Towards FPGA hardware in the loop for QCA simulation

Alan Olson

Towards FPGA Hardware in the Loop for QCA Simulation

by

Alan Olson

A Thesis Submitted in Partial Fulfillment of the Requirements for the
Degree of Master of Science
in Electrical Engineering

Supervised by

Associate Professor Dr. Dorin Patru
Department of Electrical and Microelectronic Engineering
Kate Gleason College of Engineering
Rochester Institute of Technology
Rochester, New York
May 2011

Approved by:

Dr. Dorin Patru, Associate Professor
Thesis Advisor, Department of Electrical and Microelectronic Engineering

Dr. Marcin Lukowiak, Assistant Professor
Committee Member, Department of Computer Engineering

Dr. Dhireesha Kudithipudi, Assistant Professor
Committee Member, Department of Computer Engineering

Thesis Release Permission Form

Rochester Institute of Technology
Kate Gleason College of Engineering

Title:

Towards FPGA Hardware in the Loop for QCA Simulation

I, Alan Olson, hereby grant permission to the Wallace Memorial Library to reproduce my thesis in whole or part.

Alan Olson

Date

Acknowledgements

Dr. Lukowiak and Dr. Kudithipudi for their insightful review

Matt Hibbard for his assistance with the Wildcard 4 system

Konrad Walus and Gabriel Schulhof for their correspondence regarding QCADesigner

My parents Grieg and Betty Olson for their undying love and support

Dr. Eric Peskin for his assistance and guidance during the early parts of this work

and

God for his faithfulness

Abstract

Towards FPGA Hardware in the Loop for QCA Simulation

Alan Olson

Supervising Professor: Dr. Dorin Patru

As transistors begin to hit raw physical limits and performance barriers, other technologies are being researched to potentially replace conventional integrated circuit technology. Quantum-dot Cellular Automata (QCA) is one such technology which executes computations using coulomb interactions and quantum-mechanical effects. Part of this research is pursuant to the design of circuits which exploit QCA technology and take advantage of what it has to offer. These circuits must be simulated to ensure their functionality and help prove the viability of QCA. These simulations, like many scientific computing applications, can take a long time to complete; hours or days, depending on their size and complexity. Many scientific applications have benefitted from research into Field Programmable Gate Array (FPGA) application development, which has been used to accelerate the speed at which such simulations execute.

This thesis investigates the possibility of using FPGAs to accelerate the simulation of QCA circuits. The hardware developed is a streaming type architecture using floating point arithmetic and hardware/software techniques. Hardware implementation shows the system to run slower than the existing software code, but demonstrates the ability to simulate a small QCA circuit. Analysis of the design reveals good potential for achieving speedup, and an alternate design is proposed to improve the execution time. In the course of this work, improvements to the existing software are also developed and contributed to the community.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

There is constant demand for computing resources to become more powerful, more energy efficient, and less expensive. For decades, the semiconductor industry has been able to keep up with this demand, doubling processor performance every 18 months according to Moore's Law [31]. Recently, however, semiconductor-based transistors have begun to approach absolute physical barriers that will prevent further progress at this pace. The size of state of the art transistors is not far from the atomic scale, and their behaviour is increasingly determined by quantum effects. As such, it is becoming ever more difficult to continue with the current scaling trend, and alternate paths are being researched, which specifically leverage the quantum effects at this level. QCA is one such technology, with research ongoing to demonstrate its feasibility and utility. Accelerating the simulations that are performed in the course of this research can help speed up the overall progress, just as with other research fields. This can be carried out by using FPGAs to implement specialized hardware that exploits parallelism for a more efficient architecture.

1.1 Motivation

There are two primary motives to perform this work. First is to advance the state of QCA research as a means to overcome the performance barriers currently faced by the semiconductor industry. Second is to advance the state of FPGA utilization in support of simulation speedup.

1.1.1 QCA Advancement

Currently, the primary manufacturing technology for integrated circuits is based on complementary metal-oxide-semiconductor (CMOS). This technology is presently facing absolute theoretical scaling limits as current leakage, power consumption, oxide thickness, mobility degradation, threshold voltage variation and many other characteristics become increasingly difficult to control [1]. This means that traditional scaling techniques are no longer sufficient to maintain the pace of semiconductor development.

The International Technology Roadmap for Semiconductors (ITRS) [1] outlines myriad directions industry research may take in addition to traditional scaling to keep advancement proceeding apace. These include equivalent scaling, design equivalent scaling, Emerging Research Materials (ERM), and Emerging Research Devices (ERD). Equivalent scaling and design equivalent scaling focus on new forms of architectures, such as parallel or three dimensional computing structures to improve performance through design rather than implementation technology. ERM and ERD focus on developing alternate technologies to augment or replace CMOS. The field of ERM includes the development of new channel materials to facilitate the advancement of CMOS technology as well as materials to enable further development of ERD topics such as carbon nanotubes, silicon nanowires and spin materials.

QCA is one of the technologies mentioned in the ITRS. Materials research regarding macromolecules and self-directed assembly benefits QCA, and the concept of QCA is under scrutiny as a possible replacement for CMOS. It has many potential advantages over CMOS with regard to device density [51], operating speed [28], and power consumption [9], and it may even serve as the basis for a reversible computing architecture [26, 25]. These are all desirable characteristics for a technology that supports data processing to have, and indeed are sought-after goals stated in the ITRS. However, a direct comparison to CMOS is difficult because transistors and QCA cells are not directly analogous (see Section 2.1). Furthermore, none of these aspects of QCA have been fully realized in practice yet, nor have QCA circuits been manufactured at any significant scale. QCA research is ongoing, focusing on two distinct aspects: implementation and architecture.

One direction of QCA research is that of physical implementation. Although the technology shows promise, the current state of QCA is highly experimental, and it must be made more practical if it is to compete with and supercede, CMOS. QCA must be cheaply and reliably manufacturable, operational at room temperature, and able to deliver the high speed and low energy consumption demanded of it.

There are three basic types of QCA proposed: metal-dot, molecular, and magnetic. Magnetic QCA (see Section 2.2.3) has been shown to operate at room temperature, but is not able to be clocked much faster than 100 MHz [7]. Metal-dot QCA (see Section 2.2.1) promises to operate at speeds up to 10 GHz, but has not been shown to work at room temperature [43]. Molecular QCA (see Section 2.2.2) purports to be able to achieve room temperature operation [42] at up to THz speeds [28], but no actual circuits have been manufactured at any scale. These operating speeds have been determined using simulations of the adiabatic switching properties of the materials involved, but have yet to be verified in a laboratory. This research is currently the domain of physicists, material scientists, and microelectronic engineers.

This thesis relates more closely to the other important aspect of QCA research, which is nanoarchitecture research [45]. This research models the physics of QCA and ignores the manufacturing concerns so that the engineer may focus on more familiar elements like cells, wires, and gates. It is important to demonstrate that novel circuits can be created using QCA, thus making the aforementioned physical implementation research worthwhile. Is also provides optimized circuits "in the pipe," ready to manufacture as soon as possible. This research can proceed in parallel with the other, without the immediate need for physical circuits to test on, but is still held back by the large amount of time needed to simulate each design. If accurate simulations could be sped up, rapid progress in this field will ensue, verifying and correcting the functionality of large QCA circuits in hours rather than days or weeks.

1.1.2 FPGA Advancement

Field Programmable Gate Arrays (FPGAs) are hardware devices that can be programmed to implement any arbitrary logic function. There are many ways they can be designed, but usually they consist of a large array of multiplexer-based Look-Up-Tables (LUTs) that can be individually programmed and interconnected according to a programming bit stream [16]. FPGAs are used in a variety of contexts today, from Application Specific Integrated Circuit (ASIC) prototyping, to implementation platforms for end products, to a means of scientific research.

FPGAs are powerful and versatile. Unlike ASICs, which can only serve one specialized purpose after manufacture, FPGAs can be easily reconfigured to implement a large variety of hardware tasks. They also have the advantage compared to CPUs because their hardware structure can be modified and optimized for very specific tasks, and can have a very high degree of parallelism. Therefore they have become a way to achieve performance similar to ASICs without the high monetary and labour cost of developing and manufacturing an actual ASIC chip. Furthermore, a single design can be shared between groups that use different models or brands of FPGAs, and that design can be quickly implemented, studied, and used without any need to purchase a new expensive ASIC. Thus, FPGAs are very valuable to the scientific community and electronics industry as they enable the rapid distribution, testing, and implementation of novel designs.

Up to now, FPGAs have been typically used to implement fixed point or limited precision applications; due to constraints on hardware resources, IEEE-compliant floating point cores were not able to be implemented at useful speeds. However, the performance of FPGAs has been increasing even more rapidly than that of CPUs [47], and their floating point performance has improved significantly. In [47], Keith

Underwood shows that floating point performance on FPGAs has indeed advanced to match and exceed that of general purpose CPUs. Given the importance of floating point precision to scientific applications, and the usefulness of FPGAs in general, this is a promising development. Despite this advancement, relatively few applications have been developed which take advantage of it, and none have been found that combine floating point and extensive hardware/software design. Hardware/software design (see Section 3.4) can be exploited to speed up the development process by porting scientific applications from software to hardware. Showing that hardware/software design can be used effectively with floating point applications would be significant.

1.2 Previous Work

Up to now, a fair bit of work has been done to develop a number of novel circuits that could be implemented using QCA, including a microprocessor [50], fault tolerant architectures [44, 52], and various programmable logic structures [33, 23, 9, 46, 11]. Specifically, work has been done at the Rochester Institute of Technology (RIT) to develop a Combinational Logic Block (CLB) that would form the basis of a QCA-based FPGA [46]. Thus it can be seen that nonoarchitecture research is progressing, and various circuit designs exist that could be implemented with QCA when the time comes.

There are several simulation tools that have been made to help circuit designers verify the functionality of their designs. These tools include QCADesigner, MAQUINAS, QBART and HDLQ [36]. QCADesigner and MAQUINAS both focus on the cell level (for metal-dot and molecular implementations respectively), and execute numerical simulations that are based on Schrödinger equations. QBART instantiates QCA devices into a grid, and uses *ad hoc* simulation rules. This allows simulations to run faster, but the simulation environment is less flexible and not as accurate. HDLQ is a Verilog library which provides a basis for behavioural simulation of QCA circuits using existing Verilog simulation tools. The difference between HDLQ and QCADesigner is somewhat akin to the difference between Verilog and SPICE. At RIT, QCADesigner has been used for verification of small-scale circuits, and HDLQ for circuits where QCADesigner's simulation time proves impractical. For reference, the simulation of one CLB (with 22,558 QCA cells) in [46] takes approximately 50 minutes, while the QCADesigner simulation of four CLBs takes between 8 and 12 hours.

There has been much work done developing applications for FPGAs in many categories to varying degrees of success. Image processing is commonly implemented due to its relatively light numerical demands and significant parallelizability. There are also numerous Molecular Dynamics (MD) simulations implemented on FPGAs typically achieving speedups of 5x over pure software [14]. MD simulations often start as software applications using floating point, but find themselves using some other number system such as logarithmic floating point [6] or semi-floating point [15] (see Section 3.3) that makes the hardware more compact. The precision suffers somewhat, but the size of mathematical cores is reduced to allow for greatly enhanced parallelism. A limited number of applications using fully floating point numbers have also been created to speed up various applications, in some cases up to 35x [32], showing the ability of FPGAs to handle floating point applications.

1.3 Contributions

This thesis attempts to make steps toward accelerating the execution of QCA simulations, a task which is not shown to have been attempted in the literature. The design used to this end is novel in concept among other similar applications because it attempts to use IEEE 754 compliant floating point arithmetic in conjunction with hardware/software design in order to achieve high numerical precision simultaneously with shortened development time and ease of use. One possible architecture is designed and tested, but fails to achieve any speedup due to interfacing bottlenecks. The computational core itself, however, demonstrates the operational speed necessary to serve as the basis for faster systems in the future. Another design is proposed that would eliminate the most significant observed issues and potentially run much faster, while maintaining floating point precision and hardware/software usability. Additionally, some minor improvements are made to the QCADesigner software, including a bug fix in the algorithm and an optimization that significantly speeds up a common function.

1.4 Organization

This thesis is organized as follows: Chapter 2 provides the background to QCA necessary to understand what is being simulated, including descriptions and background of existing software simulators. Chapter 3 offers a detailed look at FPGAs, their basic architecture and functionality, and how they are used to accelerate various computing applications. This section also contains a brief examination of different number systems, and how they may be applied in different designs. Chapter 4 details the design process, including the prerequisite software analysis and explication of the final implemented hardware. Chapter 5 reports and analyzes the observed results and Chapter 6 discusses their significance and suggests directions for future work.

Chapter 2

QCA Background

Quantum-dot Cellular Automata (QCA) is a technology concept devised by Tougaw and Lent in 1993 [24]. It is a way to create digital electronic devices without using transistors that has potential to be faster, denser, and more energy efficient than CMOS technology. Rather than computing with voltage levels requiring current flow, QCA computes via field polarization [2], in such a way that the ground state of the system corresponds to the solution state of the computation. Much progress has been made since 1993 in terms of design and fabrication techniques. This section explains in detail how this form of computation works and can be implemented. Note that the logical architecture and functionality is the same independent of the method of manufacturing and implementation.

2.1 Basic QCA Operation

This section describes how data is stored, transferred, and manipulated in QCA, starting with a description of a basic cell, then building wires and gates out of cells. The clocking structure, which is central to the functionality of QCA, is also described.

2.1.1 QCA Cell

A QCA Cell is the basic building block of any QCA circuit [3]. It consists of two pairs of charge sites, each pair joined by a tunnel junction,

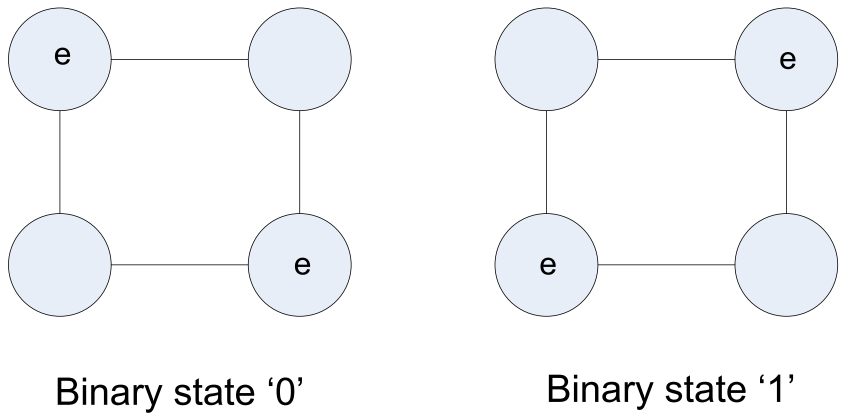


Figure 2.1: A QCA cell (reproduced from [45])

and two distinct charges. The specific configuration of the cell represents a binary value stored in the cell as shown in Figure 2.1. There are only two stable configurations in a QCA cell, corresponding to the electrons occupying opposite corners, hence representing two alternate binary values. The charges can move between charge sites by means of quantum tunneling, a process which is facilitated by the clock (section 2.1.4). The configuration of a cell can be calculated in simulation by using a Schrödinger equation with a quantum-mechanical Hamiltonian, and is influenced by any other QCA cells surrounding it. A cell can also theoretically be arranged with the charge sites rotated by 45 degrees to facilitate wire crossings [9].

2.1.2 QCA Wires

When QCA cells are arranged in a line, they form a wire. In contrast to CMOS technology where simple metal lines are used to transfer data, it is the same complex functional devices that process data in QCA which transmit data. A QCA wire functions by local coulombic interactions from one cell to the next, generating a "domino effect" that propagates data from a fixed input to the measured output as shown in Figure 2.2.

Alternately, rotated cells may be arranged to form an inversion chain

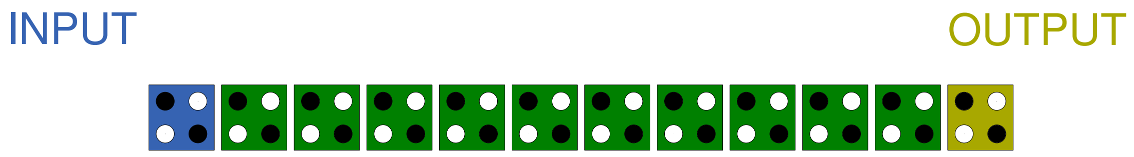


Figure 2.2: A normal QCA wire (reproduced from [45])

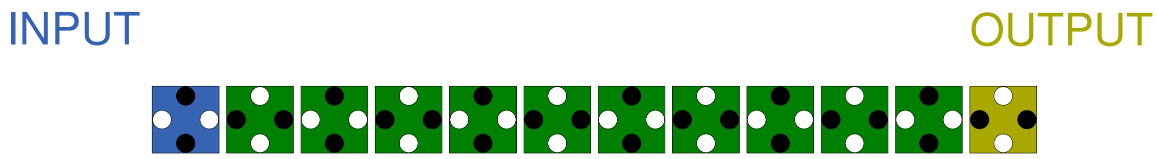


Figure 2.3: An inversion chain (reproduced from [45])

as shown in Figure 2.3. This is like a normal QCA wire, but the polarization flips between each cell. Therefore an inversion chain must be an odd number of cells in order to preserve the value from input to output. The main use of an inversion chain is to allow coplanar wire crossings [9]. In a wire crossing such as that shown in 2.4, the cell at the crossover point in the primary wire has a neutral coulombic effect on either of the nearby cells in the secondary wire, allowing data to cross over unaltered. Multi-layer QCA circuits have not been shown to be feasible, but coplanar crossings such as the ones described here may be. Wire crossings, however they are implemented, are important to having novel circuits in a compact area.

2.1.3 QCA Logic

The logical basis for QCA circuits are majority gates and inverters. A QCA inverter has the same function as the familiar CMOS inverter, *i.e.* inverts a logical value from input to output. A majority gate is a three-input gate whose output matches the most popular input value. The boolean expression for this is $F = AB + BC + AC$, and the truth table for a majority gate is shown in Table 2.1. The QCA schematics of an inverter and a majority gate are shown in Figure 2.5 and 2.6.

