S1;
                            data_en <= '1';
                        else
                            tempOut <= (others => '0');
                            data_en <= '0';
                        end if;
                    else
                        fsm2 <= S0;
                    end if;
                when S1 =>
                    if(medCounterLatched = "00001") then
                        rd_en <= '1';
                        fsm2 <= S2;
                    else
                        fsm2 <= S1;
                    end if;
                when S2 =>
                    if(medCounterLatched = "00011") then
                        tempOut(287 downto 256) <= dout;
                        fsm2 <= S3;
                    else
                        fsm2 <= S2;
                    end if;
                when S3 =>
                    if(medCounterLatched = "00101") then
                        tempOut(255 downto 224) <= dout;
                        fsm2 <= S4;
                    else
                        fsm2 <= S3;
                    end if;
                when S4 =>
                    if(medCounterLatched = "00111") then
```

```
                    tempOut(223 downto 192) <= dout;
                    fsm2 <= S5;
                else
                    fsm2 <= S4;
                end if;
            when S5 =>
                if(medCounterLatched = "01001") then
                    tempOut(191 downto 160) <= dout;
                    fsm2 <= S6;
                else
                    fsm2 <= S5;
                end if;
            when S6 =>
                if(medCounterLatched = "01011") then
                    tempOut(159 downto 128) <= dout;
                    fsm2 <= S7;
                else
                    fsm2 <= S6;
                end if;
            when S7 =>
                if(medCounterLatched = "01101") then
                    tempOut(127 downto 96) <= dout;
                    fsm2 <= S8;
                else
                    fsm2 <= S7;
                end if;
            when S8 =>
                if(medCounterLatched = "01111") then
                    tempOut(95 downto 64) <= dout;
                    fsm2 <= S9;
                else
                    fsm2 <= S8;
                end if;
            when S9 =>
                if(medCounterLatched = "10001") then
                    tempOut(63 downto 32) <= dout;
                    fsm2 <= S10;
                else
                    fsm2 <= S9;
                end if;
            when S10 =>
                if(medCounterLatched = "10011") then
                    tempOut(31 downto 0) <= dout;
                    rd_en <= '0';
                    fsm2 <= S0;
                else
                    fsm2 <= S10;
                end if;
        end case;
    end if;
end process;

-- Send data off the board
process(Accum_Clk_On, bigCounterLatched)
begin
    if(rising_edge(Accum_Clk_On)) then
        case fsm3 is
            when S0 =>
                if(bigCounterLatched = "11111") then
```

```
            if(Ext_Data_Present = '1') then
                Ext_Data_Local <= External_Data;
                fsm3 <= S1;
            else
                wr_en_temp <= '0';
                fsm3 <= S0;
            end if;
        else
            fsm3 <= S0;
        end if;
    when S1 =>
        if(bigCounterLatched = "00000") then
            wr_en_temp <=  '1';
            din_temp <= Ext_Data_Local(511 downto 480);
            fsm3 <= S2;
        else
            fsm3 <= S1;
        end if;
    when S2 =>
        if(bigCounterLatched = "00010") then
            din_temp <= Ext_Data_Local(479 downto 448);
            fsm3 <= S3;
        else
            fsm3 <= S2;
        end if;
    when S3 =>
        if(bigCounterLatched = "00100") then
            din_temp <= Ext_Data_Local(447 downto 416);
            fsm3 <= S4;
        else
            fsm3 <= S3;
        end if;
    when S4 =>
        if(bigCounterLatched = "00110") then
            din_temp <= Ext_Data_Local(415 downto 384);
            fsm3 <= S5;
        else
            fsm3 <= S4;
        end if;
    when S5 =>
        if(bigCounterLatched = "01000") then
            din_temp <= Ext_Data_Local(383 downto 352);
            fsm3 <= S6;
        else
            fsm3 <= S5;
        end if;
    when S6 =>
        if(bigCounterLatched = "01010") then
            din_temp <= Ext_Data_Local(351 downto 320);
            fsm3 <= S7;
        else
            fsm3 <= S6;
        end if;
    when S7 =>
        if(bigCounterLatched = "01100") then
            din_temp <= Ext_Data_Local(319 downto 288);
            fsm3 <= S8;
        else
            fsm3 <= S7;
```

```
        end if;
    when S8 =>
        if(bigCounterLatched = "01110") then
            din_temp <= Ext_Data_Local(287 downto 256);
            fsm3 <= S9;
        else
            fsm3 <= S8;
        end if;
    when S9 =>
        if(bigCounterLatched = "10000") then
            din_temp <= Ext_Data_Local(255 downto 224);
            fsm3 <= S10;
        else
            fsm3 <= S9;
        end if;
    when S10 =>
        if(bigCounterLatched = "10010") then
            din_temp <= Ext_Data_Local(223 downto 192);
            fsm3 <= S11;
        else
            fsm3 <= S10;
        end if;
    when S11 =>
        if(bigCounterLatched = "10100") then
            din_temp <= Ext_Data_Local(191 downto 160);
            fsm3 <= S12;
        else
            fsm3 <= S11;
        end if;
    when S12 =>
        if(bigCounterLatched = "10110") then
            din_temp <= Ext_Data_Local(159 downto 128);
            fsm3 <= S13;
        else
            fsm3 <= S12;
        end if;
    when S13 =>
        if(bigCounterLatched = "11000") then
            din_temp <= Ext_Data_Local(127 downto 96);
            fsm3 <= S14;
        else
            fsm3 <= S13;
        end if;
    when S14 =>
        if(bigCounterLatched = "11010") then
            din_temp <= Ext_Data_Local(95 downto 64);
            fsm3 <= S15;
        else
            fsm3 <= S14;
        end if;
    when S15 =>
        if(bigCounterLatched = "11100") then
            din_temp <= Ext_Data_Local(63 downto 32);
            fsm3 <= S16;
        else
            fsm3 <= S15;
        end if;
    when S16 =>
        if(bigCounterLatched = "11110") then
```

```
                        din_temp <= Ext_Data_Local(31 downto 0);
                        fsm3 <= S0;
                    else
                        fsm3 <= S16;
                    end if;
            end case;
        end if;
    end process;

    process(Accum_Clk_On)
    begin
        if(rising_edge(Accum_Clk_On)) then
            din <= din_temp;
            wr_en <= wr_en_temp;
        end if;
    end process;

end Behavioral;
```

A.11 Programming Interface

```
-- Array Programming Interface
-- Copyright (C) 2014 Christopher Daffron
--
-- Permission is hereby granted, free of charge, to any person obtaining a copy
-- of this software and associated documentation files (the "Software"), to deal
-- in the Software without restriction, including without limitation the rights
-- to use, copy, modify, merge, publish, distribute, sublicense, and/or sell
-- copies of the Software, and to permit persons to whom the Software is
-- furnished to do so, subject to the following conditions:
--
-- The above copyright notice and this permission notice shall be included in
-- all copies or substantial portions of the Software.
--
-- THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, EXPRESS OR
-- IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY,
-- FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT. IN NO EVENT SHALL THE
-- AUTHORS OR COPYRIGHT HOLDERS BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER
-- LIABILITY, WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING FROM,
-- OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER DEALINGS IN
-- THE SOFTWARE.

----------------------------------------------------------------------------------
-- Company: UTK EECS
-- Engineer: Chris Daffron
--
-- Create Date: 03/30/2014 10:01:36 PM
-- Design Name:
-- Module Name: prog_fsm - Behavioral
-- Project Name:
-- Target Devices:
-- Tool Versions:
-- Description:
--
-- Dependencies:
--
-- Revision:
```

```
-- Revision 0.01 - File Created
-- Revision 1.00 - Initial version
-- Revision 1.01 - Selective fire signals for external inputs
-- Revision 2.00 - 9/16/2014: Data output from array shift register
-- Additional Comments:
--
----------------------------------------------------------------------------------


library IEEE;
use IEEE.STD_LOGIC_1164.ALL;

-- Uncomment the following library declaration if using
-- arithmetic functions with Signed or Unsigned values
use IEEE.NUMERIC_STD.ALL;

-- Uncomment the following library declaration if instantiating
-- any Xilinx leaf cells in this code.
--library UNISIM;
--use UNISIM.VComponents.all;

entity prog_fsm is
    Port (
        -- CHANGE TO 288 BITS (287 - 0)
        Data                : in  std_logic_vector (287 downto 0); -- signal from FIFOs
        Global_Net_Clk_On   : in  std_logic; -- Always on GNC from clock module
        Acquire_Fire_Clk_On : in  std_logic; -- Always on AFC from clock module
        Acquire_Fire_Clk    : in  std_logic;
        Data_Enable         : in  std_logic; -- signal from FIFO unloading logic
        SR_output           : in  std_logic_vector (127 downto 0); -- Shift Register output from
            array
        Global_CE           : out std_logic; -- signal to clock module for clock enable
        Prog_Data           : out std_logic_vector (31 downto 0); -- Data out to elements
        Element_Addr        : out std_logic_vector (15 downto 0); -- Address of target element
        Strobe              : out std_logic; -- signal out to elements
        External_Data       : out std_logic_vector (511 downto 0); -- signal to PCIe
        Ext_Data_Present    : out std_logic; -- signal to FIFO loading logic
        Reset               : out std_logic;
        Capture             : out std_logic;
        Shift               : out std_logic;

        -- External Inputs
        -- CHANGE TO 32 INPUTS
        Ext_In_00_data      : out std_logic_vector (7 downto 0);
        Ext_In_01_data      : out std_logic_vector (7 downto 0);
        Ext_In_02_data      : out std_logic_vector (7 downto 0);
        Ext_In_03_data      : out std_logic_vector (7 downto 0);
        Ext_In_04_data      : out std_logic_vector (7 downto 0);
        Ext_In_05_data      : out std_logic_vector (7 downto 0);
        Ext_In_06_data      : out std_logic_vector (7 downto 0);
        Ext_In_07_data      : out std_logic_vector (7 downto 0);
        Ext_In_08_data      : out std_logic_vector (7 downto 0);
        Ext_In_09_data      : out std_logic_vector (7 downto 0);
        Ext_In_10_data      : out std_logic_vector (7 downto 0);
        Ext_In_11_data      : out std_logic_vector (7 downto 0);
        Ext_In_12_data      : out std_logic_vector (7 downto 0);
        Ext_In_13_data      : out std_logic_vector (7 downto 0);
        Ext_In_14_data      : out std_logic_vector (7 downto 0);
        Ext_In_15_data      : out std_logic_vector (7 downto 0);
```

```
Ext_In_16_data      : out std_logic_vector (7 downto 0);
Ext_In_17_data      : out std_logic_vector (7 downto 0);
Ext_In_18_data      : out std_logic_vector (7 downto 0);
Ext_In_19_data      : out std_logic_vector (7 downto 0);
Ext_In_20_data      : out std_logic_vector (7 downto 0);
Ext_In_21_data      : out std_logic_vector (7 downto 0);
Ext_In_22_data      : out std_logic_vector (7 downto 0);
Ext_In_23_data      : out std_logic_vector (7 downto 0);
Ext_In_24_data      : out std_logic_vector (7 downto 0);
Ext_In_25_data      : out std_logic_vector (7 downto 0);
Ext_In_26_data      : out std_logic_vector (7 downto 0);
Ext_In_27_data      : out std_logic_vector (7 downto 0);
Ext_In_28_data      : out std_logic_vector (7 downto 0);
Ext_In_29_data      : out std_logic_vector (7 downto 0);
Ext_In_30_data      : out std_logic_vector (7 downto 0);
Ext_In_31_data      : out std_logic_vector (7 downto 0);
Ext_In_00_fire      : out std_logic;
Ext_In_01_fire      : out std_logic;
Ext_In_02_fire      : out std_logic;
Ext_In_03_fire      : out std_logic;
Ext_In_04_fire      : out std_logic;
Ext_In_05_fire      : out std_logic;
Ext_In_06_fire      : out std_logic;
Ext_In_07_fire      : out std_logic;
Ext_In_08_fire      : out std_logic;
Ext_In_09_fire      : out std_logic;
Ext_In_10_fire      : out std_logic;
Ext_In_11_fire      : out std_logic;
Ext_In_12_fire      : out std_logic;
Ext_In_13_fire      : out std_logic;
Ext_In_14_fire      : out std_logic;
Ext_In_15_fire      : out std_logic;
Ext_In_16_fire      : out std_logic;
Ext_In_17_fire      : out std_logic;
Ext_In_18_fire      : out std_logic;
Ext_In_19_fire      : out std_logic;
Ext_In_20_fire      : out std_logic;
Ext_In_21_fire      : out std_logic;
Ext_In_22_fire      : out std_logic;
Ext_In_23_fire      : out std_logic;
Ext_In_24_fire      : out std_logic;
Ext_In_25_fire      : out std_logic;
Ext_In_26_fire      : out std_logic;
Ext_In_27_fire      : out std_logic;
Ext_In_28_fire      : out std_logic;
Ext_In_29_fire      : out std_logic;
Ext_In_30_fire      : out std_logic;
Ext_In_31_fire      : out std_logic;

-- External Outputs
-- CHANGE TO 32 OUTPUTS
Ext_Out_00_data     : in  std_logic_vector (7 downto 0);
Ext_Out_01_data     : in  std_logic_vector (7 downto 0);
Ext_Out_02_data     : in  std_logic_vector (7 downto 0);
Ext_Out_03_data     : in  std_logic_vector (7 downto 0);
Ext_Out_04_data     : in  std_logic_vector (7 downto 0);
Ext_Out_05_data     : in  std_logic_vector (7 downto 0);
Ext_Out_06_data     : in  std_logic_vector (7 downto 0);
Ext_Out_07_data     : in  std_logic_vector (7 downto 0);
```