It can be seen from the schematic that a three-input majority gate is fundamentally the most complex single gate structure that can be

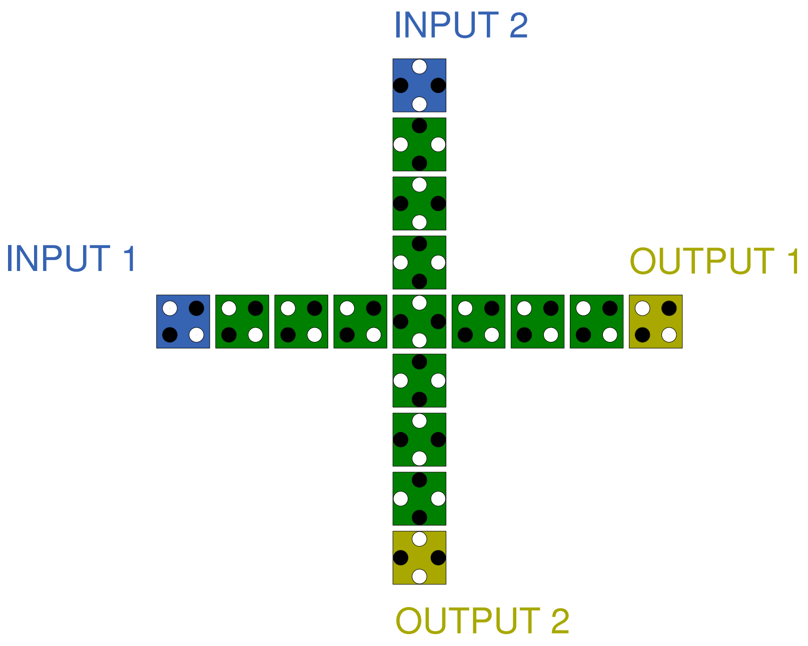


Figure 2.4: A wire crossing in QCA using an inversion chain (reproduced from [45])

implemented in QCA. It functions due to the bistable nature of a QCA cell, which will “snap” into one configuration or another based on which cell polarity has a more dominant influence. A majority gate can be programmed to function as either an AND or an OR gate by holding one of the inputs at a logical zero or one respectively. With AND, OR and INV, a complete set of logical devices can be implemented in QCA, enabling the creation of any digital circuit that can be made currently in CMOS.

2.1.4 The Clock

All QCA functions are driven by an underlying clock structure which is essential to the controlled operation of each cell and governs the flow of data. There are actually four clocks in a QCA circuit, each separated by 90 degrees of phase shift, and each cell is attached to one of the four clock zones. The four phases of each clock period may be labeled as relax, switch, hold, and release, indicating each phase’s effect on a cell [52]. A timing diagram showing their basic progression is shown in

A	B	C	F
0	0	0	0
0	0	1	0
0	1	0	0
0	1	1	1
1	0	0	0
1	0	1	1
1	1	0	1
1	1	1	1

Table 2.1: Truth table for a majority gate

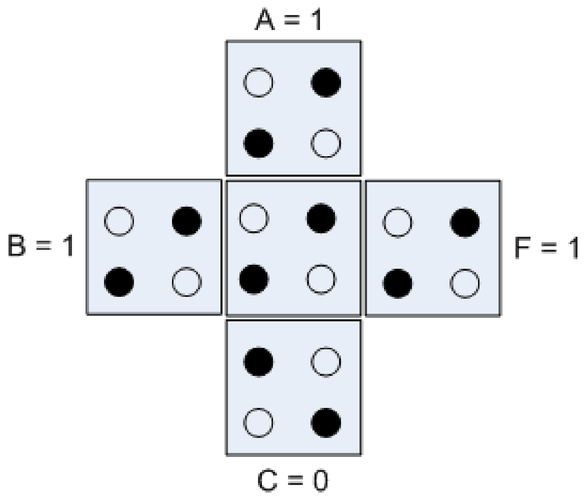


Figure 2.5: Majority gate in QCA

Figure 2.7

During the relax phase, a cell is free to switch polarizations based on the surrounding cells' influence. In the switch phase, the clock level progresses from low to high, and the polarization of the cell becomes locked in place. When the clock reaches the high level, this is the hold phase, and the polarization of the cell is fully locked and able to influence other cells without being altered itself. After the hold phase comes the release phase where the clock progresses from high to low, and the cell gradually loses its fixed value to re-enter the pliable relax state.

These alternating clock phases allow the controlled progression of

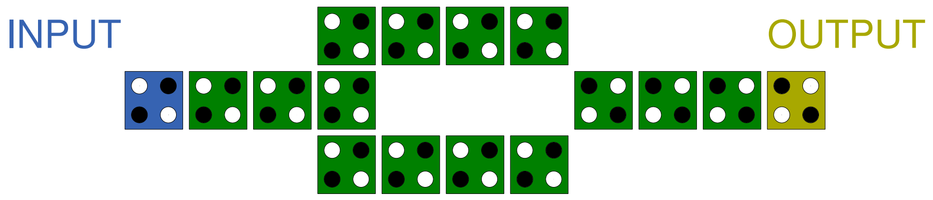


Figure 2.6: An inverter gate in QCA (reproduced from [45])

data in one direction through a QCA circuit, from input to output, with minimal error. Figure 2.8 illustrates the prescribed arrangement of clock zones in a wire. Input is on the left, output is on the right, and each colour represents a different zone, progressing in order from Clock 1 to Clock 4 from left to right. However, because every cell is clocked, including those making up the wires, QCA is a deeply pipelined architecture. Every full set of clock zones between input and output adds one clock period of latency to the circuit, and there is a limit to how many consecutive cells may be in one clock zone. Therefore, even wires will add to circuit latency, meaning the circuit designer must take extra care in aligning data arrival times.

The clock structure in QCA is currently implemented as a standard CMOS clock network directly below the QCA circuit whose voltage level directly affects the depth of the energy wells holding the charges. The clock can generally be routed wherever it is needed in the circuit, but with four clocks to route, the actual layout could be quite complex, and minimizing skew can be difficult. Research is ongoing to develop techniques to design QCA circuits with more regular and easily routable clock zones.

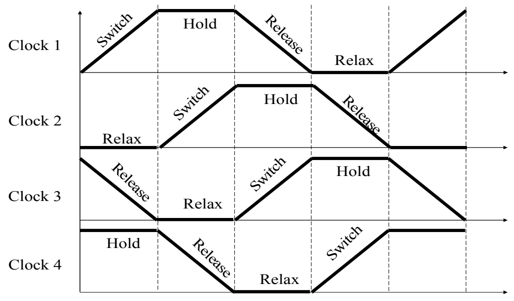


Figure 2.7: Waveform showing how QCA clock zones operate (reproduced from [52])

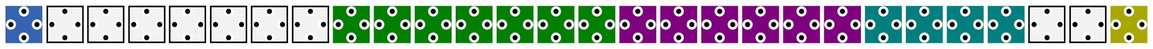


Figure 2.8: Clock zones as they should appear in a wire

Concerns have been raised that the power savings of QCA may be offset by the power consumption of a clock running at the proposed speeds of 10 GHz and more. This is addressed in [4], where Lent *et. al.* simulate the power consumption of the type of clock network used in QCA. Their findings show that the clock does not significantly contribute to the overall power consumption of a QCA circuit, and that a 100 GHz QCA clock consumes similar power to a 2 GHz clock driving a conventional CMOS circuit.

2.2 Implementations

Three methods of implementation for QCA have been proposed: Metal-dot, molecular, and magnetic. They each have their own advantages and disadvantages, and are all actively being researched.

2.2.1 Metal-Dot

The most popular form of QCA implementation being researched is the metal-dot implementation. This uses small dots of conductors such as Aluminum [22], Silicon or Gallium-Arsenic [2] as charge sites. Each 4-dot cell in a metal-dot QCA cell is typically a square of about 20 nm on a side. It has been shown to work in a lab environment, in which some very basic, though important, circuits have been demonstrated at temperatures below 1 K [34, 35]. In theory, based on adiabatic switching simulations of metal-dot cells, the clock speed of this form of QCA circuit can reach into the tens of GHz, but only up to MHz range has been demonstrated [43]. Metal-dot is the form of QCA best-suited to use current CMOS manufacturing lines without too much upgrading or modification being needed; it would use the same basic lithography techniques, and has similar feature sizes as the current state of the art transistor technology. It is proposed that if the individual cells and dots can be made smaller, higher operating temperatures may be achieved.

2.2.2 Molecular

Molecular QCA is a step up from metal-dot in every way, except feasibility. Molecular QCA proposes composing a QCA cell out of just two molecules specially designed to have two primary charge sites and mobile electrons. In theory, because individual molecules are being used, the “dots” are small enough to allow operating temperatures up to 450 K [51], *i.e.* well above room temperature. It is also theorized that molecular QCA supports the highest clock speeds out of the three implementations: up to 1 THz [28]. However, development of molecular QCA has not progressed very far beyond simulation. Several candidate molecules are being studied, and have been experimentally shown to behave as a QCA cell, but no complex circuits have been manufactured to date.

2.2.3 Magnetic

Magnetic QCA is slightly different from metal-dot and molecular, because instead of charges jumping between charge sites in a cell, a cell consists of a magnetic dipole that changes its alignment based on its neighbours. The same type of architecture and logic applies, only the mechanics of how it works changes. Magnetic QCA has been shown to work at room temperatures, and possesses a natural hardness against radiation, but simulation of the switching properties of a magnetic QCA cell suggest that it may only work at speeds up to 100 MHz [7]. Despite the low operating speed, magnetic QCA may find applications in areas where low power consumption and high reliability are more important than operating speed, such as in medical implants or space exploration.

2.3 QCA Designer

The simulation tool that RIT and many other institutions [44, 18, 41, 46] use to test their QCA circuits is QCADesigner [48, 49]. It is a program that allows a designer to easily create and test a QCA circuit using a graphical user interface (GUI). It offers two different simulation engines (coherence vector and bistable) with a variety of customizable parameters. QCADesigner was initially developed by Konrad Walus *et. al.* in the Microsystems and Nanotechnology Group at the University of British Columbia, but is now an open source project, so obtaining and modifying the source code is very easy, making it an ideal choice for an application to accelerate.

2.3.1 Bistable Simulation

The bistable simulation engine is the preferred QCADesigner engine for simulating circuits at RIT, and therefore the engine selected for the work of this thesis. It models a QCA cell as a system having two stable states, and the polarization of a cell is calculated based on the surrounding polarizations and kink energies. The kink energy between two cells refers to the energy cost of the two cells having opposing polarizations. Naturally, this value tends to decrease as the distance between cells increases. The main equation used to determine the kink energy is shown below [48].

$$E_{i,j} = \frac{1}{4\pi\epsilon_0\epsilon_r}\frac{q_i q_j}{|r_i - r_j|} \tag{2.1}$$

This equation is used to calculate the electrostatic energy between two individual dots in cell i and cell j. ϵ_0 is the permittivity of free space, and ϵ_r is the relative permittivity of the material system in use, which can be specified by the user in the simulation setup window. q_i and q_j are the charges stored on the two dots from cell i and j respectively, and $|r_i - r_j|$ is the distance between the dots. This equation is applied and summed for every combination of dots in each cell, and the difference between the results for aligned and misaligned polarities is taken as the kink energy. Because there is no net charge on the cell, the interaction is quadrupole-quadrupole based, so it decays very rapidly as distance increases. The set of kink energies for all cells only needs to be calculated once during a simulation, during the initialization phase before any polarizations are calculated.

The kink energies between a cell and its neighbours are then used as terms in the equation below to determine the polarity:

$$P_i = \frac{\frac{1}{2\gamma}\sum_j (E^k_{i,j} P_j)}{\sqrt{1 + \frac{1}{2\gamma}\sum_j (E^k_{i,j} P_j)}} \tag{2.2}$$

where γ is the tunneling potential, which changes with the clock value, P_j is the polarization of the jth neighbour of the current cell, and $E^k_{i,j}$ is the kink energy between the current cell and the jth neighbour cell. This equation is carried out once for every cell in a QCA circuit, for every iteration the simulation goes through. One "sample" of the simulation may consist of several iterations of the circuit, in which the state of each cell is recalculated until a maximum allowable change threshold is reached. This bistable simulation engine is able to simulate a large number of cells relatively quickly to verify logical correctness and basic structural validity, but it lacks certain considerations, such as temperature and dissipative effects, required to execute dynamic simulations with a high degree of accuracy.

2.3.2 Coherence Vector Simulation

The coherence vector simulation engine is a slightly more detailed simulation than the bistable simulation engine. Like the bistable engine, it assumes a cell is basically a bistable system, but the state is calculated in a more sophisticated way. The status of a cell is tracked using a coherence vector where the polarization is represented as the z component of this vector. Each component of the vector is calculated by multiplying the trace of the density matrix by each of the Pauli spin matrices.

$$\lambda_i = Tr\{\hat{\rho}\hat{\sigma}_i\} \qquad\qquad i = \{x, y, z\} \tag{2.3}$$

The main driver of the coherence vector simulation is

$$\frac{\partial}{\partial t}\overrightarrow{\lambda} = \overrightarrow{\Gamma} \times \overrightarrow{\lambda} - \frac{1}{\tau}(\overrightarrow{\lambda} - \overrightarrow{\lambda}_{ss}) \tag{2.4}$$

which models the change in the cell's state over time, including dissipative effects. The Gamma vector is defined as

$$\overrightarrow{\Gamma}\{x, y, z\} = \frac{1}{\hbar}[-2\gamma, 0, \sum_{j \in S} E^k_{i,j} P_j] \tag{2.5}$$

where gamma is the tunneling potential (tied to the clock), S is the effective neighbourhood of cell i, and $E^k_{i,j}$ and P_j are the kink energy and polarization respectively of cell j. τ is the relaxation time of the cell. $\overrightarrow{\lambda}_{ss}$ from equation 2.4 is the steady-state coherence vector, and is defined as

$$\overrightarrow{\lambda}_{ss} = -\frac{\overrightarrow{\Gamma}}{|\overrightarrow{\Gamma}|}\tanh\Delta \tag{2.6}$$

where delta is the temperature ratio:

$$\Delta = \frac{\hbar|\overrightarrow{\Gamma}|}{2k_B T} \tag{2.7}$$

where T is the temperature in kelvin and k_B is Boltzmann's constant. The simulator evaluates these equations for each cell in the circuit for each timestep until the end time is reached, as specified by the user. It is plain to see why this method may be more accurate and take much longer to run.

2.3.3 Simulation Speed

Each simulation of a 22,558 cell CLB takes a little less than an hour to complete using the bistable simulation engine. The bistable simulation engine is used in our research because it provides basic architecture verification an order of magnitude faster than the coherence vector simulation. For larger circuits, exponentially more time is required. For example, a circuit four times as big with four CLBs requires 10 hours to complete the simulation. The CMOS versions of these circuits would not be extraordinarily complex in terms of the number of gates (*i.e.* devices), 572 gates for one CLB; but because gates *and* wires are made up of complex fundamental devices in QCA, the simulation is much more complex and time consuming.

It is observed that for both of these simulation engines, the new polarization of a cell may be calculated concurrently with the new polarization of any and all other cells without flouting any data dependencies. This means that there is a great amount of parallelism that may be exploited when executing these simulations. If one computational core can be assigned to each cell, then a very large circuit would take the same time to simulate as a small circuit currently does, yielding extraordinary speedup. Such a task is well-suited to FPGAs, as they can easily replicate custom processing hardware into multiple instances on one chip.

Chapter 3

Using FPGAs for Application Speedup

An FPGA is a chip containing a reconfigurable hardware structure which can be programmed to implement any desired hardware circuit. It has no fixed functionality, and unlike a CPU, programming it results in an actual reassignment and reconfiguration of hardware resources. FPGAs often have a repeating structure with computing and memory resources spread throughout the chip, allowing them to be used for a large number of applications. Their versatility allows them to serve as specialized application specific hardware platforms without the high development costs associated with ASICs. They can be used for hardware prototyping before sending a design to ASIC production, or may be used as end platforms in themselves.

3.1 Basic Architecture and Functionality

Although there is no universal design standard for an FPGA, the most common approach is to use a combination of programmable logic blocks and programmable interconnects. Logic may be programmed using look up tables that consist of SRAM cells attached to multiplexer inputs, and interconnects may be programmed using solid-state switches. Some FPGAs may include embedded multipliers, digital signal processing (DSP) units, block RAM, and other features, but the basic principle remains the same. FPGAs are usually laid out in a repeating grid pattern, which facilitates their use for parallelizing calculations. Computing resources are distributed evenly throughout the chip, allowing multiple computational cores to be implemented at once, cores that may or may not be identical to each other. Memory resources are also distributed such that computing resources in one region may use memory in that region while other memory is free to be used by other resources, resulting in a very large effective memory bandwidth.

3.2 Use In Application Speedup

A large class of applications commonly implemented on FPGAs is image processing. These are ideally suited to FPGAs because they rely on simple fixed point mathematics and their operation is often highly parallelizable. Therefore, there is a long history and large body of work related to accelerating image processing on FPGAs. Of particular interest is the 2D Convolution algorithm [37], often used for image noise reduction, which replaces a pixel value with a weighted average of surrounding pixel values. The mathematical algorithm has some similarities to that used in the bistable simulation engine of QCADesigner, both using a sum of products term based on surrounding values. It calculates the new value of a pixel by computing the sum of products of surrounding pixel values, times the convolution weights (see figure 3.1). There is even an example of how to use a basic 3x3 convolver design to compute arbitrarily large convolution kernels. This might be analogous to using a basic QCA simulation core with a 50 nm radius of effect to enable the simulation of any larger radius of effect without having to use a different core.