```
        Ext_Out_08_data     : in  std_logic_vector (7 downto 0);
        Ext_Out_09_data     : in  std_logic_vector (7 downto 0);
        Ext_Out_10_data     : in  std_logic_vector (7 downto 0);
        Ext_Out_11_data     : in  std_logic_vector (7 downto 0);
        Ext_Out_12_data     : in  std_logic_vector (7 downto 0);
        Ext_Out_13_data     : in  std_logic_vector (7 downto 0);
        Ext_Out_14_data     : in  std_logic_vector (7 downto 0);
        Ext_Out_15_data     : in  std_logic_vector (7 downto 0);
        Ext_Out_16_data     : in  std_logic_vector (7 downto 0);
        Ext_Out_17_data     : in  std_logic_vector (7 downto 0);
        Ext_Out_18_data     : in  std_logic_vector (7 downto 0);
        Ext_Out_19_data     : in  std_logic_vector (7 downto 0);
        Ext_Out_20_data     : in  std_logic_vector (7 downto 0);
        Ext_Out_21_data     : in  std_logic_vector (7 downto 0);
        Ext_Out_22_data     : in  std_logic_vector (7 downto 0);
        Ext_Out_23_data     : in  std_logic_vector (7 downto 0);
        Ext_Out_24_data     : in  std_logic_vector (7 downto 0);
        Ext_Out_25_data     : in  std_logic_vector (7 downto 0);
        Ext_Out_26_data     : in  std_logic_vector (7 downto 0);
        Ext_Out_27_data     : in  std_logic_vector (7 downto 0);
        Ext_Out_28_data     : in  std_logic_vector (7 downto 0);
        Ext_Out_29_data     : in  std_logic_vector (7 downto 0);
        Ext_Out_30_data     : in  std_logic_vector (7 downto 0);
        Ext_Out_31_data     : in  std_logic_vector (7 downto 0);
        Ext_Out_00_fire     : in  std_logic;
        Ext_Out_01_fire     : in  std_logic;
        Ext_Out_02_fire     : in  std_logic;
        Ext_Out_03_fire     : in  std_logic;
        Ext_Out_04_fire     : in  std_logic;
        Ext_Out_05_fire     : in  std_logic;
        Ext_Out_06_fire     : in  std_logic;
        Ext_Out_07_fire     : in  std_logic;
        Ext_Out_08_fire     : in  std_logic;
        Ext_Out_09_fire     : in  std_logic;
        Ext_Out_10_fire     : in  std_logic;
        Ext_Out_11_fire     : in  std_logic;
        Ext_Out_12_fire     : in  std_logic;
        Ext_Out_13_fire     : in  std_logic;
        Ext_Out_14_fire     : in  std_logic;
        Ext_Out_15_fire     : in  std_logic;
        Ext_Out_16_fire     : in  std_logic;
        Ext_Out_17_fire     : in  std_logic;
        Ext_Out_18_fire     : in  std_logic;
        Ext_Out_19_fire     : in  std_logic;
        Ext_Out_20_fire     : in  std_logic;
        Ext_Out_21_fire     : in  std_logic;
        Ext_Out_22_fire     : in  std_logic;
        Ext_Out_23_fire     : in  std_logic;
        Ext_Out_24_fire     : in  std_logic;
        Ext_Out_25_fire     : in  std_logic;
        Ext_Out_26_fire     : in  std_logic;
        Ext_Out_27_fire     : in  std_logic;
        Ext_Out_28_fire     : in  std_logic;
        Ext_Out_29_fire     : in  std_logic;
        Ext_Out_30_fire     : in  std_logic;
        Ext_Out_31_fire     : in  std_logic
    );
end prog_fsm;
```

```
architecture Behavioral of prog_fsm is

    COMPONENT c_counter_binary_0
      PORT (
        CLK : IN STD_LOGIC;
        CE : IN STD_LOGIC;
        SCLR : IN STD_LOGIC;
        Q : OUT STD_LOGIC_VECTOR(63 DOWNTO 0)
      );
    END COMPONENT;

    COMPONENT c_counter_binary_2
        PORT (
            CLK : IN STD_LOGIC;
            CE : IN STD_LOGIC;
            SCLR : IN STD_LOGIC;
            Q : OUT STD_LOGIC_VECTOR(3 DOWNTO 0)
        );
    END COMPONENT;

    type state_type is (S0, S1, S2, S3, S4, S5, S6, S7, S8);
    signal s_curr : state_type := S0;
    type fsm2_type is (S0, S1);
    signal fsm2 : fsm2_type := S0;
    type fsm3_type is (S0, S1, S2);
    signal fsm3 : fsm3_type := S0;
    type fsm4_type is (S0, S1);
    signal fsm4 : fsm4_type := S0;
    signal counter : integer range 0 to 7 := 0;
    signal strobe_dummy : std_logic := '0';
    signal global_ce_dummy : std_logic := '0';
    signal runCounterEnable : std_logic := '0';
    signal runCounterEnable_dummy1 : std_logic := '0';
    signal runCounterEnable_dummy2 : std_logic := '0';
    signal runCounterClear : std_logic := '0';
    signal runCounterClear_dummy1 : std_logic := '0';
    signal runCounterClear_dummy2 : std_logic := '0';
    signal runCounterOut : std_logic_vector (63 downto 0);
    signal runNumCycles : std_logic_vector (63 downto 0);
    signal runCounterLoad : std_logic := '0';
    signal runCounterLoadValue : std_logic_vector (63 downto 0) := (others => '0');
    signal smallCounterOut : std_logic_vector (3 downto 0);
    signal smallCounterEnable : std_logic := '0';
    signal smallCounterClear : std_logic := '1';
    signal global_ce_dummy2 : std_logic := '0';
    signal strobeBegin : std_logic := '0';
    signal runStop : std_logic := '0';

    signal counterControlEnable : std_logic := '0';
    signal counterControlStarted : std_logic := '0';
    signal reset_sig : std_logic := '0';
    signal hasFired : std_logic := '0';

    signal xilly_eof_dummy : std_logic := '0';
    signal xilly_eof_dummy2 : std_logic := '0';

    signal hasShifted : std_logic := '0';
    signal Xilly_EOF : std_logic := '0';
```