Many advanced scientific applications are also executed on FPGAs, using a variety of implementation methods. In [6], an alternative number system, logarithmic floating point (see Section 3.3), is implemented and an application to accelerate Quantum Chromodynamics calculation is constructed using Handel-C. Handel-C is one of many alternatives to VHDL, like JHDL [19] and Carte [40, 21], that abstract the hardware description task to a level more familiar to software developers. This can make development easier by lowering the learning curve and making the hardware description language more similar to the original language of the program being adapted. Alternative number systems are

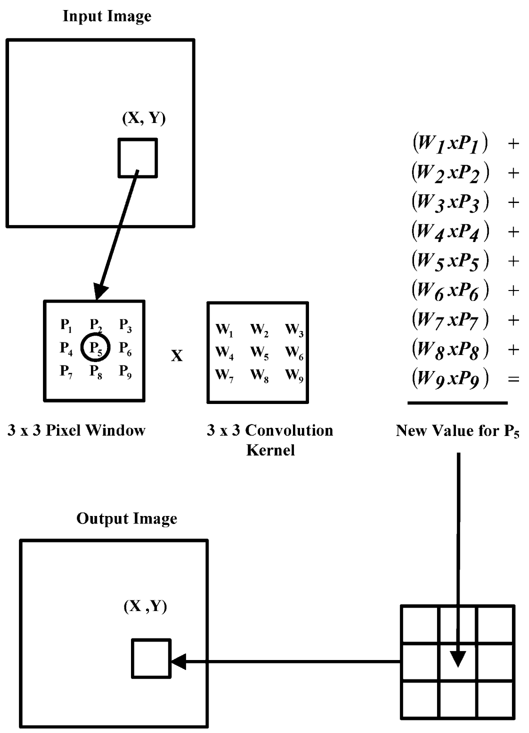
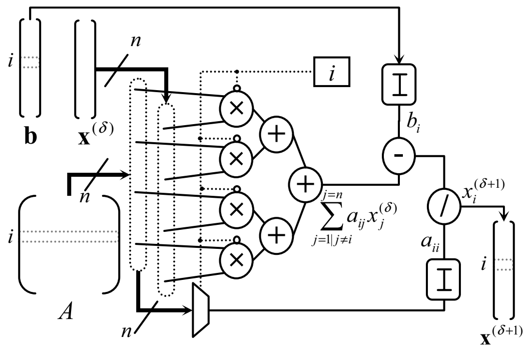


Figure 3.1: Basic mathematical structure of 2D convolution algorithm (reproduced from [37])

also a common practice in FPGA development to reduce the hardware demands of a system where possible [15, 12, 13].

Molecular Dynamics (MD) simulations are also often implemented on FPGAs to good effect, often achieving greater than 5x speedup [21, 15, 14, 13], sometimes much more. A variety of architectures and methodologies are used to obtain these results, including hardware/software implementations [21, 40, 15, 39, 13], discrete event simulation [29], and multigrid computing [14]. In hardware/software implementations, only a portion of the application is ported to hardware, while other parts are left done in software for ease of development and optimum resource usage. To speed up the simulation, discrete event simulations use a variety of simplifications to advance MD simulations by event rather than timestep. The multigrid computing technique breaks the computation up into spatial grids to parallelize computation and memory access, and iteratively uses different sized grids to balance between



n
i
b
$\mathbf{x}^{(\delta)}$
i
b_i
$x_i^{(\delta+1)}$
$\sum_{j=1\|j\neq i}^{j=n} a_{ij}x_j^{(\delta)}$
n
i
a_{ii}
i
A
n
$\mathbf{x}^{(\delta+1)}$

Figure 3.2: Binary reduction tree structure used in [32]

speed and accuracy.

Some floating point applications, including SPICE simulation [20], matrix operations [10, 27, 32], and Fast Fourier Transforms (FFTs) [17] have also been implemented on FPGAs with speedups of up to 18x compared to software implementations. For matrix operations, which involve relatively simple multiply-accumulate mathematics, many floating point cores can be implemented using spatial parallelism to produce a result quickly in a binary reduction tree pipeline. For example, the one used in a Jacobi iterative solver (see Figure 3.2) [32]. However, for the more complex mathematical demands of SPICE and FFT, some spatial parallelism is exchanged for longer pipelines that continuously compute a stream of data.

3.3 Number Systems

There are a variety of number systems that can be used for calculation in FPGA application, and which one is selected depends on the requirements of the application. Floating point is typically preferred in scientific applications due to its high dynamic range and good relative accuracy. This means that a very large range of values can be stored while at the same time having a very small percent error, independent of the magnitude. Floating point comes in two basic flavors: single precision and double precision. The IEEE 754-2008 Standard for Floating Point Arithmetic [8] defines single precision to be a 32 bit

0	1	0	0	0	1	1	1	0	0	0	1	0	0	0	1	0	1	1	1	0	0	1	1	0	0	1	1	0	1	1	1	= 37235.215
sign	exponent = 142 - 127 = 15								mantissa = 1 .00100010111001100110111 = 1.1363286																							

Figure 3.3: A 32 bit floating point number

number with 1 sign bit, 8 exponent bits, and 23 mantissa bits. Double precision has 1 sign bit, 11 exponent bits, and 52 mantissa bits to make a 64 bit number. Both of these precision levels function in the same way. The exponent is determined by the exponent bits' difference from a bias value to give a positive or negative exponent. The mantissa bits combine with an unrepresented implied leading 1 to give the base value on which the exponent will operate, and the value is read in decimal as $(-1)^s * m * 2^e$. Figure 3.3 shows a 32 bit floating point example. This is a very versatile method of representing numbers. However, due to the complexity of operating on such representations, floating point arithmetic cores are large compared to other implementable systems, so they are often eschewed in favor of simpler number systems.

The simplest form of number system is fixed point, often realized as just integer arithmetic. This system is used in applications where very little dynamic range is needed, such as some forms of image processing [37, 38, 30, 5]. Fixed point arithmetic cores are simple, compact, and fast, but only because the computations are simple. They are not very useful in scientific applications where higher dynamic ranges and more accuracy are required.

There are other number systems which are more versatile than fixed point, but easier to implement than floating point. A logarithmic floating point number system converts floating point numbers into a logarithmic floating point domain, allowing logarithm arithmetic to be used [6]. This means that multiplication, division, and square root operations are all made much simpler, but addition and subtraction are more complex than in standard floating point. This is because when multiplying or dividing logarithms, only simple addition is required whereas addition and subtraction operations require large look up tables This number system is effective when the majority of operations are multiplication, division, or square root, but the advantage is lost if too many additions or subtractions are present.

A system called semi-floating point [15] may also be used. In this system, the decimal may shift only to certain places. This allows the arithmetic to be carried out using a limited number of fixed point cores or look up tables, simplifying the hardware requirements compared to floating point. However, it requires that a small amount of dynamic range is available and that all numbers in the computational problem can be represented accurately with the limited set of decimal places. Additionally, unlike floating point, neither logarithmic nor semi-floating point arithmetic are standardized.

FPGAs have not historically been well-known as effective platforms for floating point applications due to the large hardware requirements, but this has changed in recent years. Studies such as [47] have shown that FPGA floating point performance has increased to the point of surpassing general purpose CPUs, especially since the inclusion of embedded multiplication units. Still, only a small number of floating point FPGA applications have been developed, so there is much room to explore this feature.

3.4 Hardware/Software Design

Hardware/software design is an application design technique where the hardware and software aspects are designed simultaneously to allow a software program to communicate with and use an FPGA to carry out specialized functions. This is a practice that is possible because of the programmable nature of FPGAs. It differs from using plug-in ASICs as part of an application because the hardware can be more specialized, and the software can potentially use it more effectively [39]. Hardware/software design allows a developer to exploit the strengths of both software and hardware. For example, software is good at handling tasks that require complex control structures, and can easily present setup or results in GUI form for ease of use and interpretation, while hardware is able to parallelize computation for faster execution. Hardware/software design is also very well suited to porting an application from a software-only implementation to a platform which allows hardware acceleration, because a lesser portion of the application will need to be redeveloped, cutting down on development time. In order to be effective, however, the communication avenue between the hardware and software must not present a significant bottleneck.

3.5 Nature of QCADesigner Computation

The actual algorithmic structure of QCADesigner bears some striking similarity to the 2D convolution image processing algorithm [37]. A central cell (analogous to a pixel) is acted upon by a sum of weighted products in the form of surrounding cells' polarizations and kink energies (analogous to the pixel values and convolution weights). There were techniques presented in [37] which could be used to make arbitrarily large and efficient convolution grids (analogous to a variable radius of effect) for large images (analogous to circuits), using repeated instances of a relatively small mathematical core. However the mathematics present in the 2D convolution problem are comparatively simple, using integer arithmetic and a fixed convolution grid with 4-bit weights for each pixel, and only 16-bit pixel values. QCADesigner requires 32-bit floating point values for both cell polarizations and kink energies (pixel values and convolution weights), so the hardware would consume far too much area to allow this type of architecture to be implemented on available resources.

The numerics of QCADeisgner are more similar to N-body physics simulations such as molecular dynamics and quantum chromodynamics. If a QCA cell is considered like a "particle," the two problems have many similarities. Both the cell and the particle change their state based on the state and proximity of other surrounding particles, and both problems require high-precision mathematics. On the other hand, QCA is a simpler problem than N-body simulations because the cells ("particles") do not move, and are generally limited to two dimensions instead of three, therefore making some forms of grid-based memory structures [14] easier to manage.

The floating point FPGA applications in [20], [17], and [32], also have similar numerical demands to QCADesigner, but their architecture and controls are more straightforward. They are implemented using floating point computational pipelines and data streaming techniques, easily adaptable to other mathematical requirements. Although, these applications' mathematics are limited to multiplication and addition, while QCADesigner demands division and square roots as well. Division and square roots are more complex functions than multiplication and addition, so there will be more hardware consumed in each core, leaving less room for multiple cores.

3.6 The Wildcard Hardware

After completion of the background research, and after it was determined that hardware/software design was viable for this application, an appropriate implementation platform is selected. An Annapolis Microsystems Wildstar II is used in [13] and [14] to implement a hardware/software application, which is a PCI-based peripheral for use with desktop computers. It possesses two Xilinx Virtex II FPGA chips as computing elements, as well as on-board memory and PCI communication drivers for interaction with software.

A similar upgraded card is available, the Wildcard 4, which has a Xilinx Virtex 4 FPGA and increased on-board memory. Ready-made libraries for both hardware and software development exist which support high data transfer rates over the PCI interface, facilitating the rapid establishment of effective communication between hardware and software.

In addition to the Virtex 4 chip, the Wildcard 4 also has 128 MB of DRAM, 8 MB of SRAM, two clocks (one fixed on the communication bus, the other programmable up to 240 MHz), and PCI communication. Refer to figure 3.4 for a block diagram of the system. Also included are libraries for interacting with the card via C programs and simulating a software driver in VHDL. The PCI interface allows for several types of communication, including register transfer, direct memory access

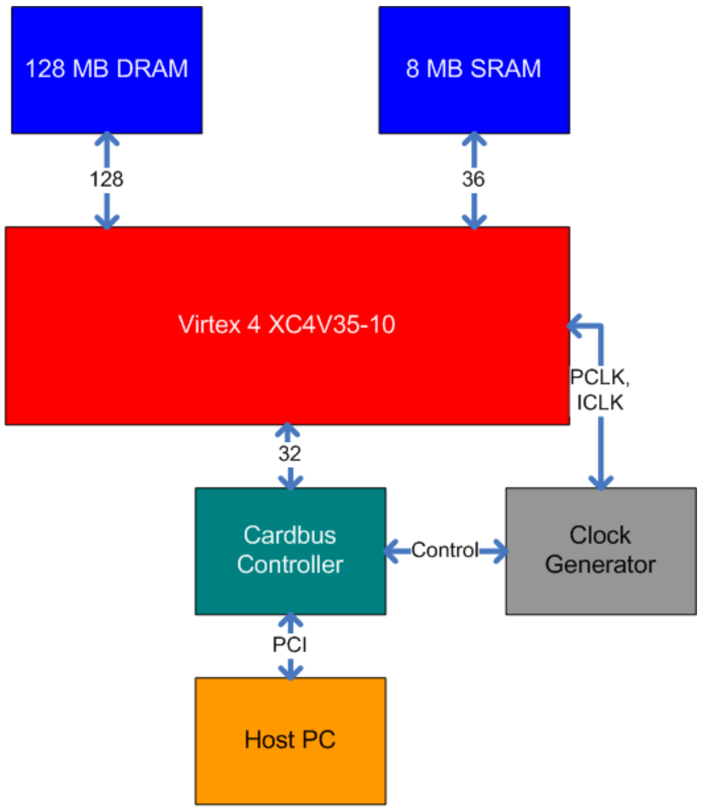


Figure 3.4: Block diagram of Wildcard 4 hardware

(DMA), and interrupt generation.

Chapter 4

Proposed Simulation Architecture

The design of a system that could potentially speed up the execution of QCADesigner could take one of many shapes based on a large set of constraints. The functionality of the design is explained in this chapter, as well as some of the related key decisions and software analysis that took place beforehand.

4.1 Software Analysis

Prior to committing to any new design, the QCADesigner software is thoroughly analyzed. This analysis includes application profiling using gprof; qualitative analysis of the code by means of examination and debug outputs to determine where most of the computation occurs; further code analysis to determine the memory structures used to store data and assess how best to use the FPGA resources; numerical analysis to track in detail the dynamic range of critical values in the calculation and compare the accuracy of double precision to single precision floating point. The goal here is to identify which section of code should be converted to hardware, and confirm that it has as much parallelizability as expected based on the simulation engine's mathematics.

4.1.1 Code Analysis

The first step in analyzing the QCADesigner source code is done using gprof, which generates an execution profile of the application. This breaks down how much computation time is spent in each function, and gives other statistics like how many times each function is called, and where it is called from. This is done by adding a `-pg` option to the `gcc` input, running the application, and running the gprof tool on the resulting profile data dump. Each such execution profile is generated by opening QCADesigner, loading a circuit, setting up the simulation, running the simulation to completion, then closing QCADesigner. The results of this gprof analysis show that for simulations of any appreciable size (*i.e.* more than one second long), the dominant function is `run_bistable_simulation` (see Appendix A for a complete code listing of this function), which takes approximately 90% to 98% of the execution time. This initial analysis shows that there is a high enough concentration of processor time spent executing a single function for QCADesigner to be a promising candidate for hardware acceleration.

Deeper analysis of the code and application execution details reveals greater detail in terms of what specific operations are taking up the most execution time. For a particular run of the CLB simulation, 96.6% of the execution time is spent in the function `run_bistable_simulation`, which includes 141 seconds of “initialization” and 3500 seconds of “simulation.” Initialization takes place in lines 89 through 168 of listing A.1, and most of the time consumed there is actually used by a function called `bistable_refresh_all_Ek`, which calculates all of the kink energies that will be used in the simulation. The simulation takes place in lines 170 through 362 of listing A.1, but the execution time is dominated by a smaller loop of code, lines 268 through 318 in listing A.1 (reproduced below for convenience).

```
    iteration = 0;
    stable = FALSE;
    while (!stable && iteration < max_iterations_per_sample)
      {
      iteration++;
      // -- assume that the circuit is stable -- //
      stable = TRUE;

      for (icLayers = 0; icLayers < number_of_cell_layers; icLayers++)
        {
#ifdef REDUCE_DEREF
```

```
        number_of_cells_in_current_layer = number_of_cells_in_layer[
            icLayers] ;
        for (icCellsInLayer = 0 ; icCellsInLayer <
            number_of_cells_in_current_layer ; icCellsInLayer++)
#else
        for (icCellsInLayer = 0 ; icCellsInLayer <
            number_of_cells_in_layer[icLayers] ; icCellsInLayer++)
#endif
          {
          cell = sorted_cells[icLayers][icCellsInLayer] ;

          if (!((QCAD_CELL_INPUT == cell->cell_function)||
                (QCAD_CELL_FIXED == cell->cell_function)))
            {
            current_cell_model = ((bistable_model *)cell->cell_model) ;
            old_polarization = current_cell_model->polarization;
            polarization_math = 0;

            for (q = 0; q < current_cell_model->number_of_neighbours; q
                ++)
              polarization_math += (current_cell_model->Ek[q] * ((
                  bistable_model *)current_cell_model->neighbours[q]->
                  cell_model)->polarization);

            // math = math / 2 * gamma
            polarization_math /= (2.0 * sim_data->clock_data[cell->
                cell_options.clock].data[j]);

            // -- calculate the new cell polarization -- //
            // if math < 0.05 then math/sqrt(1+math^2) ~= math with
                error <= 4e-5
            // if math > 100 then math/sqrt(1+math^2) ~= +-1 with error
                <= 5e-5
            new_polarization =
              (polarization_math >  1000.0)   ?  1  :
              (polarization_math < -1000.0)   ? -1  :
              (fabs (polarization_math) <     0.001) ?
                  polarization_math :
                polarization_math / sqrt (1 + polarization_math *
                    polarization_math) ;

            // -- set the polarization of this cell -- //
            current_cell_model->polarization = new_polarization;

            // If any cells polarization has changed beyond this
                threshold
            // then the entire circuit is assumed to have not converged.
            stable = (fabs (new_polarization - old_polarization) <=
                tolerance) && stable ;
            }
        }
    }
}//WHILE !STABLE
```

Listing 4.1: Main simulation loop

The loop in listing 4.1 iterates through every cell in every layer of the circuit being simulated and calculates a new polarization for that cell, forming the essential basis of the simulation engine. It reiterates until total circuit stability is achieved, or until the maximum number of iterations has elapsed. It executes to its own completion once for every sample in the simulation, thus these 50 lines of code constitute a very large portion of the total execution time of a simulation run by QCADesigner. Thus, listing 4.1 represents a portion of QCADesigner code that would be desirable to be executed in hardware.

Examining the code for data dependencies, which would affect how it could be accelerated, it is clear that each sample of the simulation must be executed sequentially, so there is no possible way to parallelize the computation per sample. It can also be observed that some cells whose polarizations are calculated in later iterations of the loop in listing 4.1 will be affected by cell polarizations that are calculated in preceding iterations. However, this represents a “numerical problem” within the calculation, rather than an actual data dependency, as mentioned in the code’s comments (lines 147-149 of listing A.1). Comparison of the theory of the bistable engine with the actual C code execution indicates that no such data dependency is necessary in the calculation. Its existence in the software is due to the nature of the storage structures and limited resources. Memory resources on the PC are constrained due to operating system overhead and the need for multitasking, preventing the data duplication necessary to avoid the numerical problem altogether. The software attempts to mitigate this numerical problem by randomizing the cell list order (lines 152 to 162 in listing A.1) so that a minimum of influence is bequeathed by the early changes.

The lack of any true data dependencies between cells means the idea presented in Section 2.3 of calculating every cell simultaneously is mathematically plausible. This confirms that great parallelizability exists to be exploited. The limiting factor of the practical implementation would lie with the availability of hardware resources, but given a large enough chip, such an architecture could be realized.

The critical data in the calculation, as can be seen in listing 4.1, are the kink energies, polarizations, and gamma (clock) value associated with each cell. The specifics of how they are stored in memory, and how often they change, influences what the most effective design for retrieving, calculating, and delivering data will be. The communication functions in the wildcard libraries operate much more efficiently with data in contiguous blocks such as arrays, and larger transfers are always more efficient. Kink energies are stored as contiguous arrays associated with each cell, but only contiguous by cell, so high volume transfers are not practical. However, kink energies do not change between iterations or samples in the simulation, so they can be transferred one time at the beginning and kept for the entire execution time. Polarizations are stored by cell, individually, so no bulk transfers can be made without rearranging the data. Additionally, they change for each iteration of the simulation, so they must be updated each time. Gamma values are stored in an array by sample index, but there are only four distinct values that need to be used at any given time. They could be transferred in high volume for all samples, or could use the alternate register transfer method on a per-sample basis.

The total memory this data occupies is also important to determine how it may be used. They are stored in 64 bit double precision floating point variables, and a CLB has about 22,000 cells. Given an effective radius of 500 nm (for reference, RIT's simulations use a radius of 50 nm), the average number of neighbours per cell is about 400, which means 400 kink energies must be stored for each cell. This means that a total of 70 MB of storage is required for kink energies. For polarizations, only one polarization per cell is required, for a total of just 176 kB. For a 300,000 sample simulation as we are running, 9.6 MB would be required for gamma value storage. These totals would be halved if using single precision floating point. Either way, there is enough memory on-board the FPGA to accommodate all this data locally.

4.1.2 Numerical Analysis

An extensive numerical analysis is conducted on QCADesigner to determine what number systems may be acceptable candidates for implementation. The dynamic ranges of the kink energy, polarization, and other intermediate variables are tracked in detail throughout a variety of simulations in order to determine the numerical accuracy demanded by QCADesigner. To begin with, the polarization value of a cell is typically close to 1 or -1, however it can in theory be arbitrarily small. In practice polarizations are not often reported to be any smaller than 1e-6. Kink energies are more extreme. With a large radius of effect, a kink energy of less than 1e-42 is not uncommon, and can also come in at 1e-22, representing a minimum of 20 orders of magnitude dynamic range that must be supported by the Kink energy variable. With less extreme radii of effect, kink energies of less than 1e-39 become increasingly rare, which is a value readily handled by single-precision floating point.

The first intermediate calculation value that comes up to be analyzed is `polarization_math`, specifically the code shown in listing 4.2. In this code, the `polarization_math` variable is a destination for the sum of all neighbouring cells' polarization × kink-energy products.

```
for (q = 0; q < current_cell_model->number_of_neighbours; q++)
  polarization_math += (current_cell_model->Ek[q] * ((bistable_model *)
      current_cell_model->neighbours[q]->cell_model)->polarization);
```

Listing 4.2: Polarization math accumulator

A possible issue with reducing the usable precision of variables here is cumulative rounding errors. Up to 20% of the polarization × kink-energy values could be less than 5e-42, which is the floating point half-precision threshold at which point a significant amount of accuracy is lost. Adding that many rounded values together can create a noticeable rounding error in the final total, depending on what the final total is. This error, once created, could propagate itself through all subsequent iterations and samples of the simulation. However, the nature of this accumulation operation typically involves many higher magnitude values as well. Even in double precision floating point, the least significant digit of a number can have no greater difference from the most significant digit of 16 orders of base ten magnitude. So the net effect of low-magnitude rounding errors will be numerically insignificant if not zero. In fact, analysis shows that the total contribution of low-magnitude values to the total in this accumulation is very rarely more than zero, partly due to having offsetting signs.

```
polarization_math /= (2.0 * sim_data->clock_data[cell->cell_options.
    clock].data[j]);

new_polarization =
  (polarization_math >  1000.0) ?  1 :
  (polarization_math < -1000.0) ? -1 :
  (fabs (polarization_math) < 0.001) ? polarization_math :
    polarization_math / sqrt(1 + polarization_math * polarization_math);
```

Listing 4.3: Gamma and square root simplification for polarization

Listing 4.3 shows another place in the code where dynamic range consideration is important. Here, the `polarization_math` variable is divided by the clock value, which typically ranges between 3.8e-23 and 9.8e-22, bringing the magnitude of `polarization_math` back up. Then, to get the final new polarization value,

$$\sqrt{1 + \frac{1}{2\gamma}\sum_j (E^k_{i,j} P_j)} \tag{4.1}$$

is only executed mathematically if the absolute value of $\frac{1}{2\gamma}\sum_j (E^k_{i,j} P_j)$ is greater than 0.001 and less than 1000. Therefore the squared term has no chance of being less than 1e-6, and cannot possibly become a bottleneck regarding precision. The simplification in this part of the equation also renders earlier imprecision even more insignificant.

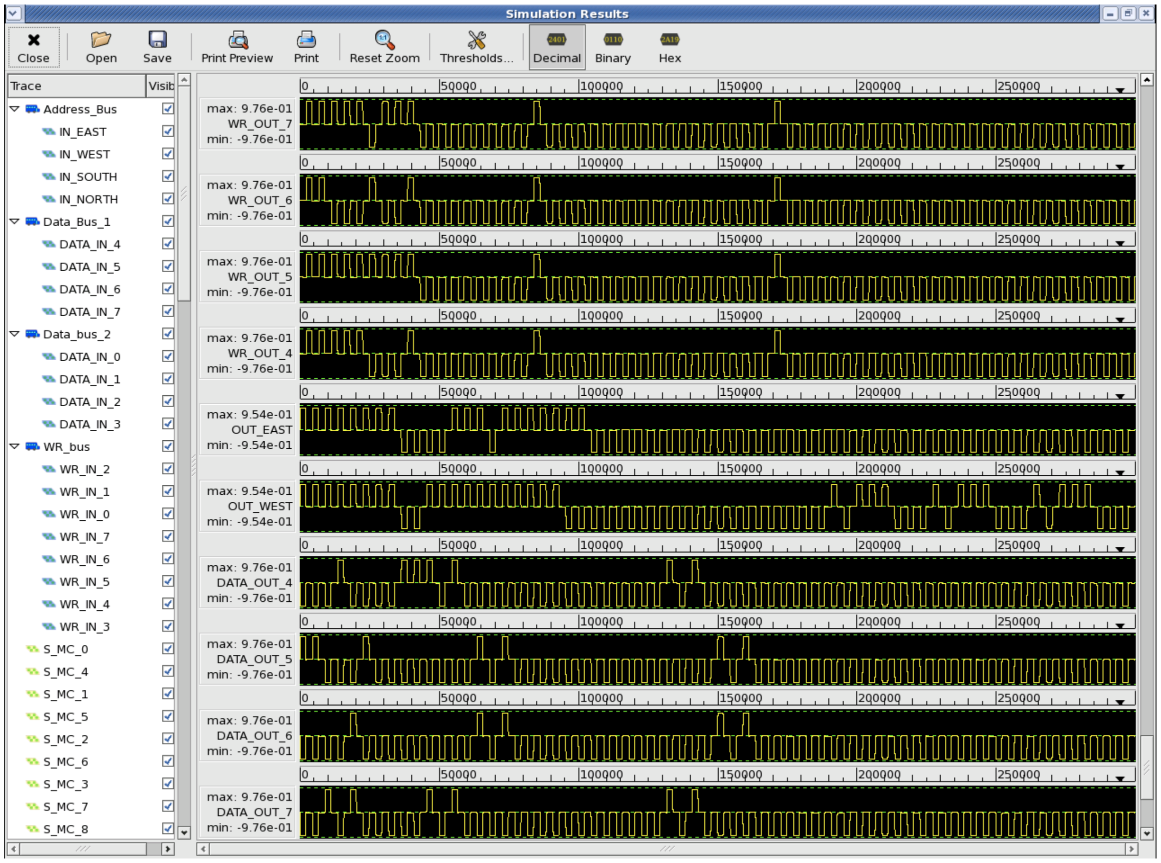

Simulation Results
Close
Open
Save
Print Preview
Print
Reset Zoom
Thresholds...
Decimal
Binary
Hex
Trace
Visib
Address_Bus
IN_EAST
IN_WEST
IN_SOUTH
IN_NORTH
Data_Bus_1
DATA_IN_4
DATA_IN_5
DATA_IN_6
DATA_IN_7
Data_bus_2
DATA_IN_0
DATA_IN_1
DATA_IN_2
DATA_IN_3
WR_bus
WR_IN_2
WR_IN_1
WR_IN_0
WR_IN_7
WR_IN_6
WR_IN_5
WR_IN_4
WR_IN_3
S_MC_0
S_MC_4
S_MC_1
S_MC_5
S_MC_2
S_MC_6
S_MC_3
S_MC_7
S_MC_8
max: 9.76e-01 WR_OUT_7 min: -9.76e-01
max: 9.76e-01 WR_OUT_6 min: -9.76e-01
max: 9.76e-01 WR_OUT_5 min: -9.76e-01
max: 9.76e-01 WR_OUT_4 min: -9.76e-01
max: 9.54e-01 OUT_EAST min: -9.54e-01
max: 9.54e-01 OUT_WEST min: -9.54e-01
max: 9.76e-01 DATA_OUT_4 min: -9.76e-01
max: 9.76e-01 DATA_OUT_5 min: -9.76e-01
max: 9.76e-01 DATA_OUT_6 min: -9.76e-01
max: 9.76e-01 DATA_OUT_7 min: -9.76e-01
0
50000
100000
150000
200000
250000

Figure 4.1: QCADesigner simulation using single precision floating point

Considering this analysis, it is clear that floating point is the best number system to use. This application is not suited to use with fixed or semi-floating number systems, because of the large dynamic ranges required, and the fact that a large portion of the operations are additions rules out logarithmic floating point. In addition, floating point arithmetic cores are readily available as IPCores in the Xilinx IDE, which greatly reduces development time. Considering the data channel for communication with the FPGA is only 32 bits wide, and the fact that double precision floating point cores are amazingly large, single precision floating point is preferable to double precision for this design.

With this in mind, QCADesigner is tested using single precision floating point in software to confirm that simulation accuracy is maintained. By simply changing variable types from `double` to `float` and

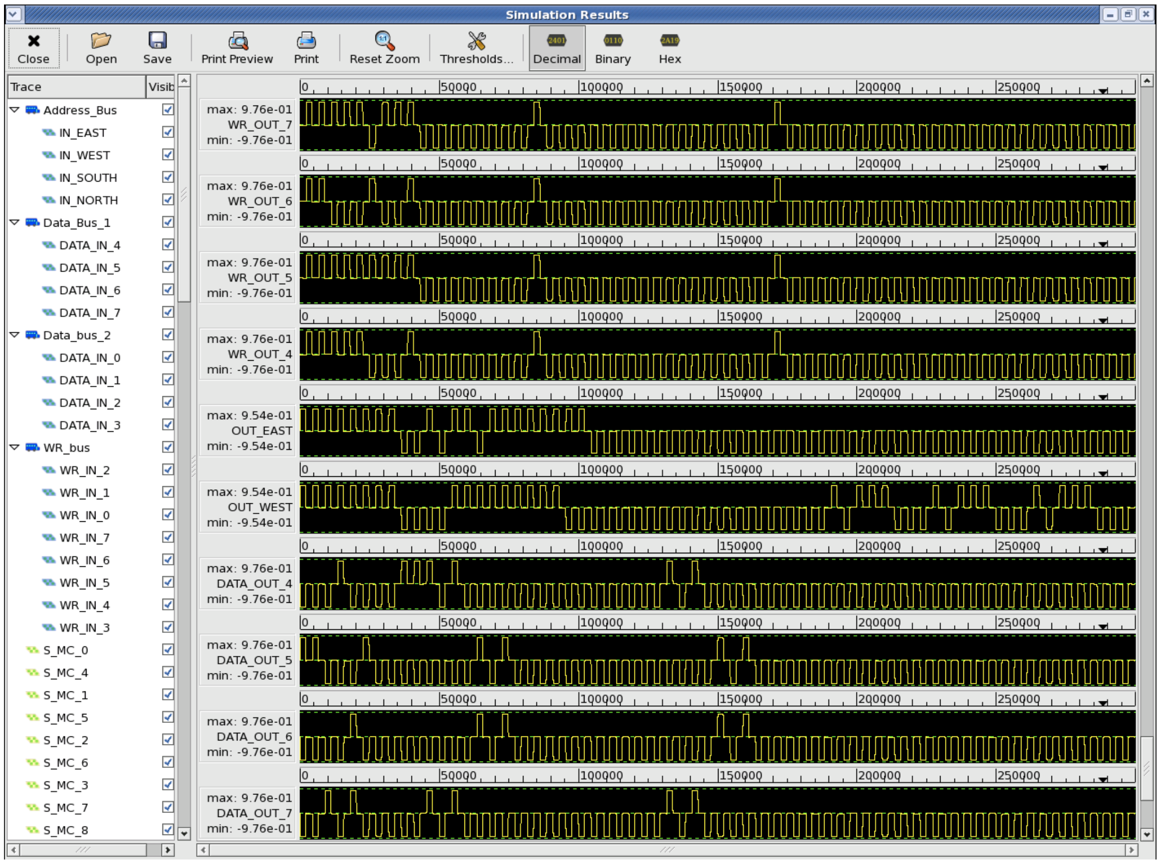

Simulation Results
Close
Open
Save
Print Preview
Print
Reset Zoom
Thresholds...
Decimal
Binary
Hex
Trace
Visib
Address_Bus
IN_EAST
IN_WEST
IN_SOUTH
IN_NORTH
Data_Bus_1
DATA_IN_4
DATA_IN_5
DATA_IN_6
DATA_IN_7
Data_bus_2
DATA_IN_0
DATA_IN_1
DATA_IN_2
DATA_IN_3
WR_bus
WR_IN_2
WR_IN_1
WR_IN_0
WR_IN_7
WR_IN_6
WR_IN_5
WR_IN_4
WR_IN_3
S_MC_0
S_MC_4
S_MC_1
S_MC_5
S_MC_2
S_MC_6
S_MC_3
S_MC_7
S_MC_8
max: 9.76e-01 WR_OUT_7 min: -9.76e-01
max: 9.76e-01 WR_OUT_6 min: -9.76e-01
max: 9.76e-01 WR_OUT_5 min: -9.76e-01
max: 9.76e-01 WR_OUT_4 min: -9.76e-01
max: 9.54e-01 OUT_EAST min: -9.54e-01
max: 9.54e-01 OUT_WEST min: -9.54e-01
max: 9.76e-01 DATA_OUT_4 min: -9.76e-01
max: 9.76e-01 DATA_OUT_5 min: -9.76e-01
max: 9.76e-01 DATA_OUT_6 min: -9.76e-01
max: 9.76e-01 DATA_OUT_7 min: -9.76e-01
0
50000
100000
150000
200000
250000

Figure 4.2: QCADesigner simulation using double precision floating point

comparing the simulation results, it is determined that single precision floating point is a safe implementation to use. Figure 4.1 shows the simulation results using single precision, and figure 4.2 shows the simulation results using double precision. It can be seen that the expected results (samples 190000 to 300000 on output OUT_WEST) are the same and correct for both. There is some minor discrepancy between the implementations for the early portion of the results, but this is inconsequential. Because the latency to the outputs is 16 cycles [45], any values that appear on the outputs before this latency has elapsed are simply noise that has been amplified, so they have no special relevance to the simulation. Closer inspection of the numerical results shows that there is far less than 1% error between double precision and single precision results, even at the end of the simulation where rounding errors would have a greater chance to accumulate to significant values. Thus single precision floating point is deemed acceptable for this application.

4.2 Hardware Architecture

To accelerate the above simulations, several architectures are considered for implementation: including streaming, grid-based parallelism, cellular automata, and instance specific design. A streaming architecture takes a constant stream of data in, processes it, and sends a stream out, achieving application speedup by means of parallelization in a computational pipeline. Grid-based parallelism would break the problem into spatial regions and compute results within each grid, achieving speedup through spatial parallelism. Cellular automata (unrelated to QCA) refers to a very fine-grained form of grid parallelism in which one computational core corresponds to one element that needs to be computed (*i.e.* one core per QCA cell), and these cores are laid out in a large repeating structure. Instance specific design uses a basic starting design and procedurally generates a large design to be implemented on the FPGA, unique to the specific circuit being simulated [16]. Cellular automata would be the ideal implementation to achieve maximum speedup, but the floating point mathematics are too complex to support a large number of cores, so it must be discounted. The Wildcard 4 libraries do not support any means of procedural circuit generation, so instance specific design is not feasible. A streaming architecture, possibly one with multiple pipelines to increase parallelization as much as possible, is decided on. This also allows easy integration with the software.

4.2.1 Proof of Concept

Before designing a final general purpose version of the hardware, a small proof of concept version is created, dubbed "Minisim." It is an instance specific design, simulating a QCA majority gate in one clock zone using a streaming architecture. It is meant to confirm that a QCA circuit can be simulated accurately using single precision floating point on hardware in a reasonable time while using a reasonable amount of resources, and to gauge how much of the simulation can be put on hardware. Initially, the design attempts to run with a starting point of just cell positions, dynamically calculating the kink energies to be used in the simulation. The additional hardware required to calculate the kink energies proves to be excessive, and because kink energy calculation only contributes a small fraction of execution time, the concept is abandoned.

Minisim's final design involves hard-coded kink energies implying the structure of a majority gate, with the input values and a stream of gamma values coming from software, and a stream of output values going back to the software. This hardware/software execution proceeds to completion in the same perceived amount of time as the software implementation, even for long simulations. The numerical results match software to within 0.001% for steady state values, but deviate from each other more during transitional values. This greater deviation is only in terms of relative error, the absolute error is still quite small, on the order of thousandths. Plots of the hardware and software results of minisim are shown in Figures 4.3 and 4.4 respectively. The synthesis report of Minisim shows that 33% of the board's total available resources are used by the hardware, giving initial indications that three or four computational cores could be included on the FPGA simultaneously. However, these are still rather large cores, so any sort of granular grid computing structure is out of the question.