```
signal Ext_In_00_data_temp : std_logic_vector (7 downto 0) := "00000000";
signal Ext_In_01_data_temp : std_logic_vector (7 downto 0) := "00000000";
signal Ext_In_02_data_temp : std_logic_vector (7 downto 0) := "00000000";
signal Ext_In_03_data_temp : std_logic_vector (7 downto 0) := "00000000";
signal Ext_In_04_data_temp : std_logic_vector (7 downto 0) := "00000000";
signal Ext_In_05_data_temp : std_logic_vector (7 downto 0) := "00000000";
signal Ext_In_06_data_temp : std_logic_vector (7 downto 0) := "00000000";
signal Ext_In_07_data_temp : std_logic_vector (7 downto 0) := "00000000";
signal Ext_In_08_data_temp : std_logic_vector (7 downto 0) := "00000000";
signal Ext_In_09_data_temp : std_logic_vector (7 downto 0) := "00000000";
signal Ext_In_10_data_temp : std_logic_vector (7 downto 0) := "00000000";
signal Ext_In_11_data_temp : std_logic_vector (7 downto 0) := "00000000";
signal Ext_In_12_data_temp : std_logic_vector (7 downto 0) := "00000000";
signal Ext_In_13_data_temp : std_logic_vector (7 downto 0) := "00000000";
signal Ext_In_14_data_temp : std_logic_vector (7 downto 0) := "00000000";
signal Ext_In_15_data_temp : std_logic_vector (7 downto 0) := "00000000";
signal Ext_In_16_data_temp : std_logic_vector (7 downto 0) := "00000000";
signal Ext_In_17_data_temp : std_logic_vector (7 downto 0) := "00000000";
signal Ext_In_18_data_temp : std_logic_vector (7 downto 0) := "00000000";
signal Ext_In_19_data_temp : std_logic_vector (7 downto 0) := "00000000";
signal Ext_In_20_data_temp : std_logic_vector (7 downto 0) := "00000000";
signal Ext_In_21_data_temp : std_logic_vector (7 downto 0) := "00000000";
signal Ext_In_22_data_temp : std_logic_vector (7 downto 0) := "00000000";
signal Ext_In_23_data_temp : std_logic_vector (7 downto 0) := "00000000";
signal Ext_In_24_data_temp : std_logic_vector (7 downto 0) := "00000000";
signal Ext_In_25_data_temp : std_logic_vector (7 downto 0) := "00000000";
signal Ext_In_26_data_temp : std_logic_vector (7 downto 0) := "00000000";
signal Ext_In_27_data_temp : std_logic_vector (7 downto 0) := "00000000";
signal Ext_In_28_data_temp : std_logic_vector (7 downto 0) := "00000000";
signal Ext_In_29_data_temp : std_logic_vector (7 downto 0) := "00000000";
signal Ext_In_30_data_temp : std_logic_vector (7 downto 0) := "00000000";
signal Ext_In_31_data_temp : std_logic_vector (7 downto 0) := "00000000";
signal Ext_In_00_fire_temp : std_logic := '0';
signal Ext_In_01_fire_temp : std_logic := '0';
signal Ext_In_02_fire_temp : std_logic := '0';
signal Ext_In_03_fire_temp : std_logic := '0';
signal Ext_In_04_fire_temp : std_logic := '0';
signal Ext_In_05_fire_temp : std_logic := '0';
signal Ext_In_06_fire_temp : std_logic := '0';
signal Ext_In_07_fire_temp : std_logic := '0';
signal Ext_In_08_fire_temp : std_logic := '0';
signal Ext_In_09_fire_temp : std_logic := '0';
signal Ext_In_10_fire_temp : std_logic := '0';
signal Ext_In_11_fire_temp : std_logic := '0';
signal Ext_In_12_fire_temp : std_logic := '0';
signal Ext_In_13_fire_temp : std_logic := '0';
signal Ext_In_14_fire_temp : std_logic := '0';
signal Ext_In_15_fire_temp : std_logic := '0';
signal Ext_In_16_fire_temp : std_logic := '0';
signal Ext_In_17_fire_temp : std_logic := '0';
signal Ext_In_18_fire_temp : std_logic := '0';
signal Ext_In_19_fire_temp : std_logic := '0';
signal Ext_In_20_fire_temp : std_logic := '0';
signal Ext_In_21_fire_temp : std_logic := '0';
signal Ext_In_22_fire_temp : std_logic := '0';
signal Ext_In_23_fire_temp : std_logic := '0';
signal Ext_In_24_fire_temp : std_logic := '0';
signal Ext_In_25_fire_temp : std_logic := '0';
signal Ext_In_26_fire_temp : std_logic := '0';
```

```
    signal Ext_In_27_fire_temp : std_logic := '0';
    signal Ext_In_28_fire_temp : std_logic := '0';
    signal Ext_In_29_fire_temp : std_logic := '0';
    signal Ext_In_30_fire_temp : std_logic := '0';
    signal Ext_In_31_fire_temp : std_logic := '0';
    signal SR_output_latched : std_logic_vector (127 downto 0) := X"
        00000000000000000000000000000000";

    signal runCounterSum : std_logic_vector (63 downto 0) := X"0000000000000000";
    signal addValue : std_logic_vector (63 downto 0) := X"0000000000000000";

    signal hasFired2 : std_logic := '0';
    signal runCounterCE : std_logic;

    constant Config_ID : std_logic_vector (15 downto 0) := X"4321";

begin

    Strobe <= strobe_dummy;
    Global_CE <= global_ce_dummy or global_ce_dummy2;
    runCounterEnable <= runCounterEnable_dummy1 or runCounterEnable_dummy2;
    runCounterClear <= reset_sig;
    Reset <= reset_sig;
    Xilly_EOF <= xilly_eof_dummy or xilly_eof_dummy2;
    runCounterCE <= runCounterEnable or runCounterClear;

    -- 64 bit counter for number of cycles running
    runCounter : c_counter_binary_0
    PORT MAP (
        CLK => Global_Net_Clk_On,
        CE => runCounterCE,
        SCLR => runCounterClear,
        Q => runCounterOut
    );

    -- 3 bit counter to know wher in the GNC cycle it is
    smallCounter : c_counter_binary_2
    PORT MAP (
        CLK => Acquire_Fire_Clk_On,
        CE => smallCounterEnable,
        SCLR => smallCounterClear,
        Q => smallCounterOut
    );


    -- process which controls the counter for run N cycles
    process(Acquire_Fire_Clk_On, Global_Net_Clk_On)
    begin
        if(rising_edge(Acquire_Fire_Clk_On)) then
            case fsm3 is
                -- not started, looking for start command
                when S0 =>
                    if(counterControlEnable = '1') then
                        fsm3 <= S1;
                        runCounterEnable_dummy2 <= '1';
                        global_ce_dummy2 <= '1';
                    else
                        fsm3 <= S0;
                    end if;
```

```
            -- started, check if finished
            when S1 =>
                if((runCounterOut = runNumCycles) or (runStop = '1')) then
                    fsm3 <= S2;
                    runCounterEnable_dummy2 <= '0';
                    global_ce_dummy2 <= '0';
                    xilly_eof_dummy2 <= '1';
                else
                    fsm3 <= S1;
                end if;
            when S2 =>
                if(smallCounterOut = "0000") then
                    xilly_eof_dummy2 <= '0';
                    fsm3 <= S0;
                end if;
        end case;
    end if;
end process;

process(Global_Net_Clk_On)
begin
    if(rising_edge(Global_Net_Clk_On)) then
        SR_output_latched <= SR_output;
    end if;
end process;