It was determined earlier in Section 3.6 that the total set of kink energy data for a 22,000 cell circuit with a radius of effect of 500 nm uses about 35 MB of the Wildcard's available 128 MB of DRAM. Using this same radius of effect, circuits with 50,000 or even 80,000 cells can be easily accommodated, or larger if the radius of effect is reduced. And because the kink energies do not change throughout the simulation, they can be loaded once and referenced later. All the kink energies are therefore stored on the Wildcard DRAM rather the host memory when

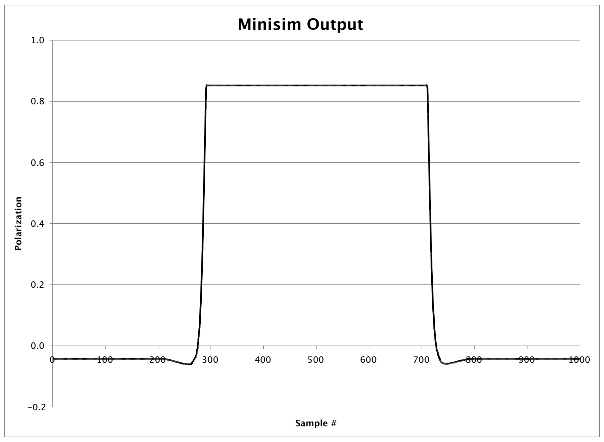


Figure 4.3: Hardware results from minisim test

they are calculated. There is no need for them to be in host memory when all further calculations are done in hardware, so extra transfers are avoided. Furthermore, they are stored in one solid contiguous block with no form of demarkation between cells. Instead, reliance is put upon having a consistent list of cells that does not change its order, and whose neighbour counts are known.

Therefore, cell randomization must be abandoned. This is of no consequence however, because the hardware will calculate an entire iteration without replacing any values in memory, thus completely avoiding the numerical problem the randomization was trying to reduce. Recall from Section 4.1.1 that cell list randomization is done to minimize the effect of the numerical problem by shuffling the order in which cell polarizations are calculated.

4.2.2 DRAM Interfacing

Using the DRAM to store kink energies is not completely without issue, however. First of all, the DRAM has a latency of 173 cycles between getting a read request and delivering the requested data, so steps must be taken to ensure this latency is incurred as infrequently as possible. Second, each address refers to a 128 bit word whereas the data being

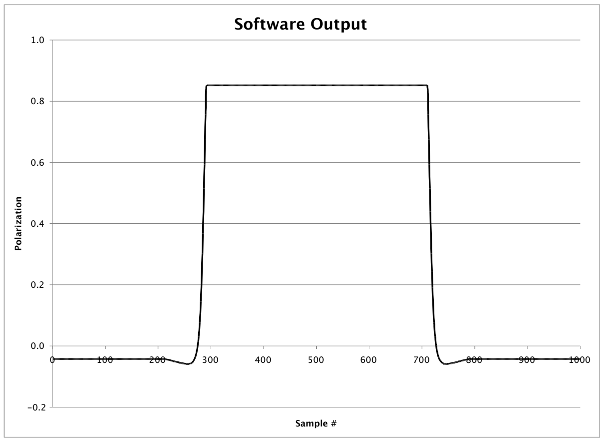


Figure 4.4: Software results from minisim test

stored is only 32 bits wide, so a special interface must be created to write and read data 32 bits at a time.

The DRAM's "startup latency" of 173 cycles applies when a new set of read operations is started. Additional latency applies when seeking new non-sequential addresses, but after the initial latency has passed, each further sequential address read is delivered on the immediate next cycle. If a constant stream of reads can be sustained, the entire contents of DRAM can be delivered with the only latency cost being the initial 173 cycles. The storage method of the kink energies in the design actually allows for this to take place independent of the radius of effect, because they are stored in the same order as they are processed. However, the mathematical core demands data at a much slower rate than it is delivered from DRAM, so the DRAM output must go through a First-In-First-Out queue (FIFO) which can accumulate data and deliver it when requested. This FIFO and the DRAM are carefully controlled to ensure no over or underflow occur, so that no data is lost or miscalculated. It is 256 words large, and the state machine will only send a read request to the DRAM if less than 50 values are in the FIFO (indicated by `dram_fifo_prog_empty`), allowing the next 173 "cool down" words to be received without overflow. Requesting a new read when the FIFO data count is back down to 50, allows time for the requested data to get into the FIFO before underflow occurs.

```
    port map (
      din => dram_data_in ,
      rd_clk => i_clock ,
      rd_en => dram_fifo_rd , --controlled from state machine when data
          is needed
      rst => dram_fifo_clr , --controlled in the state machine so empty
          flag resets
      wr_clk => p_clock ,
      wr_en => dram_data_in_valid ,
      dout => dram_fifo_out , --goes to DRAM output buffer
      empty => dram_fifo_empty ,
      full => dram_fifo_full ,
      prog_empty => dram_fifo_prog_empty); --set to turn on when less
          than 50 data are present
```

Listing 4.4: DRAM output FIFO

The structures used to mange the data flow into and out of DRAM are shown in their VHDL form in listing 4.5. They refer to `dram_addr(1 downto 0)`, which are two address line bits that are added onto the normal 23 address line bits of the DRAM module. They are used to refer to the 32 bit parts of the 128 bit DRAM word individually. This part of the address line is controlled independently from the other 23 bits so that new DRAM addresses can continue being read at their own pace independent of which value is actually demanded for use on the output. The DRAM output manager is quite simple, with the low address bits essentially controlling a multiplexer between the DRAM's output FIFO and the main math core.

The DRAM data input control is slightly more complex. Since data is written to DRAM in larger portions than it is received, it must be stored in parts to a register whose value is written every fourth address. An override (`dram_super_write`) is built into the logic controlling the write line to allow the final set of data to be written if less than four 32 bit words occur in the last set.

```
--DRAM output buffer , goes directly to p_math core
Ek_in <= dram_fifo_out(127 downto 96) when (dram_addr(1 downto 0) =
    "11") else
```

```
      dram_fifo_out(95 downto 64) when (dram_addr(1 downto 0) = "10")
          else
      dram_fifo_out(63 downto 32) when (dram_addr(1 downto 0) = "01")
          else
      dram_fifo_out(31 downto 0) when (dram_addr(1 downto 0) = "00")
          else
      (others => '0');

--dram input buffer
DRAM_input_buffer: process(i_clock, reset)
begin
  if(reset = '1') then
    dram_data_out <= (others => '0');
    dram_write <= '0';
  else
    if rising_edge(i_clock) then
      if(dram_buf_write = '1') then
        case dram_addr(1 downto 0) is
        when "00" =>
          dram_data_out(31 downto 0) <= in_fifo_out;
        when "01" =>
          dram_data_out(63 downto 32) <= in_fifo_out;
        when "10" =>
          dram_data_out(95 downto 64) <= in_fifo_out;
        when "11" =>
          dram_data_out(127 downto 96) <= in_fifo_out;
        when others =>
          dram_data_out <= dram_data_out;
        end case;

        if((dram_addr(1 downto 0) = "11") or (dram_super_write = '1'))
            then
          dram_write <= '1';
        else
          dram_write <= '0';
        end if;
      else
        dram_write <= '0';
        dram_data_out <= dram_data_out;
      end if;
    end if;
  end if;
end process;
```

Listing 4.5: DRAM buffer control

4.2.3 Other Interfacing Issues

Other interfacing concerns are involved with sending the polarization and gamma data to the FPGA, and reporting back the new polarization results from the FPGA. Methods of efficient transfer must be devised that are able to meet a large range of parameters. Most importantly, the hardware must be able to support any reasonable number of neighbours (effective radius) and all reasonable circuit sizes.

The input into the FPGA is more restrictive than the output from the FPGA, because data comes in faster than it is used, so data overflow is a risk, whereas on the output, data can be written to host memory as soon as it is calculated with less concern of overflow, because DMA writes are more explicitly controlled in hardware. Care must be taken in software and hardware to avoid any read-before-write errors that would corrupt the results of a calculation, while also carrying out necessary data copying on the host in parallel with data computation on the hardware.

Another key communication issue is interfacing between different clock domains. The data transfer bus runs at 40 MHz, while the arithmetic core runs at 140 MHz. Therefore, dual-clock FIFOs are used on both the input and output, which are generated using the IPCoreGen tool in Xilinx ISE. They are also specifically sized and controlled to prevent over and underflow while meeting the requirements of the application.

The input FIFO is designed to have 1024 locations, which will allow up to 1024 neighbours per cell. This is large enough to accommodate a radius of effect of well over 500 nm, in which no cell in the CLB has more than 650 neighbours. Should it be determined that support for more neighbours is desired, it is easy to create a new, larger FIFO. The DMA buffer on the software side is allocated at a matching fixed size.

For the hardware's output, the FIFO does not need to be as large as there are cells in the circuit, it only needs to be large enough to prevent any over or underflow problems that may arise due to control signals having laggy actuation. 256 locations is deemed an appropriate size for this, and is tested to confirm functionality. The DMA buffer for the results in software is allocated to be able to receive as many cells as are calculated, which is *not* equal to the total number of cells in the circuit; it only needs to accommodate the cells that are not fixed or inputs, as seen in lines 93 and 94 of listing B.1.

With the architectures in place, the communication algorithm must be created. In order to send kink energy or polarization data to the FPGA without garbage data existing at the end of any block transfers, the FPGA must know how many words to request in the DMA transfer. This is accomplished by sending the number of neighbours to the FPGA using a register transfer, which is slower than DMA for large transactions, but better suited to small ones. For transferring polarizations, the gamma value for the cell to be calculated is also part of the register transaction. At the end of a full phase (*i.e.* all kink energies have been transferred, or all cell data has been transferred for an iteration), a special totem value is sent to the hardware instead of the normal data, so the hardware knows to proceed with the next phase of operation. The control hardware for this is simpler than including cell number counters. The hardware sends interrupts back to the software to indicate when all data has been read or calculated to ensure that no read-before-write errors occur.

Due to the way polarization data is stored (not in solid blocks), each neighbour polarization value must be copied individually to the DMA transfer buffer, as in line 382 of listing B.1. This operation takes the place of where the software would calculate the accumulation of `polarization_math`. The software waits for the interrupt signifying the completion of a cell's calculation *after* this memory transfer is complete to parallelize the workload between host and hardware.

4.2.4 The Arithmetic Core

The main mathematical core is where all of the computation happens. It consists of a pipeline of floating point math cores as shown in Figure 4.5. The floating point cores are all generated using the IPCoreGen tool

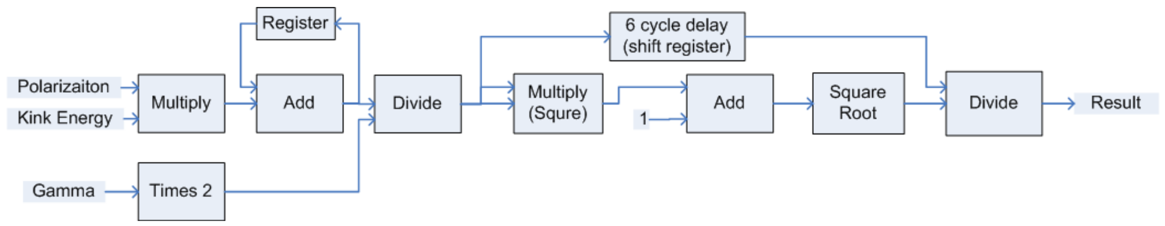


Figure 4.5: Block diagram of the polarization math pipeline

in the Xilinx ISE. The VHDL code for this core is shown in Appendix D.

The control signals on each floating point core are for the most part connected in such a way that the next core in the line is automatically started when the data on the previous core is ready. Controlling the multiplier and adder together to form the accumulator in the first stage is slightly more complex, requiring precise timing and counting of inputs. The delay line shown is simply a shift register in place to sync up data propagation through multiple branches of one pipeline. Gamma being multiplied by a constant value of two is accomplished by left shifting the exponent of the floating point value to save on hardware cost. It works without issue because the value in gamma is never large enough where a 1 would be shifted out of the MSB. The circuit's stability is not calculated in this core, or in fact anywhere else on the hardware. Because all polarizations are delivered back to the host platform for each iteration, this comparatively simple operation can be done in software with little extra cost.

The floating point cores are all tuned to have minimum latency, while guaranteeing a certain operating clock speed. Of special importance is the adder's latency in the accumulator, because this is the critical factor of how fast a complete cell can be calculated, and how efficiently the full pipeline may be used. Lower latency for a floating point unit means that more levels of combinational logic must be waited on between stages in the computational pipeline, so more delay is inherent per pipeline stage. Increased latency means the clock may run faster, but more hardware is required to register more stages of the pipeline. So, the least amount of latency is desired while still maintaining a minimum clock rate.

There are two clocks on the Wildcard, one is fixed at 40 MHz and governs data bus transactions. The other runs at a user-specified speed up to 240 MHz, and in this case must be set to at least 130 MHz for the DRAM to function properly. Calculating the total time required to calculate the polarization of one cell with ten neighbours shows that there is little difference between latency configurations, less than 50 ns difference out of a total around 750 ns. A latency configuration is selected which enables an optimum balance between speed and stability at 140 MHz, as indicated by the Xilinx place and route tool.

At this point, the duration of a complete QCA circuit simulation using this core can be calculated. It is based on the simulation of a CLB (22,558 cells) with a radius of effect of 50 nm (average number of neighbours per cell is 5), 300,000 samples, and a convergence tolerance of 0.001 (average reiteration rate per sample of 1.7). Keeping in mind, this calculation assumes maximum pipeline usage and no overhead once the calculation is started, it represents a theoretical minimum time. The combined latency of the accumulator stage is the only real per-cell calculation cost, because a new cell can be accumulated as the previous cell flows down the pipeline. Similarly, the only real per-neighbour cost in the accumulator is in the adder, as the multiplier only contributes its own additional latency once before any addition can start. The accumulator latency is equivalent to the multiplier latency, plus the adder latency for each neighbour, plus one cycle to account for informing the core that a cell is finished with input. This value gets multiplied by the number of cells to get the number of cycles for one iteration, added with the number of cycles for the last part of the pipeline to complete on the last cell, then multiplied by the number of iterations per sample, then multiplied by the number of samples. This total number of cycles for a simulation is divided by the clock speed to get the elapsed time.

$$\frac{((6+6*5+1)*22558+14+4+5+14+14)*1.7*300000}{140\text{ MHz}} = 3040.68\text{ s} \quad (4.2)$$

This result is on par with the current execution time in software. Although it assumes perfect pipeline usage and no overhead, several cores can fit on the FPGA simultaneously. The execution time could be reduced by a linear factor equivalent to the number of cores present, thus compensating for non-idealities and potentially achieving the desired speedup.

4.2.5 Operation and Synthesis

Hardware operation is selected via a checkbox in the simulation setup window. The block diagram of the system is shown in Figure 4.6, illustrating the basic connections between major components of the design. The combined hardware/software system operates as shown in the flowchart in Figure 4.7.

There is some overhead involved in the frequent DMA transfers, but any such overhead's contributing factor is reduced as the number of neighbours in a circuit increases (*i.e.* as the size of each transfer increases). This hardware/software system is shown to work in simulation using the simplified host software simulator prescribed by the Annapolis tools. The Xilinx place and route report shows that the main arithmetic core uses 1834 slices (3037 LUTs, or 11% of the Virtex 4) of the available hardware resources, and the rest of the control and interface hardware consumes 1809 slices (1608 LUTs, or 12% of the Virtex 4), indicating that up to 6 arithmetic cores may be used simultaneously.

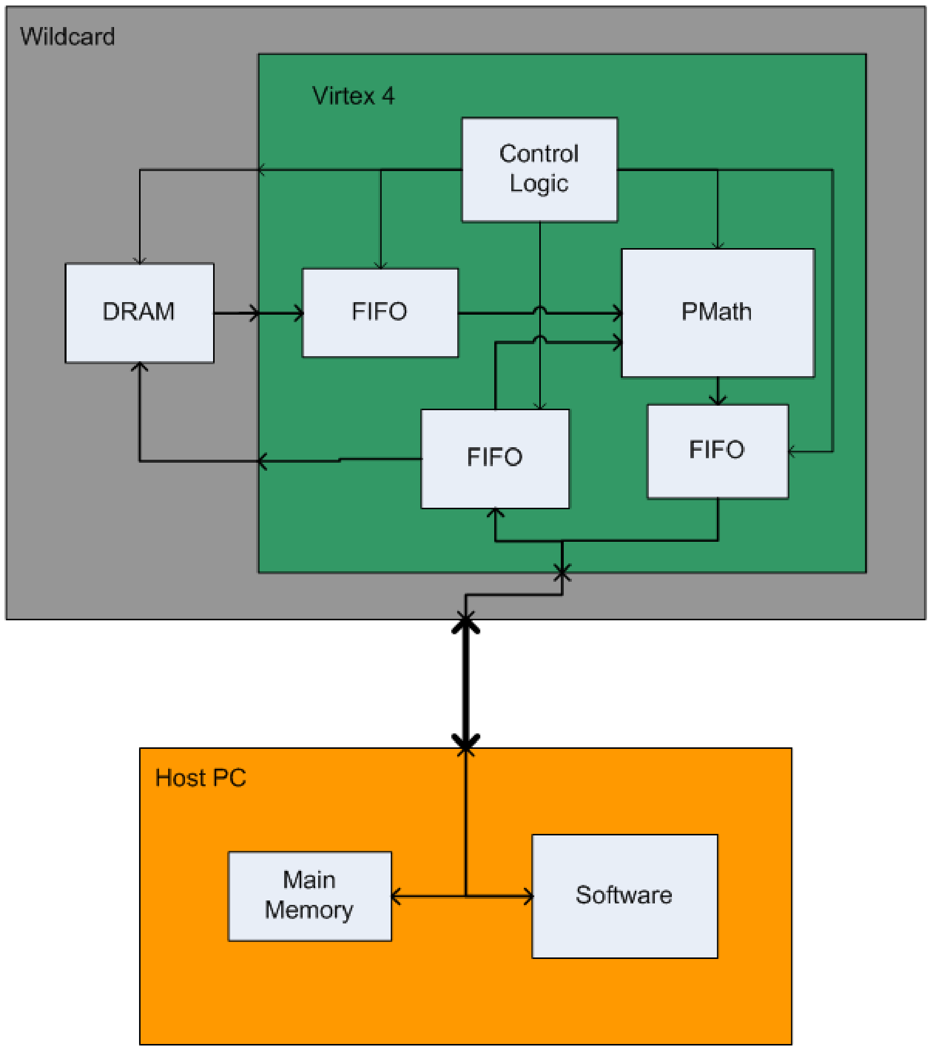


Figure 4.6: Block diagram of high level system architecture

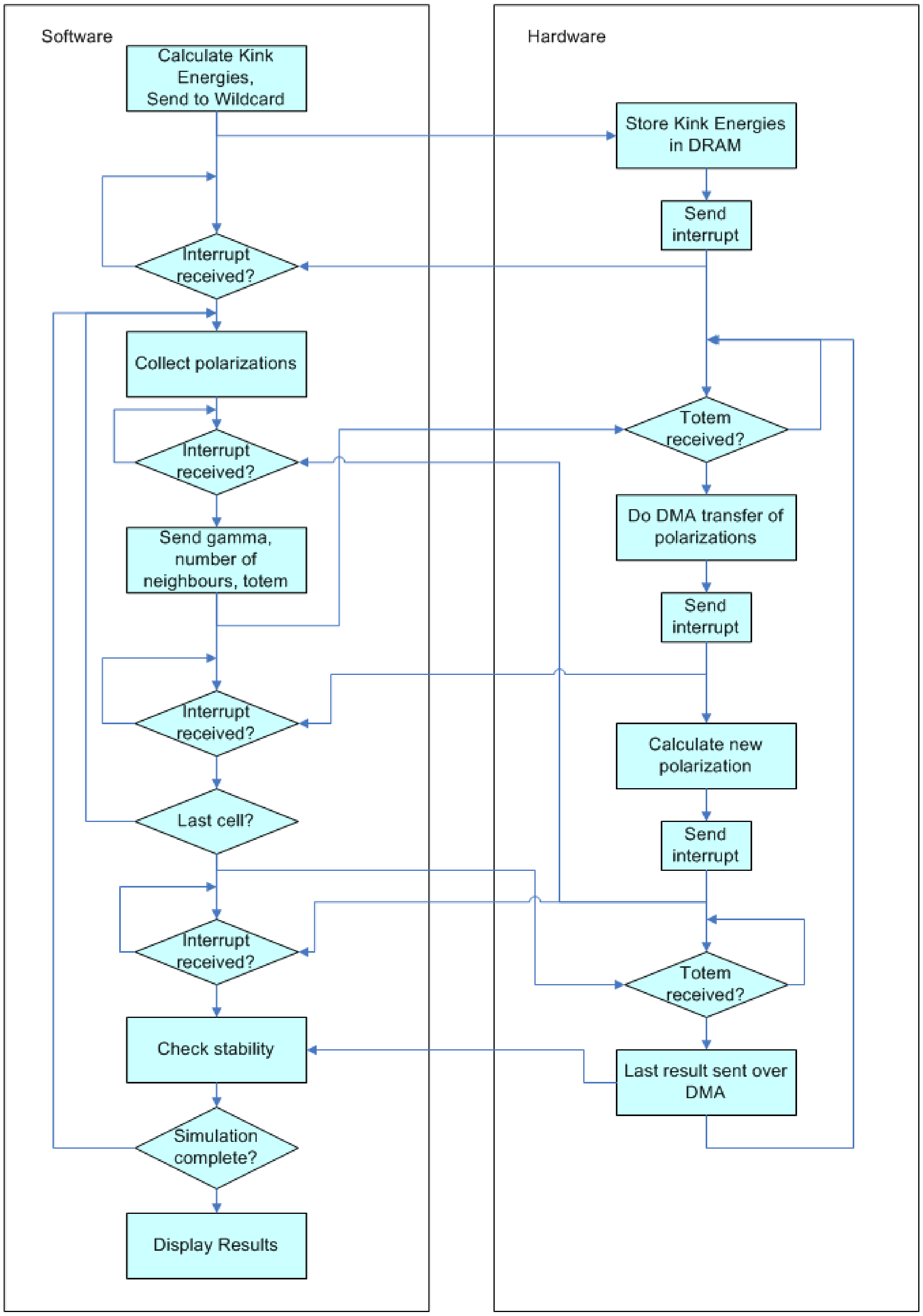


Figure 4.7: Flowchart for the operation of the combined hardware/software system

Chapter 5

Results and Discussion

As mentioned in Section 4.2.4, the predicted execution speed of the hardware is expected to be on the same order of magnitude as the software execution speed. However, upon actually using and testing the hardware, the observed execution speed is several orders of magnitude slower. An 80,000 sample simulation of a majority gate with an 8 cell wire on the output takes a reported 1 second to run to completion in software, but 157 seconds in hardware; a very surprising result. Despite this, the hardware is still capable of delivering an accurate result for small circuits. Figure 5.1 shows a simple majority gate simulation running in software and Figure 5.2 shows the same simulation done in hardware, clearly illustrating the same result being obtained in both cases, with some minor numerical discrepancy due to switching the floating point precision level. This result verifies the basic functionality of the floating point pipeline developed. This and the analysis showing its potential speed are encouraging results in terms of a proof of concept design on which to base other designs. Additionally, some notable improvements have been made to the QCADesigner software during the course of this work.

5.1 Cause of Slowdown

The first effort to try to determine why the hardware simulation takes so long is to apply gprof and see if any of the new functions being applied in the main simulation loop such as `IntWait` are taking an excessive amount of time, which would indicate the hardware rather

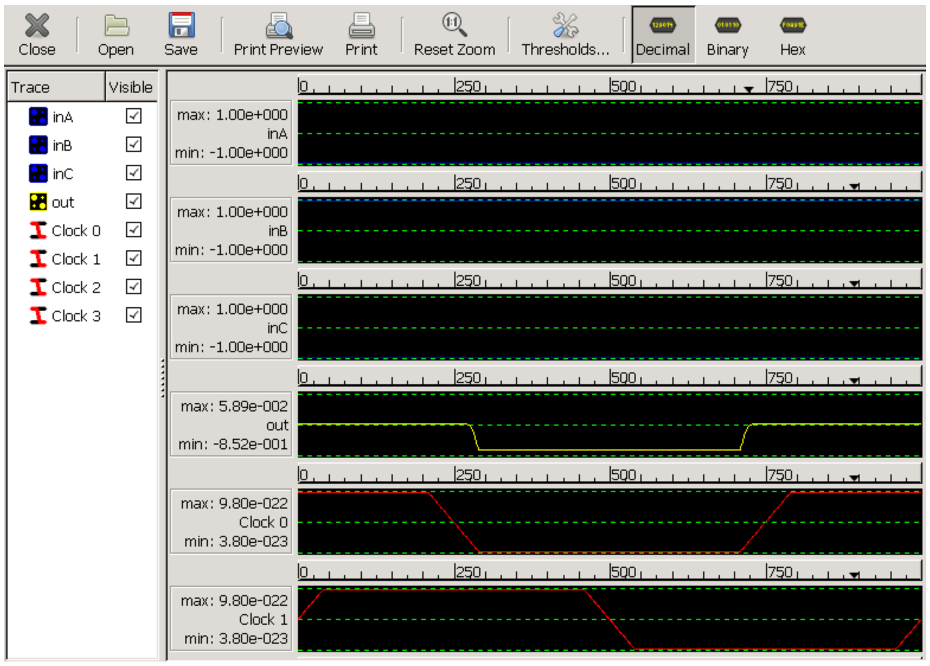


Figure 5.1: Result for software run of a majority gate

than the software is taking too long to finish the computation. This doesn't yield any useful insight. The next step is to manually instrument the code with the functions `QueryPerformanceFrequency` and `QueryPerformanceCounter`. `QueryPerformanceFrequency` gives the number of processor ticks that occur every second (*i.e.* the clock speed), and `QueryPerformanceCounter` gives the number of processor ticks that have occurred since startup. Using these functions, is can be determined precisely how much time is consumed executing certain sections of the code.

To begin with, it is determined that one cell in software takes roughly 600 ns to compute, with some variance depending on the specific run. In hardware mode, the calculation time for a single cell is between 18 and 45 μs. Upon further analysis, this rather alarming increase in execution time is determined to be caused by the interrupt reset procedures, of which there are two, taking 10 to 20 μs each (usually about 14 μs). The repeated interrupt reset, shown in listing 5.1, is a required procedure given the architecture in use, because every time an interrupt is used, it must be reset before being used again. The only way to avoid using

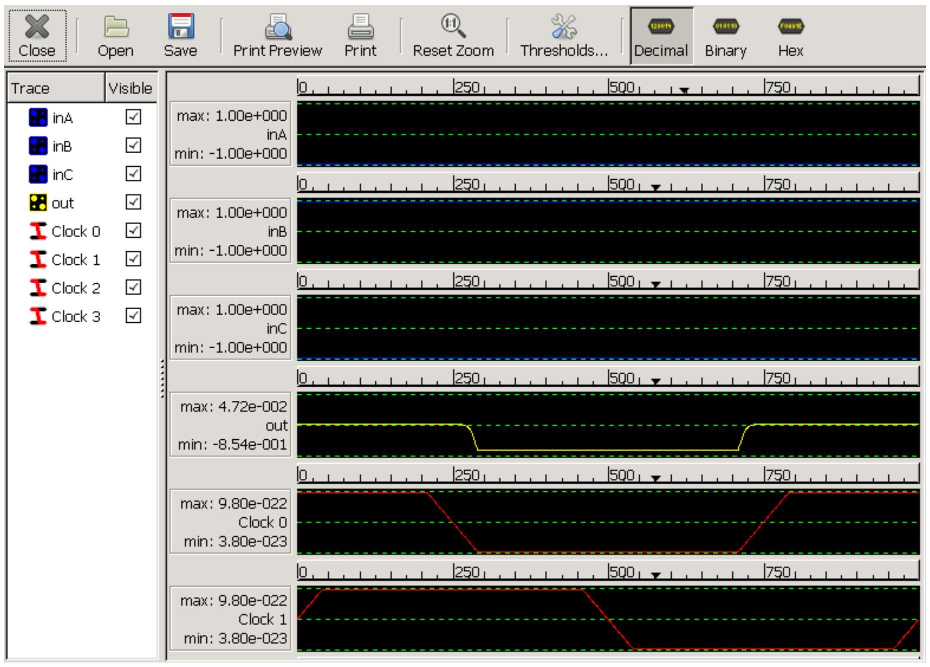


Figure 5.2: Result for a hardware run of a majority gate

interrupts with this architecture would be to wait a fixed amount of time for hardware operations to complete, but there is no good way to estimate the completion time dynamically or wait for such brief precise periods. There also is no way for the software to detect the logical value assigned to the interrupt bit by the hardware, because the value is latched at 1 when triggered until reset by the software.

```
rc=WC_IntEnable(TestInfo.DeviceNum, FALSE); //reset interrupt
rc=WC_IntReset(TestInfo.DeviceNum);
rc=WC_IntEnable(TestInfo.DeviceNum, TRUE);
```

Listing 5.1: Interrupt reset procedure

Due to the large variance in these measurements, it is impossible to say exactly how long the calculation would take without these interrupt resets. An estimate based on the difference between total execution time and total interrupt reset time might come between 1 and 5 μs, low enough to achieve speedup with six pipelines. To be clear, it is just the interrupt reset that takes that long; the interrupt wait function itself takes a much shorter amount of time, indicating that the hardware calculation is proceeding apace. This is consistent with the determination in Section 4.2.4 that the main arithmetic core should be able to calculate the results at the same rate as the software. If better interfacing and control hardware can be developed, speedup may be achieved when using multiple cores in parallel.

A small test program is created to test in simpler terms what difference the interrupt reset has on execution time. Two separate data transfers are initiated in sequence, with an interrupt between them, and another interrupt at the end, requiring a reset in between. With the reset in place, the program takes 40 μs to execute, but simply waiting the short transfer out without the reset results in a 1.5 μs. There was no indication in any manual or simulation that interrupt resets would take so long.

5.2 Hardware Reliability

In addition to the extra long execution time, there are significant reliability problems with the control hardware. Any circuit much larger than 6 cells encounters significant difficulty in proceeding to the end of simulation. One problem appears to be related to bad data being reported back to the host, which could be due to problems on either the input or output sides of the hardware. This causes the simulation to constantly reiterate until it hits the reiteration limit, and results in a flatlined simulation result. Another problem is related to the hardware and software getting out of sync, causing interrupts to timeout, again resulting in flatlined results. The inconsistent nature of these issues suggests a problem related to timing, possibly a clock running too fast. However, reconfiguring the hardware to run on the 40 MHz clock rather than 140 MHz does not inspire any changes.

Further debugging is quite difficult. Because the programming interface is on the PCI bus rather than a JTAG interface, it is impossible to use the Xilinx ChipScope hardware analyzer to examine signal timelines. The closed form factor of the card also prevents examining any signals using an oscilloscope or logic analyzer. The simulation works, so that is not a debugging avenue to pursue. The only way to debug the hardware is to request debug outputs from the hardware. But the times these data can be requested without interfering with the standard operation of the hardware are very limited, making it nigh impossible to get comprehensive debugging information. Because of the great difficulty in debugging and the timing analysis showing a very low performance ceiling for the given architecture, further development is abandoned.

5.3 QCADesigner Enhancements

The results of this work are not a complete write-off, however. Some notable improvements to the QCADesigner software have been made. First, an error was found in the stability checking method. The version in current distribution has the following for a stability check.

```
1 // If any cells polarization has changed beyond this threshold
2 // then the entire circuit is assumed to have not converged.
3 stable = (fabs (new_polarization - old_polarization) <= tolerance) ;
```

Listing 5.2: Original stability check

The comment suggests that the sample should reiterate if *any* cell's polarization changes goes beyond the threshold, but the way the code is written, it only reiterates if the *last* cell's polarization changes beyond the threshold limit. The fix is very simple, changing line three of listing 5.2 to listing 5.3, thus forcing any unstable cell to cause all future stability checks to fail. The fix causes simulations to take roughly four times as long, as expected, because samples reiterate more often. The fix is also confirmed as needed by Konrad Walus, the QCADesigner lead.

```
stable = (fabs (new_polarization - old_polarization) <= tolerance) &&
    stable ;
```

Listing 5.3: Fixed stability check

The other improvement made to QCADesigner is an optimization made to the `select_cells_in_radius` function. In the current version of QCADesigner, a cell's neighbours are found by using a complex conditional statement shown in listing 5.4

```
if (sqrt ((QCAD_DESIGN_OBJECT(sorted_cells[i][j])->x -
    QCAD_DESIGN_OBJECT(cell)->x) *
  (QCAD_DESIGN_OBJECT(sorted_cells[i][j])->x - QCAD_DESIGN_OBJECT(cell)
     ->x) +
  (QCAD_DESIGN_OBJECT(sorted_cells[i][j])->y - QCAD_DESIGN_OBJECT(cell)
     ->y) *
  (QCAD_DESIGN_OBJECT(sorted_cells[i][j])->y - QCAD_DESIGN_OBJECT(cell)
     ->y) +
  ((double) ABS(the_cells_layer-i)*layer_separation) *
  ((double) ABS(the_cells_layer-i)*layer_separation)) < world_radius)
    number_of_selected_cells++;
```

Listing 5.4: select_cells_in_radius original

This condition contains within it several arithmetic operations, including a square root function, which take time to execute. These operations are computing a distance between cells, which takes time and does not need to be precisely calculated each time to achieve the desired result. A precondition (shown in listing 5.5) can be added to test just the horizontal and vertical components of the distance, and if either is greater than the radius of effect, the detailed distance calculation can be avoided.

```
if( ((QCAD_DESIGN_OBJECT(sorted_cells[i][j])->x - QCAD_DESIGN_OBJECT(
    cell)->x) < world_radius) && ((QCAD_DESIGN_OBJECT(sorted_cells[i][j
    ])->y - QCAD_DESIGN_OBJECT(cell)->y) < world_radius) )
```

Listing 5.5: select_cells_in_radius optimization

This modification reduces initialization time for large circuits by 37%.

5.4 Design Alternatives

Several options exist for pursuing alternate solutions to this design problem. A variety of other platforms could be used which are larger and better equipped, or different architectures could be explored.

5.4.1 New Architecture

One possible design architecture that could be implemented for future research would store all cells' polarizations on the Wildcard's SRAM and compute the entire length of the simulation in hardware, rather than constantly exchanging polarization data and using repeated interrupts. This would eliminate almost all communication overhead, including most importantly the interrupt resets.

The storage structure of this new architecture would store all cell polarizations in the form of a spatial grid in SRAM, where every 36 bit word in SRAM will store the polarization and clock zone (34 bits of data) for the cell at a particular location. The cell's location would directly correspond to the address in SRAM, and zeros would be in SRAM for locations where there is no cell. The location encoding would work by each sequential address representing the cell with its location at the next 10 nm increment. QCADesigner can only place cells at 10 nm increments (10 nm is half of a cell's length), so no more location precision than this is needed. A new row is simply defined by counting off the number of cells in the previous row. Finding and retrieving neighbour polarization data would be accomplished by simple rectangular address arithmetic; a given radius of effect would be converted to an integer number of half-cell distances, and cells within a square rather than a circle would be counted to simplify the mathematics a little cost to accuracy. There is enough SRAM on the Wildcard 4 to hold a circuit 4500 nm $\times$ 4500 nm, which is enough for 4 CLBs. Most of the data in SRAM would be zeros, so larger circuits could be stored if some form of compression can be developed.

Kink energies would be stored in much the same way, but the delivery of gamma values would have to be modified to be delivered in bulk rather than per cell, and results would still be sent back to the host as they are calculated using DMA. Extra software development is required to sort the cells into the form which they will be stored in, and to change the calculation of kink energies to consider the new square of effect.