-- External Outputs
-- Look into if it should be clocked on the CE'd GNC
-- Updated 9/11/14: Only output fires when clock is running
process(Global_Net_Clk_On)
begin
    if(rising_edge(Global_Net_Clk_On)) then
        if(global_ce_dummy = '1' or global_ce_dummy2 = '1' or hasShifted = '1' or Xilly_EOF =
            '1') then
            if( ( Ext_Out_00_fire = '1' ) or ( Ext_Out_01_fire = '1' ) or ( Ext_Out_02_fire =
                '1' ) or ( Ext_Out_03_fire = '1' ) or ( Ext_Out_04_fire = '1' ) or (
                Ext_Out_05_fire = '1' ) or ( Ext_Out_06_fire = '1' ) or ( Ext_Out_07_fire =
                '1' ) or ( Ext_Out_08_fire = '1' ) or ( Ext_Out_09_fire = '1' ) or (
                Ext_Out_10_fire = '1' ) or ( Ext_Out_11_fire = '1' ) or ( Ext_Out_12_fire =
                '1' ) or ( Ext_Out_13_fire = '1' ) or ( Ext_Out_14_fire = '1' ) or (
                Ext_Out_15_fire = '1' ) or ( Ext_Out_16_fire = '1' ) or ( Ext_Out_17_fire =
                '1' ) or ( Ext_Out_18_fire = '1' ) or ( Ext_Out_19_fire = '1' ) or (
                Ext_Out_20_fire = '1' ) or ( Ext_Out_21_fire = '1' ) or ( Ext_Out_22_fire =
                '1' ) or ( Ext_Out_22_fire = '1' ) or ( Ext_Out_23_fire = '1' ) or (
                Ext_Out_24_fire = '1' ) or ( Ext_Out_25_fire = '1' ) or ( Ext_Out_26_fire =
                '1' ) or ( Ext_Out_27_fire = '1' ) or ( Ext_Out_28_fire = '1' ) or (
                Ext_Out_29_fire = '1' ) or ( Ext_Out_30_fire = '1' ) or ( Ext_Out_31_fire =
                '1' ) or ( hasShifted = '1' ) or ( Xilly_EOF = '1' ) ) then
                Ext_Data_Present <= '1';
                External_Data (511 downto 504) <= runCounterOut(7 downto 0);
                External_Data (503 downto 496) <= runCounterOut(15 downto 8);
                External_Data (495 downto 488) <= runCounterOut(23 downto 16);
                External_Data (487 downto 480) <= runCounterOut(31 downto 24);
                External_Data (479 downto 472) <= runCounterOut(39 downto 32);
                External_Data (471 downto 464) <= runCounterOut(47 downto 40);
                External_Data (463 downto 456) <= runCounterOut(55 downto 48);
                External_Data (455 downto 448) <= runCounterOut(63 downto 56);
                External_Data (447 downto 440) <= Ext_Out_00_data;
                External_Data (439 downto 432) <= Ext_Out_01_data;
```

```
            External_Data (431 downto 424) <= Ext_Out_02_data;
            External_Data (423 downto 416) <= Ext_Out_03_data;
            External_Data (415 downto 408) <= Ext_Out_04_data;
            External_Data (407 downto 400) <= Ext_Out_05_data;
            External_Data (399 downto 392) <= Ext_Out_06_data;
            External_Data (391 downto 384) <= Ext_Out_07_data;
            External_Data (383 downto 376) <= Ext_Out_08_data;
            External_Data (375 downto 368) <= Ext_Out_09_data;
            External_Data (367 downto 360) <= Ext_Out_10_data;
            External_Data (359 downto 352) <= Ext_Out_11_data;
            External_Data (351 downto 344) <= Ext_Out_12_data;
            External_Data (343 downto 336) <= Ext_Out_13_data;
            External_Data (335 downto 328) <= Ext_Out_14_data;
            External_Data (327 downto 320) <= Ext_Out_15_data;
            External_Data (319 downto 312) <= Ext_Out_16_data;
            External_Data (311 downto 304) <= Ext_Out_17_data;
            External_Data (303 downto 296) <= Ext_Out_18_data;
            External_Data (295 downto 288) <= Ext_Out_19_data;
            External_Data (287 downto 280) <= Ext_Out_20_data;
            External_Data (279 downto 272) <= Ext_Out_21_data;
            External_Data (271 downto 264) <= Ext_Out_22_data;
            External_Data (263 downto 256) <= Ext_Out_23_data;
            External_Data (255 downto 248) <= Ext_Out_24_data;
            External_Data (247 downto 240) <= Ext_Out_25_data;
            External_Data (239 downto 232) <= Ext_Out_26_data;
            External_Data (231 downto 224) <= Ext_Out_27_data;
            External_Data (223 downto 216) <= Ext_Out_28_data;
            External_Data (215 downto 208) <= Ext_Out_29_data;
            External_Data (207 downto 200) <= Ext_Out_30_data;
            External_Data (199 downto 192) <= Ext_Out_31_data;
            External_Data (191 downto 160) <= (others => '0');
            External_Data (159 downto 152) <= SR_output_latched(7 downto 0);
            External_Data (151 downto 144) <= SR_output_latched(15 downto 8);
            External_Data (143 downto 136) <= SR_output_latched(23 downto 16);
            External_Data (135 downto 128) <= SR_output_latched(31 downto 24);
            External_Data (127 downto 120) <= SR_output_latched(39 downto 32);
            External_Data (119 downto 112) <= SR_output_latched(47 downto 40);
            External_Data (111 downto 104) <= SR_output_latched(55 downto 48);
            External_Data (103 downto 96) <= SR_output_latched(63 downto 56);
            External_Data (95 downto 88) <= SR_output_latched(71 downto 64);
            External_Data (87 downto 80) <= SR_output_latched(79 downto 72);
            External_Data (79 downto 72) <= SR_output_latched(87 downto 80);
            External_Data (71 downto 64) <= SR_output_latched(95 downto 88);
            External_Data (63 downto 56) <= SR_output_latched(103 downto 96);
            External_Data (55 downto 48) <= SR_output_latched(111 downto 104);
            External_Data (47 downto 40) <= SR_output_latched(119 downto 112);
            External_Data (39 downto 32) <= SR_output_latched(127 downto 120);
            External_Data (31 downto 18) <= (others => '0');
            External_Data (17) <= hasShifted;
            External_Data (16) <= Xilly_EOF;
            External_Data (15 downto 0) <= (others => '0');
        else
            Ext_Data_Present <= '0';
            External_Data <= (others => '0');
        end if;
    else
        Ext_Data_Present <= '0';
        External_Data <= (others => '0');
    end if;
```

```
    end if;
end process;

-- process which starts the 3 bit counter
process(Global_Net_Clk_On)
begin
    if(rising_edge(Global_Net_Clk_On)) then
        case fsm2 is
            when S0 =>
                smallCounterEnable <= '1';
                smallCounterClear <= '0';
                fsm2 <= S1;
            when S1 =>
                fsm2 <= S0;
        end case;
    end if;
end process;

addValue (7 downto 0) <= Data (279 downto 272);
addValue (15 downto 8) <= Data (271 downto 264);
addValue (23 downto 16) <= Data (263 downto 256);
addValue (31 downto 24) <= Data (255 downto 248);
addValue (39 downto 32) <= Data (247 downto 240);
addValue (47 downto 40) <= Data (239 downto 232);
addValue (55 downto 48) <= Data (231 downto 224);
addValue (63 downto 56) <= Data (223 downto 216);

process(Acquire_Fire_Clk_On)
begin
    if(rising_edge(Acquire_Fire_Clk_On)) then
        runCounterSum <= std_logic_vector(unsigned(runCounterOut) + unsigned(addValue));
    end if;
end process;

-- process which controls everything else
process(Global_Net_Clk_On, Data, Acquire_Fire_Clk_On)
begin
    if(rising_edge(Acquire_Fire_Clk_On)) then
        case s_curr is
            when S0 =>
                if(smallCounterOut = "0000") then
                    -- if signal is high, set certain signals to low
                    if(hasFired = '1' or hasShifted = '1') then
                        Element_Addr <= (others => '1');
                        strobe_dummy <= '0';
                        runCounterClear_dummy1 <= '0';
                        Capture <= '0';
                        Shift <= '0';
                        reset_sig <= '0';
                        xilly_eof_dummy <= '0';
                        hasShifted <= '0';
                        runStop <= '0';
                        hasFired <= '0';
                    end if;

                    -- load command
                    if(Data(287 downto 280) = "00000001") then
                        s_curr <= S1;
                        -- Look into if this is correct
```