5.4.2 Other Platforms

Other hardware platforms may also be used to great effect. For example, a top of the line Virtex 6 card has over 85,000 logic slices, each containing four 6-input LUTs [53]. The Virtex 4 used in this thesis only has two 4-input LUTs for each of just 15,000 slices. A Virtex 6 also has PCI Express integration, which would facilitate communication with a host PC, as well as a faster clock speed. With the much larger amount of resources, a design based on a Virtex 6 may be able to support up to 100 floating point pipelines in parallel, moving one step closing to being able to implement a cellular automata style computational architecture.

Many hardware platforms also offer on-board processors, or have the ability to generate soft-core processors on the FPGA. This type of hardware may allow the creation of a hardware/software design with much more efficient communication between the hardware and software kernels.

Chapter 6

Conclusion and Future Work

The work done in this thesis has shown the potential for a hardware coprocessor to accelerate the simulations needed in QCA research. Although the system designed here did not succeed in speeding up the execution of QCA simulations, this has more to do with interrupt resets taking much longer than expected than any other aspect of the design. It is possible to achieve the desired speedup using the arithmetic core presented here if data can be fed to it fast enough. Avoiding the long interrupt resets, either by avoiding interrupts altogether or moving to a different hardware which resets them more quickly, could help provide the necessary throughput. An implementation of this application that achieves speedup would be a significant achievement for FPGAs, showing that floating point applications may be sped up on easily available reconfigurable hardware using hardware/software design. This would be attractive to the realm of scientific research because it combines floating point accuracy with the relative ease of use and development of hardware/software design.

The implementation of this alternate architecture is beyond the scope of this thesis, but is a prime candidate for future research. Alternate hardware platforms may also be sought that have better interrupt routines or larger FPGAs, and the QCADesigner coherence vector simulation engine may also be accelerated. The vein of this research may also be applied to broader applications such as the physical side of QCA research, advancing more than just QCA architecture development.

Bibliography

[1] Wolfgang Arden, Patrick Cogez, Mart Graef, Reinhard Mahnkopf, Hidemi Ishiuchi, Toshihiko Osada, JooTae Moon, JaeSung Roh, Carlos H Diaz, Burn Lin, Pushkar Apte, Bob Doering, and Paolo Gargini. 2009 international technology roadmap for semiconductors. Technical report, IEEE Computer Society, 2009.

[2] Gary H. Bernstein. Quantum-dot cellular automata: computing by field polarization. In *DAC '03: Proceedings of the 40th annual Design Automation Conference*, pages 268–273. ACM, 2003.

[3] E. P. Blair and C. S. Lent. Quantum-dot cellular automata: an architecture for molecular computing. In *Simulation of Semiconductor Processes and Devices, 2003. SISPAD 2003. International Conference on*, pages 14–18, 2003.

[4] Enrique Blair, Eric Yost, and Craig Lent. Power dissipation in clocking wires for clocked molecular quantum-dot cellular automata. *Journal of Computational Electronics*, 9(1):49–55, March 2010.

[5] B. Bosi, G. Bois, and Y. Savaria. Reconfigurable pipelined 2-D convolvers for fast digital signal processing. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 7(3):299–308, September 1999.

[6] Owen Callanan, Andy Nisbet, Emre Ozer, James Sexton, and David Gregg. FPGA implementation of a lattice quantum chromodynamics algorithm using logarithmic arithmetic. In *IPDPS '05: Proceedings of the 19th IEEE International Parallel and Distributed Processing Symposium (IPDPS'05) - Workshop 3*, page 146.2, Washington, DC, USA, 2005. IEEE Computer Society.

[7] R. P. Cowburn and M. E. Welland. Room temperature magnetic quantum cellular automata. *Science*, 287(5457):1466–1468, 2000.

[8] Mike Cowlinshaw, editor. *754-2008 IEEE Standard for Floating-Point Arithmetic*. IEEE Computer Society, 445 Hoes Lane, P, 2008.

[9] M. Crocker, Xiaobo Sharon Hu, M. Niemier, Minjun Yan, and G. Bernstein. PLAs in quantum-dot cellular automata. *IEEE Trans. Nanotechnol.*, 7(3):376–386, May 2008.

[10] Yong Dou, S. Vassiliadis, G. K. Kuzmanov, and G. N. Gaydadjiev. 64-bit floating-point FPGA matrix multiplication. In *Proceedings of the 2005 ACM/SIGDA 13th international symposium on Field-programmable gate arrays*, FPGA '05, pages 86–95, New York, NY, USA, 2005. ACM.

[11] E. N. Ganesh, Lal Kishore, and M. J. S. Rangachar. Study of complex gate structures in quantum cellular automata technology for FPGA applications. In *Proceedings of the IET-UK International Conference on Information and Communication Technology in Electrical Sciences (ICTES 2007)*, pages 789–794, December 2007.

[12] Akila Gothandaraman, Gregory D. Peterson, G. Lee Warren,

Robert J. Hinde, and Robert J. Harrison. FPGA acceleration of a quantum monte carlo application. *Parallel Computing*, 34(4-5):278–291, 2008.

[13] Yongfeng Gu, Tom Van Court, and Martin C. Herbordt. Improved interpolation and system integration for FPGA-based molecular dynamics simulations. In *Proceedings of the 2006 International Conference on Field Programmable Logic and Applications (FPL)*, pages 1–8. IEEE, August 2006.

[14] Yongfeng Gu and Martin C. Herbordt. FPGA-based multi-grid computation for molecular dynamics simulations. In *FCCM '07: Proceedings of the 15th Annual IEEE Symposium on Field-Programmable Custom Computing Machines*, pages 117–126, Washington, DC, USA, 2007. IEEE Computer Society.

[15] Yongfeng Gu, Tom VanCourt, and Martin C. Herbordt. Explicit design of FPGA-based coprocessors for short-range force computations in molecular dynamics simulations. *Parallel Comput.*, 34(4-5):261–277, 2008.

[16] Scott Hauck and André DeHon, editors. *Reconfigurable Computing.* Systems on Silicon. Morgan Kaufman, 30 Corporate Drive, Suite 400, Burlington, MA 01803, 2008.

[17] K. S. Hemmert and K. D. Underwood. An Analysis of the Double-Precision Floating-Point FFT on FPGAs. In *FCCM '05: Proceedings of the 13th Annual IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM'05)*, pages 171–180, Washington, DC, USA, 2005. IEEE Computer Society.

[18] Jing Huang, Mariam Momenzadeh, and Fabrizio Lombardi. Defect

tolerance of qca tiles. In *DATE '06: Proceedings of the conference on Design, automation and test in Europe*, pages 774–779, 3001 Leuven, Belgium, Belgium, 2006. European Design and Automation Association.

[19] Brad Hutchings, Peter Bellows, Joseph Hawkins, Scott Hemmert, Brent Nelson, and Mike Rytting. A CAD suite for high-performance FPGA design. In *FCCM '99: Proceedings of the Seventh Annual IEEE Symposium on Field-Programmable Custom Computing Machines*, page 12, Washington, DC, USA, 1999. IEEE Computer Society.

[20] Nachiket Kapre and André DeHon. Accelerating SPICE Model-Evaluation using FPGAs. *Field-Programmable Custom Computing Machines, Annual IEEE Symposium on*, 0:37–44, 2009.

[21] V. Kindratenko and D. Pointer. A case study in porting a production scientific supercomputing application to a reconfigurable computer. In *FCCM '06: Proceedings of the 14th Annual IEEE Symposium on Field-Programmable Custom Computing Machines*, pages 13–22, Washington, DC, USA, April 2006. IEEE Computer Society.

[22] R. K. Kummamuru, A. O. Orlov, R. Ramasubramaniam, C. S. Lent, G. H. Bernstein, and G. L. Snider. Operation of a quantum-dot cellular automata (qca) shift register and analysis of errors. *IEEE Transactions on Electron Devices*, 50(9):1906–1913, August 2003.

[23] Timothy D. Lantz and Eric R. Peskin. A QCA implementation of a configurable logic block for an FPGA. In *Proceedings of the*

Third International Conference on Reconfigurable Computing and FPGAs (ReConFig 2006), pages 132–141, September 2006.

[24] C. S. Lent, P. D. Tougaw, W. Porod, and G. H. Bernstein. Quantum cellular automata. *Nanotechnology*, 4(1):49–57, January 1993.

[25] Craig S. Lent, Sarah E. Frost, and Peter M. Kogge. Reversible computation with quantum-dot cellular automata (qca). In *CF '05: Proceedings of the 2nd conference on Computing frontiers*, page 403, New York, NY, USA, 2005. ACM.

[26] Craig S. Lent, Mo Liu, and Yuhui Lu. Bennett clocking of quantum-dot cellular automata and the limits to binary logic scaling. *Nanotechnology*, 17(16):4240+, August 2006.

[27] Antonio Lopes and George Constantinides. A High Throughput FPGA-Based Floating Point Conjugate Gradient Implementation. In Roger Woods, Katherine Compton, Christos Bouganis, and Pedro Diniz, editors, *Reconfigurable Computing: Architectures, Tools and Applications*, volume 4943 of *Lecture Notes in Computer Science*, chapter 10, pages 75–86. Springer Berlin / Heidelberg, Berlin, Heidelberg, 2008.

[28] Yuhui Lu, Mo Liu, and Craig Lent. Molecular quantum-dot cellular automata: From molecular structure to circuit dynamics. *Journal of Applied Physics*, 102(3):034311, 2007.

[29] Josh Model and Martin C. Herbordt. Discrete event simulation of molecular dynamics with configurable logic. In Koen Bertels,

Walid A. Najjar, Arjan J. van Genderen, and Stamatis Vassiliadis, editors, *FPL 2007, International Conference on Field Programmable Logic and Applications, Amsterdam, The Netherlands, 27-29 August 2007*, pages 151–158. IEEE, 2007.

[30] Tamer S. Mohamed and Wael Badawy. Integrated Hardware-Software Platform for Image Processing Applications. *System-on-Chip for Real-Time Applications, International Workshop on*, 0:145–148, 2004.

[31] Gordon E. Moore. Cramming more components onto integrated circuits. *Electronics*, 38(8):114 ff., April 1965.

[32] G. R. Morris and V. K. Prasanna. An FPGA-Based Floating-Point Jacobi Iterative Solver. In *Parallel Architectures,Algorithms and Networks, 2005. ISPAN 2005. Proceedings. 8th International Symposium on*, pages 420–427, 2005.

[33] Michael Thaddeus Niemier, Arun Francis Rodrigues, and Peter M. Kogge. A potentially implementable FPGA for quantum dot cellular automata. In *Proceedings of the First Workshop on Non-Silicon Computation (NSC-1)*, pages 38–45, 2002.

[34] A. O. Orlov, I. Amlani, G. Toth, C. S. Lent, G. H. Bernstein, and G. L. Snider. Experimental demonstration of a binary wire for quantum-dot cellular automata. *Applied Physics Letters*, 74(19):2875–2877, 1999.

[35] Alexei O. Orlov, Ravi K. Kummamuru, Rajagopal Ramasubramaniam, Geza Toth, Craig S. Lent, Gary H. Bernstein, and Gregory L.

Snider. Experimental demonstration of a latch in clocked quantum-dot cellular automata. *Applied Physics Letters*, 78(11):1625–1627, 2001.

[36] Marco Ottavi, Luca Schiano, Fabrizio Lombardi, and Douglas Tougaw. HDLQ: A HDL environment for QCA design. *J. Emerg. Technol. Comput. Syst.*, 2(4):243–261, 2006.

[37] S. Perri, M. Lanuzza, P. Corsonello, and G. Cocorullo. A high-performance fully reconfigurable FPGA-based 2D convolution processor. *Microprocessors and Microsystems*, 29(8-9):381–391, November 2005.

[38] Anders K. Roger and Roger D. Hersch. Designing a Halftoning Co-processor. In *8th Eurographics Workshop on Graphics Hardware*. Swiss Federal Institute of Technology, September 1993.

[39] R. Scrofano, M. Gokhale, F. Trouw, and V. K. Prasanna. Hardware/software approach to molecular dynamics on reconfigurable computers. In *FCCM '06: Proceedings of the 14th Annual IEEE Symposium on Field-Programmable Custom Computing Machines*, volume 0, pages 23–34, Los Alamitos, CA, USA, April 2006. IEEE Computer Society.

[40] Ronald Scrofano, Maya B. Gokhale, Frans Trouw, and Viktor K. Prasanna. Accelerating molecular dynamics simulations with reconfigurable computers. *IEEE Trans. Parallel Distrib. Syst.*, 19(6):764–778, 2008.

[41] B. Sen, M. Dalui, and B. K. Sikdar. Introducing universal qca logic

gate for synthesizing symmetric functions with minimum wire-crossings. In *ICWET '10: Proceedings of the International Conference and Workshop on Emerging Trends in Technology*, pages 828–833, New York, NY, USA, 2010. ACM.

[42] G. L. Snider, A. O. Orlov, I. Amlani, X. Zuo, G. H. Bernstein, C. S. Lent, J. L. Merz, and W. Porod. Quantum-dot cellular automata. In *Papers from the 45th National Symposium of the American Vacuum Society*, volume 17, pages 1394–1398. AVS, 1999.

[43] Yong Tang, Alexei O. Orlov, Gregory L. Snider, and Patrick J. Fay. Radio frequency operation of clocked quantum-dot cellular automata latch. *Applied Physics Letters*, 95(19):193109+, 2009.

[44] Himanshu Thapliyal and Nagarajan Ranganathan. Conservative qca gate (cqca) for designing concurrently testable molecular qca circuits. In *VLSID '09: Proceedings of the 2009 22nd International Conference on VLSI Design*, pages 511–516, Washington, DC, USA, 2009. IEEE Computer Society.

[45] Chia-Ching Tung. Implementation of multi-clb designs using quantum-dot cellular automata. Master's thesis, Rochester Institute of Technology, Rochester, NY, February 2010.

[46] Chia-Ching Tung, Ruchi B. Rungta, and Eric R. Peskin. Simulation of a QCA-based CLB and a multi-CLB application. In *Proceedings of the 2009 International Conference on Field-Programmable Technology (FPT'09)*, pages 62–69. IEEE, December 2009.

[47] Keith Underwood. Fpgas vs. cpus: trends in peak floating-point performance. In *FPGA '04: Proceedings of the 2004 ACM/SIGDA*

12th international symposium on Field programmable gate arrays, pages 171–180, New York, NY, USA, 2004. ACM.

[48] K. Walus. QCADesigner — Microsystems and Nanotechnology Group (MiNa). http://www.mina.ubc.ca/qcadesigner, June 2009.

[49] K. Walus, T. J. Dysart, G. A. Jullien, and R. A. Budiman. QCADesigner: a rapid design and simulation tool for quantum-dot cellular automata. *IEEE Trans. Nanotechnol.*, 3(1):26–31, March 2004.

[50] K. Walus, G. Schulhof, Mazur, and G. A. Jullien. Simple 4-bit processor based on quantum-dot cellular automata (QCA). In *ASAP '05: Proceedings of the 2005 IEEE International Conference on Application-Specific Systems, Architecture Processors*, pages 288–293, Washington, DC, USA, 2005. IEEE Computer Society.

[51] Yuliang Wang and M. Lieberman. Thermodynamic behavior of molecular-scale quantum-dot cellular automata (QCA) wires and logic devices. *IEEE Trans. Nanotechnol.*, 3(3):368–376, September 2004.

[52] Tongquan Wei, Kaijie Wu, Ramesh Karri, and Alex Orailoglu. Fault tolerant quantum cellular array (qca) design using triple modular redundancy with shifted operands. In *ASP-DAC '05: Proceedings of the 2005 Asia and South Pacific Design Automation Conference*, pages 1192–1195, New York, NY, USA, 2005. ACM.