```
                Element_Addr (15 downto 8) <= Data(271 downto 264);
                Element_Addr (7 downto 0) <= Data(279 downto 272);
                Prog_Data <= Data (247 downto 216);

            -- Halt command
            elsif(Data(287 downto 280) = "00000010") then
                s_curr <= S2;

            -- Run command
            elsif(Data(287 downto 280) = "00000100") then
                s_curr <= S3;

            -- Run N cycles command
            elsif(Data(287 downto 280) = "00001000") then
                s_curr <= S4;

            -- RESET COMMAND
            elsif(Data(287 downto 280) = "00100000") then
                s_curr <= S6;
            -- CAPTURE COMMAND
            elsif(Data(287 downto 280) = "01000000") then
                s_curr <= S7;
            -- SHIFT COMMAND
            elsif(Data(287 downto 280) = "10000000") then
                s_curr <= S8;
            else
                s_curr <= S0;
            end if;
        else
            s_curr <= S0;
        end if;

    -- Load command
    when S1 =>
        if(smallCounterOut = "0001") then
            strobe_dummy <= '1';
            s_curr <= S1;
        elsif(smallCounterOut = "0111") then
            strobe_dummy <= '0';
            Prog_Data(31 downto 16) <= (others => '0');
            Prog_Data(15 downto 8) <=  Data(207 downto 200);
            Prog_Data(7 downto 0) <=  Data(215 downto 208);
            s_curr <= S1;
        elsif(smallCounterOut = "1000") then
            strobe_dummy <= '1';
            hasFired <= '1';
            s_curr <= S0;
        end if;

    -- Halt command (halt at the next positive edge of global_net_clk)
    when S2 =>
        if(smallCounterOut = "0011") then
            runStop <= '1';
            global_ce_dummy <= '0';
            runCounterEnable_dummy1 <= '0';
            runCounterClear_dummy1 <= '1';
            hasFired <= '1';
            xilly_eof_dummy <= '1';
            s_curr <= S0;
```

```
        end if;

    -- Run command (run at the next positive edge of global_net_clk)
    when S3 =>
        if(smallCounterOut = "0011") then
            global_ce_dummy <= '1';
            runCounterEnable_dummy1 <= '1';
            s_curr <= S0;
        end if;

    -- Run N cycles command
    when S4 =>
        if(smallCounterOut = "0010") then
            runNumCycles <= runCounterSum;
            s_curr <= S4;
        elsif(smallCounterOut = "0011") then
            counterControlEnable <= '1';
            s_curr <= S4;
        elsif(smallCounterOut = "0101") then
            counterControlEnable <= '0';
            s_curr <= S0;
        end if;

    -- Fire command
    when S5 =>
        if(smallCounterOut = "0110") then
            hasFired <= '1';
            s_curr <= S0;
        else
            s_curr <= S5;
        end if;

    -- Reset Command
    when S6 =>
        if(smallCounterOut = "0001") then
            reset_sig <= '1';
            hasFired <= '1';
            s_curr <= S0;
        else
            s_curr <= S6;
        end if;

    -- Capture Command
    when S7 =>
        if(smallCounterOut = "0001") then
            Capture <= '1';
            s_curr <= S7;
        elsif(smallCounterOut = "0011") then
            strobe_dummy <= '1';
            hasFired <= '1';
            s_curr <= S0;
        else
            s_curr <= S7;
        end if;

    -- Shift Command
    when S8 =>
        if(smallCounterOut = "0001") then
            Shift <= '1';
```

```
                        s_curr <= S8;
                    elsif(smallCounterOut = "0011") then
                        strobe_dummy <= '1';
--                          hasFired <= '1';
                        hasShifted <= '1';
                        s_curr <= S0;
                    else
                        s_curr <= S8;
                    end if;
            end case;
        end if;
    end process;

    process(Acquire_Fire_Clk)
    begin
        if(falling_edge(Acquire_Fire_Clk)) then
            case fsm4 is
                when S0 =>
                    if(smallCounterOut = "1111") then
                        if(hasFired2 = '1') then
                            Ext_In_00_data_temp <= (others => '0');
                            Ext_In_01_data_temp <= (others => '0');
                            Ext_In_02_data_temp <= (others => '0');
                            Ext_In_03_data_temp <= (others => '0');
                            Ext_In_04_data_temp <= (others => '0');
                            Ext_In_05_data_temp <= (others => '0');
                            Ext_In_06_data_temp <= (others => '0');
                            Ext_In_07_data_temp <= (others => '0');
                            Ext_In_08_data_temp <= (others => '0');
                            Ext_In_09_data_temp <= (others => '0');
                            Ext_In_10_data_temp <= (others => '0');
                            Ext_In_11_data_temp <= (others => '0');
                            Ext_In_12_data_temp <= (others => '0');
                            Ext_In_13_data_temp <= (others => '0');
                            Ext_In_14_data_temp <= (others => '0');
                            Ext_In_15_data_temp <= (others => '0');
                            Ext_In_16_data_temp <= (others => '0');
                            Ext_In_17_data_temp <= (others => '0');
                            Ext_In_18_data_temp <= (others => '0');
                            Ext_In_19_data_temp <= (others => '0');
                            Ext_In_20_data_temp <= (others => '0');
                            Ext_In_21_data_temp <= (others => '0');
                            Ext_In_22_data_temp <= (others => '0');
                            Ext_In_23_data_temp <= (others => '0');
                            Ext_In_24_data_temp <= (others => '0');
                            Ext_In_25_data_temp <= (others => '0');
                            Ext_In_26_data_temp <= (others => '0');
                            Ext_In_27_data_temp <= (others => '0');
                            Ext_In_28_data_temp <= (others => '0');
                            Ext_In_29_data_temp <= (others => '0');
                            Ext_In_30_data_temp <= (others => '0');
                            Ext_In_31_data_temp <= (others => '0');
                            Ext_In_00_fire_temp <= '0';
                            Ext_In_01_fire_temp <= '0';
                            Ext_In_02_fire_temp <= '0';
                            Ext_In_03_fire_temp <= '0';
                            Ext_In_04_fire_temp <= '0';
                            Ext_In_05_fire_temp <= '0';
                            Ext_In_06_fire_temp <= '0';
```

```
        Ext_In_07_fire_temp <= '0';
        Ext_In_08_fire_temp <= '0';
        Ext_In_09_fire_temp <= '0';
        Ext_In_10_fire_temp <= '0';
        Ext_In_11_fire_temp <= '0';
        Ext_In_12_fire_temp <= '0';
        Ext_In_13_fire_temp <= '0';
        Ext_In_14_fire_temp <= '0';
        Ext_In_15_fire_temp <= '0';
        Ext_In_16_fire_temp <= '0';
        Ext_In_17_fire_temp <= '0';
        Ext_In_18_fire_temp <= '0';
        Ext_In_19_fire_temp <= '0';
        Ext_In_20_fire_temp <= '0';
        Ext_In_21_fire_temp <= '0';
        Ext_In_22_fire_temp <= '0';
        Ext_In_23_fire_temp <= '0';
        Ext_In_24_fire_temp <= '0';
        Ext_In_25_fire_temp <= '0';
        Ext_In_26_fire_temp <= '0';
        Ext_In_27_fire_temp <= '0';
        Ext_In_28_fire_temp <= '0';
        Ext_In_29_fire_temp <= '0';
        Ext_In_30_fire_temp <= '0';
        Ext_In_31_fire_temp <= '0';
        hasFired2 <= '0';
    end if;

    if(Data(287 downto 280) = "00010000") then
        Ext_In_00_data_temp <= Data(279 downto 272);
        Ext_In_01_data_temp <= Data(271 downto 264);
        Ext_In_02_data_temp <= Data(263 downto 256);
        Ext_In_03_data_temp <= Data(255 downto 248);
        Ext_In_04_data_temp <= Data(247 downto 240);
        Ext_In_05_data_temp <= Data(239 downto 232);
        Ext_In_06_data_temp <= Data(231 downto 224);
        Ext_In_07_data_temp <= Data(223 downto 216);
        Ext_In_08_data_temp <= Data(215 downto 208);
        Ext_In_09_data_temp <= Data(207 downto 200);
        Ext_In_10_data_temp <= Data(199 downto 192);
        Ext_In_11_data_temp <= Data(191 downto 184);
        Ext_In_12_data_temp <= Data(183 downto 176);
        Ext_In_13_data_temp <= Data(175 downto 168);
        Ext_In_14_data_temp <= Data(167 downto 160);
        Ext_In_15_data_temp <= Data(159 downto 152);
        Ext_In_16_data_temp <= Data(151 downto 144);
        Ext_In_17_data_temp <= Data(143 downto 136);
        Ext_In_18_data_temp <= Data(135 downto 128);
        Ext_In_19_data_temp <= Data(127 downto 120);
        Ext_In_20_data_temp <= Data(119 downto 112);
        Ext_In_21_data_temp <= Data(111 downto 104);
        Ext_In_22_data_temp <= Data(103 downto 96);
        Ext_In_23_data_temp <= Data(95 downto 88);
        Ext_In_24_data_temp <= Data(87 downto 80);
        Ext_In_25_data_temp <= Data(79 downto 72);
        Ext_In_26_data_temp <= Data(71 downto 64);
        Ext_In_27_data_temp <= Data(63 downto 56);
        Ext_In_28_data_temp <= Data(55 downto 48);
        Ext_In_29_data_temp <= Data(47 downto 40);
```

```
Ext_In_30_data_temp <= Data(39 downto 32);
Ext_In_31_data_temp <= Data(31 downto 24);
if( Data(279 downto 272) /= "00000000" ) then
    Ext_In_00_fire_temp <= '1';
else
    Ext_In_00_fire_temp <= '0';
end if;

if( Data(271 downto 264) /= "00000000" ) then
    Ext_In_01_fire_temp <= '1';
else
    Ext_In_01_fire_temp <= '0';
end if;

if( Data(263 downto 256) /= "00000000" ) then
    Ext_In_02_fire_temp <= '1';
else
    Ext_In_02_fire_temp <= '0';
end if;

if( Data(255 downto 248) /= "00000000" ) then
    Ext_In_03_fire_temp <= '1';
else
    Ext_In_03_fire_temp <= '0';
end if;

if( Data(247 downto 240) /= "00000000" ) then
    Ext_In_04_fire_temp <= '1';
else
    Ext_In_04_fire_temp <= '0';
end if;

if( Data(239 downto 232) /= "00000000" ) then
    Ext_In_05_fire_temp <= '1';
else
    Ext_In_05_fire_temp <= '0';
end if;

if( Data(231 downto 224) /= "00000000" ) then
    Ext_In_06_fire_temp <= '1';
else
    Ext_In_06_fire_temp <= '0';
end if;

if( Data(223 downto 216) /= "00000000" ) then
    Ext_In_07_fire_temp <= '1';
else
    Ext_In_07_fire_temp <= '0';
end if;

if( Data(215 downto 208) /= "00000000" ) then
    Ext_In_08_fire_temp <= '1';
else
    Ext_In_08_fire_temp <= '0';
end if;

if( Data(207 downto 200) /= "00000000" ) then
    Ext_In_09_fire_temp <= '1';
else
```

```
    Ext_In_09_fire_temp <= '0';
end if;

if( Data(199 downto 192) /= "00000000" ) then
    Ext_In_10_fire_temp <= '1';
else
    Ext_In_10_fire_temp <= '0';
end if;

if( Data(191 downto 184) /= "00000000" ) then
    Ext_In_11_fire_temp <= '1';
else
    Ext_In_11_fire_temp <= '0';
end if;

if( Data(183 downto 176) /= "00000000" ) then
    Ext_In_12_fire_temp <= '1';
else
    Ext_In_12_fire_temp <= '0';
end if;

if( Data(175 downto 168) /= "00000000" ) then
    Ext_In_13_fire_temp <= '1';
else
    Ext_In_13_fire_temp <= '0';
end if;

if( Data(167 downto 160) /= "00000000" ) then
    Ext_In_14_fire_temp <= '1';
else
    Ext_In_14_fire_temp <= '0';
end if;

if( Data(159 downto 152) /= "00000000" ) then
    Ext_In_15_fire_temp <= '1';
else
    Ext_In_15_fire_temp <= '0';
end if;

if( Data(151 downto 144) /= "00000000" ) then
    Ext_In_16_fire_temp <= '1';
else
    Ext_In_16_fire_temp <= '0';
end if;

if( Data(143 downto 136) /= "00000000" ) then
    Ext_In_17_fire_temp <= '1';
else
    Ext_In_17_fire_temp <= '0';
end if;

if( Data(135 downto 128) /= "00000000" ) then
    Ext_In_18_fire_temp <= '1';
else
    Ext_In_18_fire_temp <= '0';
end if;

if( Data(127 downto 120) /= "00000000" ) then
    Ext_In_19_fire_temp <= '1';
```

```
else
    Ext_In_19_fire_temp <= '0';
end if;

if( Data(119 downto 112) /= "00000000" ) then
    Ext_In_20_fire_temp <= '1';
else
    Ext_In_20_fire_temp <= '0';
end if;

if( Data(111 downto 104) /= "00000000" ) then
    Ext_In_21_fire_temp <= '1';
else
    Ext_In_21_fire_temp <= '0';
end if;

if( Data(103 downto 96) /= "00000000" ) then
    Ext_In_22_fire_temp <= '1';
else
    Ext_In_22_fire_temp <= '0';
end if;

if( Data(95 downto 88) /= "00000000" ) then
    Ext_In_23_fire_temp <= '1';
else
    Ext_In_23_fire_temp <= '0';
end if;

if( Data(87 downto 80) /= "00000000" ) then
    Ext_In_24_fire_temp <= '1';
else
    Ext_In_24_fire_temp <= '0';
end if;

if( Data(79 downto 72) /= "00000000" ) then
    Ext_In_25_fire_temp <= '1';
else
    Ext_In_25_fire_temp <= '0';
end if;

if( Data(71 downto 64) /= "00000000" ) then
    Ext_In_26_fire_temp <= '1';
else
    Ext_In_26_fire_temp <= '0';
end if;

if( Data(63 downto 56) /= "00000000" ) then
    Ext_In_27_fire_temp <= '1';
else
    Ext_In_27_fire_temp <= '0';
end if;

if( Data(55 downto 48) /= "00000000" ) then
    Ext_In_28_fire_temp <= '1';
else
    Ext_In_28_fire_temp <= '0';
end if;

if( Data(47 downto 40) /= "00000000" ) then
```

```
                        Ext_In_29_fire_temp <= '1';
                    else
                        Ext_In_29_fire_temp <= '0';
                    end if;

                    if( Data(39 downto 32) /= "00000000" ) then
                        Ext_In_30_fire_temp <= '1';
                    else
                        Ext_In_30_fire_temp <= '0';
                    end if;

                    if( Data(31 downto 24) /= "00000000" ) then
                        Ext_In_31_fire_temp <= '1';
                    else
                        Ext_In_31_fire_temp <= '0';
                    end if;

                    fsm4 <= S1;
                end if;
            else
                fsm4 <= S0;
            end if;
        when S1 =>
            if(smallCounterOut = "0110") then
                hasFired2 <= '1';
                fsm4 <= S0;
            else
                fsm4 <= S1;
            end if;
    end case;
  end if;
end process;

Ext_In_latch : process (Acquire_Fire_Clk)
begin
  if(rising_edge(Acquire_Fire_Clk)) then
    Ext_In_00_data <= Ext_In_00_data_temp;
    Ext_In_01_data <= Ext_In_01_data_temp;
    Ext_In_02_data <= Ext_In_02_data_temp;
    Ext_In_03_data <= Ext_In_03_data_temp;
    Ext_In_04_data <= Ext_In_04_data_temp;
    Ext_In_05_data <= Ext_In_05_data_temp;
    Ext_In_06_data <= Ext_In_06_data_temp;
    Ext_In_07_data <= Ext_In_07_data_temp;
    Ext_In_08_data <= Ext_In_08_data_temp;
    Ext_In_09_data <= Ext_In_09_data_temp;
    Ext_In_10_data <= Ext_In_10_data_temp;
    Ext_In_11_data <= Ext_In_11_data_temp;
    Ext_In_12_data <= Ext_In_12_data_temp;
    Ext_In_13_data <= Ext_In_13_data_temp;
    Ext_In_14_data <= Ext_In_14_data_temp;
    Ext_In_15_data <= Ext_In_15_data_temp;
    Ext_In_16_data <= Ext_In_16_data_temp;
    Ext_In_17_data <= Ext_In_17_data_temp;
    Ext_In_18_data <= Ext_In_18_data_temp;
    Ext_In_19_data <= Ext_In_19_data_temp;
    Ext_In_20_data <= Ext_In_20_data_temp;
    Ext_In_21_data <= Ext_In_21_data_temp;
    Ext_In_22_data <= Ext_In_22_data_temp;
```

```
            Ext_In_23_data <= Ext_In_23_data_temp;
            Ext_In_24_data <= Ext_In_24_data_temp;
            Ext_In_25_data <= Ext_In_25_data_temp;
            Ext_In_26_data <= Ext_In_26_data_temp;
            Ext_In_27_data <= Ext_In_27_data_temp;
            Ext_In_28_data <= Ext_In_28_data_temp;
            Ext_In_29_data <= Ext_In_29_data_temp;
            Ext_In_30_data <= Ext_In_30_data_temp;
            Ext_In_31_data <= Ext_In_31_data_temp;
            Ext_In_00_fire <= Ext_In_00_fire_temp;
            Ext_In_01_fire <= Ext_In_01_fire_temp;
            Ext_In_02_fire <= Ext_In_02_fire_temp;
            Ext_In_03_fire <= Ext_In_03_fire_temp;
            Ext_In_04_fire <= Ext_In_04_fire_temp;
            Ext_In_05_fire <= Ext_In_05_fire_temp;
            Ext_In_06_fire <= Ext_In_06_fire_temp;
            Ext_In_07_fire <= Ext_In_07_fire_temp;
            Ext_In_08_fire <= Ext_In_08_fire_temp;
            Ext_In_09_fire <= Ext_In_09_fire_temp;
            Ext_In_10_fire <= Ext_In_10_fire_temp;
            Ext_In_11_fire <= Ext_In_11_fire_temp;
            Ext_In_12_fire <= Ext_In_12_fire_temp;
            Ext_In_13_fire <= Ext_In_13_fire_temp;
            Ext_In_14_fire <= Ext_In_14_fire_temp;
            Ext_In_15_fire <= Ext_In_15_fire_temp;
            Ext_In_16_fire <= Ext_In_16_fire_temp;
            Ext_In_17_fire <= Ext_In_17_fire_temp;
            Ext_In_18_fire <= Ext_In_18_fire_temp;
            Ext_In_19_fire <= Ext_In_19_fire_temp;
            Ext_In_20_fire <= Ext_In_20_fire_temp;
            Ext_In_21_fire <= Ext_In_21_fire_temp;
            Ext_In_22_fire <= Ext_In_22_fire_temp;
            Ext_In_23_fire <= Ext_In_23_fire_temp;
            Ext_In_24_fire <= Ext_In_24_fire_temp;
            Ext_In_25_fire <= Ext_In_25_fire_temp;
            Ext_In_26_fire <= Ext_In_26_fire_temp;
            Ext_In_27_fire <= Ext_In_27_fire_temp;
            Ext_In_28_fire <= Ext_In_28_fire_temp;
            Ext_In_29_fire <= Ext_In_29_fire_temp;
            Ext_In_30_fire <= Ext_In_30_fire_temp;
            Ext_In_31_fire <= Ext_In_31_fire_temp;
        end if;
    end process;

end Behavioral;
```

Appendix B

Element RTL Schematic

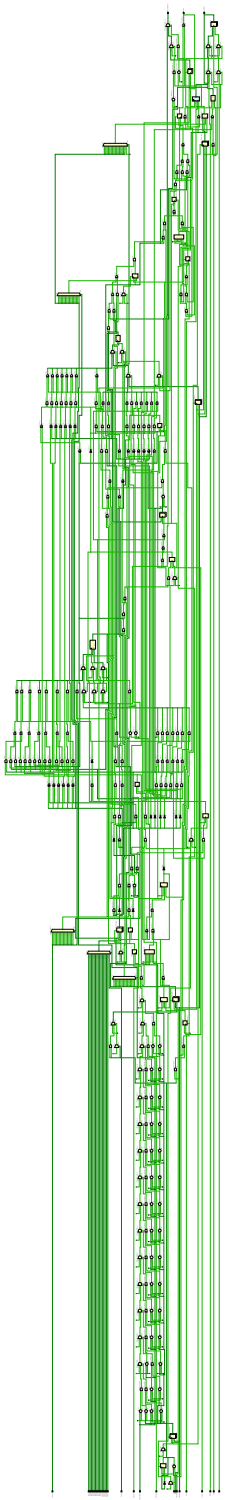


Figure B.1: RTL Schematic of a single element

Vita

Christopher Daffron is from Franklin, TN. He graduated from Centennial High School in 2010 and then began his studies at the University of Tennessee, Knoxville, pursuing a Bachelor of Science degree in Computer Engineering from the Department of Electrical Engineering and Computer Science in the College of Engineering. He graduated with this degree in May of 2014. The following summer, he completed an internship as an Aviation Software Engineer at Garmin International in Olathe, KS. In August of 2014, he began pursuing a Master of Science degree in Computer Engineering at the University of Tennessee. He graduated with that degree in August of 2015 and then begin working for Intel Corporation.