[53] Xilinx, Inc., 2100 Logic Drive, San Jose, CA, 95124. *Virtex-6 Family Overview.*

Appendix A

run_bistable_simulation C Code Listing

The following is the C code for the run_bistable_simulation function from QCADesigner.

```
//———————————————————————————————————————————————————————————————————//
// -- this is the main simulation procedure -- //
//———————————————————————————————————————————————————————————————————//
simulation_data *run_bistable_simulation (int SIMULATION_TYPE, DESIGN *
    design, bistable_OP *options, VectorTable *pvt)
  {
  int i, j, k, l, total_cells = 0 ;
  int icLayers, icCellsInLayer;
  time_t start_time, end_time;
  simulation_data *sim_data = NULL ;
  // optimization variables //
  int number_of_cell_layers = 0, *number_of_cells_in_layer = NULL ;
  QCADCell ***sorted_cells = NULL ;
  double clock_shift = (options->clock_high + options->clock_low)/2 +
      options->clock_shift;
  double clock_prefactor = (options->clock_high - options->clock_low) *
      options->clock_amplitude_factor;
  double four_pi_over_number_samples = 4.0 * PI / (double) options->
      number_of_samples;
  double two_pi_over_number_samples = 2.0 * PI / (double) options->
      number_of_samples;
  int idxMasterBitOrder = -1 ;
  int max_iterations_per_sample = ((bistable_OP *)options)->
      max_iterations_per_sample;
  BUS_LAYOUT_ITER bli ;
#ifdef REDUCE_DEREF
  // For dereference reduction
  int sim_data_number_samples = 0, pvt_vectors_icUsed = 0,
    design_bus_layout_outputs_icUsed = 0,
        design_bus_layout_inputs_icUsed = 0, pvt_inputs_icUsed = 0 ;
  int number_of_cells_in_current_layer = 0 ;
  EXP_ARRAY *pvt_inputs = NULL ;
  EXP_ARRAY *pvt_vectors = NULL ;
  EXP_ARRAY *design_bus_layout_inputs = NULL ;
  EXP_ARRAY *design_bus_layout_outputs = NULL ;
  BUS_LAYOUT *design_bus_layout = NULL ;
#endif
  // For randomization
  int Nix, Nix1, idxCell1, idxCell2 ;
  QCADCell *swap = NULL ;
  // -- these used to be inside run_bistable_iteration -- //
  int q, iteration = 0;
  int stable = FALSE;
  double old_polarization;
  double new_polarization;
  double tolerance = ((bistable_OP *)options)->convergence_tolerance;
  double polarization_math;
  bistable_model *current_cell_model = NULL ;
  QCADCell *cell;

  STOP_SIMULATION = FALSE;

  // -- get the starting time for the simulation -- //
  if((start_time = time (NULL)) < 0)
    fprintf(stderr, "Could not get start time\n");

  // Create per-layer cell arrays to be used by the engine
  simulation_inproc_data_new (design, &number_of_cell_layers, &
      number_of_cells_in_layer, &sorted_cells) ;

  for(i = 0; i < number_of_cell_layers; i++)
    {
#ifdef REDUCE_DEREF
    number_of_cells_in_current_layer = number_of_cells_in_layer[i] ;
    for(j = 0; j < number_of_cells_in_current_layer ; j++)
#else
    for(j = 0; j < number_of_cells_in_layer[i] ; j++)
#endif
      {
      // attach the model parameters to each of the simulation cells //
      current_cell_model = g_malloc0 (sizeof(bistable_model)) ;
      sorted_cells[i][j]->cell_model = current_cell_model;

      // -- Clear the model pointers so they are not dangling -- //
      current_cell_model->neighbours = NULL;
      current_cell_model->Ek = NULL;

      // -- set polarization in cell model for fixed cells since they
          are set with actual dot charges by the user -- //
      if(QCAD_CELL_FIXED == sorted_cells[i][j]->cell_function)
       current_cell_model->polarization =
           qcad_cell_calculate_polarization(sorted_cells[i][j]);

      total_cells++;
      }
    }

  // if we are performing a vector table simulation we consider only the
      activated inputs //
  if(SIMULATION_TYPE == VECTOR_TABLE)
   {
   for (Nix = 0 ; Nix < pvt->inputs->icUsed ; Nix++)
     if (!exp_array_index_1d (pvt->inputs, VT_INPUT, Nix).active_flag)
       exp_array_index_1d (pvt->inputs, VT_INPUT, Nix).input->
           cell_function = QCAD_CELL_NORMAL ;
   }

  // write message to the command history window //
  command_history_message (_("Simulation found %d inputs %d outputs %d
      total cells\n"), design->bus_layout->inputs->icUsed, design->
      bus_layout->outputs->icUsed, total_cells) ;

  command_history_message(_("Starting initialization\n"));
  set_progress_bar_visible (TRUE) ;
  set_progress_bar_label (_("Bistable simulation:")) ;

  // -- Initialize the simualtion data structure -- //
  sim_data = g_malloc0 (sizeof(simulation_data));
  sim_data->number_of_traces = design->bus_layout->inputs->icUsed +
      design->bus_layout->outputs->icUsed;
  sim_data->number_samples = options->number_of_samples;
  sim_data->trace = g_malloc0 (sizeof (struct TRACEDATA) * sim_data->
      number_of_traces);

  // create and initialize the inputs into the sim data structure //
  for (i = 0; i < design->bus_layout->inputs->icUsed; i++)
    {
    sim_data->trace[i].data_labels = g_strdup (qcad_cell_get_label (
        QCAD_CELL (exp_array_index_1d (design->bus_layout->inputs,
        BUS_LAYOUT_CELL, i).cell))) ;
    sim_data->trace[i].drawtrace = TRUE;
    sim_data->trace[i].trace_function = QCAD_CELL_INPUT;
    sim_data->trace[i].data = g_malloc0 (sizeof (double) * sim_data->
        number_samples);
    }

  // create and initialize the outputs into the sim data structure //
  for (i = 0; i < design->bus_layout->outputs->icUsed; i++)
    {
    sim_data->trace[i + design->bus_layout->inputs->icUsed].data_labels
        = g_strdup (qcad_cell_get_label(QCAD_CELL(exp_array_index_1d (
        design->bus_layout->outputs, BUS_LAYOUT_CELL, i).cell))) ;
    sim_data->trace[i + design->bus_layout->inputs->icUsed].drawtrace =
        TRUE;
    sim_data->trace[i + design->bus_layout->inputs->icUsed].
        trace_function = QCAD_CELL_OUTPUT;
    sim_data->trace[i + design->bus_layout->inputs->icUsed].data =
        g_malloc0 (sizeof (double) * sim_data->number_samples);
    }

  // create and initialize the clock data //
  sim_data->clock_data = g_malloc0 (sizeof (struct TRACEDATA) * 4);

  for (i = 0; i < 4; i++)
    {
    sim_data->clock_data[i].data_labels = g_strdup_printf ("CLOCK %d", i
        );
    sim_data->clock_data[i].drawtrace = 1;
    sim_data->clock_data[i].trace_function = QCAD_CELL_FIXED; // Abusing
         the notation here

    sim_data->clock_data[i].data = g_malloc0 (sizeof (double) * sim_data
        ->number_samples);

    if (SIMULATION_TYPE == EXHAUSTIVE_VERIFICATION)
      for (j = 0; j < sim_data->number_samples; j++)
        {
        sim_data->clock_data[i].data[j] = clock_prefactor * cos (((
            double)(1 << design->bus_layout->inputs->icUsed)) * (double)
             j * four_pi_over_number_samples - PI * i / 2) + clock_shift
             ;
        sim_data->clock_data[i].data[j] = CLAMP (sim_data->clock_data[i
            ].data[j], options->clock_low, options->clock_high) ;
        }
    else
//      if (SIMULATION_TYPE == VECTOR_TABLE)
```

```
      for (j = 0; j < sim_data->number_samples; j++)
        {
        sim_data->clock_data[i].data[j] = clock_prefactor * cos (((
            double)pvt->vectors->icUsed) * (double)j *
            two_pi_over_number_samples - PI * i / 2) + clock_shift ;
        sim_data->clock_data[i].data[j] = CLAMP (sim_data->clock_data[i
            ].data[j], options->clock_low, options->clock_high) ;
        }
    }

  // -- refresh all the kink energies to all the cells neighbours within
       the radius of effect -- //
  bistable_refresh_all_Ek (number_of_cell_layers,
      number_of_cells_in_layer, sorted_cells, options);
  //this function takes up pretty much the whole initialization time

  // randomize the cells in the design so as to minimize any numerical
      problems associated //
  // with having cells simulated in some predefined order. //
  // randomize the order in which the cells are simulated //
  //if (options->randomize_cells)
  // for each layer ...
  for (Nix = 0 ; Nix < number_of_cell_layers ; Nix++)
    // ...perform as many swaps as there are cells therein
    for (Nix1 = 0 ; Nix1 < number_of_cells_in_layer[Nix] ; Nix1++)
      {
      idxCell1 = rand () % number_of_cells_in_layer[Nix] ;
      idxCell2 = rand () % number_of_cells_in_layer[Nix] ;

      swap = sorted_cells[Nix][idxCell1] ;
      sorted_cells[Nix][idxCell1] = sorted_cells[Nix][idxCell2] ;
      sorted_cells[Nix][idxCell2] = swap ;
      }

  // -- get and print the total initialization time -- //
  if((end_time = time (NULL)) < 0)
     fprintf(stderr, "Could not get end time\n");

  command_history_message("Total initialization time: %g s\n", (double)(
      end_time - start_time));

  command_history_message("Starting Simulation\n");

  set_progress_bar_fraction (0.0) ;

  // perform the iterations over all samples //
```

```
#ifdef REDUCE_DEREF
  // Dereference some structures now so we don't do it over and over in the loop
  sim_data_number_samples = sim_data->number_samples ;
  pvt_inputs = pvt->inputs ;
  pvt_inputs_icUsed = pvt_inputs->icUsed ;
  pvt_vectors = pvt->vectors ;
  pvt_vectors_icUsed = pvt->vectors->icUsed ;
  design_bus_layout = design->bus_layout ;
  design_bus_layout_inputs = design_bus_layout->inputs ;
  design_bus_layout_inputs_icUsed = design_bus_layout_inputs->icUsed ;
  design_bus_layout_outputs = design_bus_layout->outputs ;
  design_bus_layout_outputs_icUsed = design_bus_layout_outputs->icUsed ;
#else
  #define sim_data_number_samples sim_data->number_samples
  #define pvt_inputs pvt->inputs
  #define pvt_inputs_icUsed pvt_inputs->icUsed
  #define pvt_vectors pvt->vectors
  #define pvt_vectors_icUsed pvt->vectors->icUsed
  #define design_bus_layout design->bus_layout
  #define design_bus_layout_inputs design_bus_layout->inputs
  #define design_bus_layout_inputs_icUsed design_bus_layout_inputs->icUsed
  #define design_bus_layout_outputs design_bus_layout->outputs
  #define design_bus_layout_outputs_icUsed design_bus_layout_outputs->icUsed
#endif
  for (j = 0; j < sim_data_number_samples ; j++)
    {
    if (j % 100 == 0)
      {
      // write the completion percentage to the command history window //
      set_progress_bar_fraction ((float) j / (float) sim_data_number_samples) ;
      // redraw the design if the user wants it to appear animated //
      if(options->animate_simulation)
        {
        // update the charges to reflect the polarizations so that they can be animated //
        for(icLayers = 0; icLayers < number_of_cell_layers; icLayers++)
          {
#ifdef REDUCE_DEREF
          number_of_cells_in_current_layer = number_of_cells_in_layer[icLayers] ;
          for(icCellsInLayer = 0; icCellsInLayer <
              number_of_cells_in_current_layer; icCellsInLayer++)
#else
          for(icCellsInLayer = 0; icCellsInLayer <
              number_of_cells_in_layer[icLayers]; icCellsInLayer++)
#endif
            qcad_cell_set_polarization(sorted_cells[icLayers][
                icCellsInLayer],((bistable_model *)sorted_cells[icLayers
                ][icCellsInLayer]->cell_model)->polarization);
          }
#ifdef DESIGNER
        redraw_async(NULL);
        gdk_flush () ;
#endif /* def DESIGNER */
        }
      }

    // -- for each of the (VECTOR_TABLE => active?) inputs -- //
    if (EXHAUSTIVE_VERIFICATION == SIMULATION_TYPE)
      for (idxMasterBitOrder = 0, design_bus_layout_iter_first (
          design_bus_layout, &bli, QCAD_CELL_INPUT, &i) ; i > -1 ;
          design_bus_layout_iter_next (&bli, &i), idxMasterBitOrder++)
        ((bistable_model *)exp_array_index_1d (design_bus_layout_inputs,
            BUS_LAYOUT_CELL, i).cell->cell_model)->polarization =
          sim_data->trace[i].data[j] = (-1 * sin (((double)(1 <<
              idxMasterBitOrder)) * (double)j * FOUR_PI / (double)
              sim_data_number_samples) > 0) ? 1 : -1 ;
    else
//      if (VECTOR_TABLE == SIMULATION_TYPE)
      for (design_bus_layout_iter_first (design_bus_layout, &bli,
          QCAD_CELL_INPUT, &i) ; i > -1 ; design_bus_layout_iter_next (&
          bli, &i))
        if (exp_array_index_1d (pvt_inputs, VT_INPUT, i).active_flag)
          ((bistable_model *)exp_array_index_1d (pvt_inputs, VT_INPUT, i
              ).input->cell_model)->polarization =
            sim_data->trace[i].data[j] = exp_array_index_2d (pvt_vectors
                , gboolean, (j * pvt_vectors_icUsed) /
                sim_data_number_samples, i) ? 1 : -1 ;

    // randomize the order in which the cells are simulated to try and
        minimize numerical errors
    // associated with the imposed simulation order.
    if(options->randomize_cells)
      // for each layer ...
      for (Nix = 0 ; Nix < number_of_cell_layers ; Nix++)
        {
```

```
        // ...perform as many swaps as there are cells therein
#ifdef REDUCE_DEREF
        number_of_cells_in_current_layer = number_of_cells_in_layer[Nix]
            ;
        for (Nix1 = 0 ; Nix1 < number_of_cells_in_current_layer ; Nix1
           ++)
#else
        for (Nix1 = 0 ; Nix1 < number_of_cells_in_layer[Nix] ; Nix1++)
#endif
          {
#ifdef REDUCE_DEREF
          idxCell1 = rand () % number_of_cells_in_current_layer ;
          idxCell2 = rand () % number_of_cells_in_current_layer ;
#else
          idxCell1 = rand () % number_of_cells_in_layer[Nix] ;
          idxCell2 = rand () % number_of_cells_in_layer[Nix] ;
#endif

          swap = sorted_cells[Nix][idxCell1] ;
          sorted_cells[Nix][idxCell1] = sorted_cells[Nix][idxCell2] ;
          sorted_cells[Nix][idxCell2] = swap ;
          }
        }

    // -- run the iteration with the given clock value -- //
    // -- iterate until the entire design has stabalized -- //
    iteration = 0;
    stable = FALSE;
    while (!stable && iteration < max_iterations_per_sample)
      {
      iteration++;
      // -- assume that the circuit is stable -- //
      stable = TRUE;

      for (icLayers = 0; icLayers < number_of_cell_layers; icLayers++)
        {
#ifdef REDUCE_DEREF
        number_of_cells_in_current_layer = number_of_cells_in_layer[
            icLayers] ;
        for (icCellsInLayer = 0 ; icCellsInLayer <
            number_of_cells_in_current_layer ; icCellsInLayer++)
#else
        for (icCellsInLayer = 0 ; icCellsInLayer <
            number_of_cells_in_layer[icLayers] ; icCellsInLayer++)
#endif
          {
```

```
          cell = sorted_cells[icLayers][icCellsInLayer] ;

          if (!((QCAD_CELL_INPUT == cell->cell_function)||
                (QCAD_CELL_FIXED == cell->cell_function)))
            {
            current_cell_model = ((bistable_model *)cell->cell_model) ;
            old_polarization = current_cell_model->polarization;
            polarization_math = 0;

            for (q = 0; q < current_cell_model->number_of_neighbours; q
                ++)
              polarization_math += (current_cell_model->Ek[q] * ((
                  bistable_model *)current_cell_model->neighbours[q]->
                  cell_model)->polarization);

            // math = math / 2 * gamma
            polarization_math /= (2.0 * sim_data->clock_data[cell->
                cell_options.clock].data[j]);

            // -- calculate the new cell polarization -- //
            // if math < 0.05 then math/sqrt(1+math^2) ~= math with
                error <= 4e-5
            // if math > 100 then math/sqrt(1+math^2) ~= +-1 with error
                <= 5e-5
            new_polarization =
              (polarization_math        >  1000.0)   ?  1
                                         :
              (polarization_math        < -1000.0)   ? -1
                                         :
              (fabs (polarization_math) <     0.001) ?
                  polarization_math :
                polarization_math / sqrt (1 + polarization_math *
                    polarization_math) ;

            // -- set the polarization of this cell -- //
            current_cell_model->polarization = new_polarization;

            // If any cells polarization has changed beyond this
                threshold
            // then the entire circuit is assumed to have not converged.
           stable = (fabs (new_polarization - old_polarization) <=
              tolerance) && stable ;
            }
          }
        }
      }//WHILE !STABLE
```

```

    if (VECTOR_TABLE == SIMULATION_TYPE)
      for (design_bus_layout_iter_first (design_bus_layout, &bli,
          QCAD_CELL_INPUT, &i) ; i > -1 ; design_bus_layout_iter_next (&
          bli, &i))
        if (!exp_array_index_1d (pvt_inputs, VT_INPUT, i).active_flag)
          sim_data->trace[i].data[j] = ((bistable_model *)
              exp_array_index_1d (pvt_inputs, VT_INPUT, i).input->
              cell_model)->polarization;

    // -- collect all the output data from the simulation -- //
    for (design_bus_layout_iter_first (design_bus_layout, &bli,
        QCAD_CELL_OUTPUT, &i) ; i > -1 ; design_bus_layout_iter_next (&
        bli, &i)){
      sim_data->trace[design_bus_layout_inputs_icUsed + i].data[j] = ((
          bistable_model *)exp_array_index_1d (design_bus_layout_outputs
          , BUS_LAYOUT_CELL, i).cell->cell_model)->polarization;
  }
    // -- if the user wants to stop the simulation then exit. -- //
    if(TRUE == STOP_SIMULATION)
      j = sim_data_number_samples ;
    }//for number of samples

  // Free the neigbours and Ek array introduced by this simulation//
  for (k = 0; k < number_of_cell_layers; k++)
    {
#ifdef REDUCE_DEREF
    number_of_cells_in_current_layer = number_of_cells_in_layer[k] ;
    for (l = 0 ; l < number_of_cells_in_current_layer ; l++)
#else
    for (l = 0 ; l < number_of_cells_in_layer[k] ; l++)
#endif
      {
      g_free(((bistable_model *)sorted_cells[k][l]->cell_model)->
          neighbours);
      g_free(((bistable_model *)sorted_cells[k][l]->cell_model)->
          neighbour_layer);
      g_free(((bistable_model *)sorted_cells[k][l]->cell_model)->Ek);
      }
    }

  simulation_inproc_data_free (&number_of_cell_layers, &
      number_of_cells_in_layer, &sorted_cells) ;

// Restore the input flag for the inactive inputs
  if (VECTOR_TABLE == SIMULATION_TYPE)
```

```
    for (i = 0 ; i < pvt_inputs_icUsed ; i++)
      exp_array_index_1d (pvt_inputs, VT_INPUT, i).input->cell_function
          = QCAD_CELL_INPUT ;

// -- get and print the total simulation time -- //
  if ((end_time = time (NULL)) < 0)
    fprintf(stderr, "Could not get end time\n");

  command_history_message ("Total simulation time: %g s\n", (double)(
      end_time - start_time));
  set_progress_bar_visible (FALSE) ;

#ifndef REDUCE_DEREF
  #undef sim_data_number_samples
  #undef pvt_inputs
  #undef pvt_inputs_icUsed
  #undef pvt_vectors
  #undef pvt_vectors_icUsed
  #undef design_bus_layout
  #undef design_bus_layout_inputs
  #undef design_bus_layout_inputs_icUsed
  #undef design_bus_layout_outputs
  #undef design_bus_layout_outputs_icUsed
#endif

  return sim_data;
  }//run_bistable
```

Listing A.1: Bistable simulation engine software code

Appendix B

run_bistable_simulation_hardware C Code Listing

The following is the code for the version of run_bistable_simulation created to use the hardware

```
//calls the hardware to run an alternate version of the bistable
    simulation engine
simulation_data *run_bistable_simulation_hardware (int SIMULATION_TYPE,
    DESIGN *design, bistable_OP *options, VectorTable *pvt)
  {
  int i, j, k, l, total_cells = 0 ;
  int icLayers, icCellsInLayer;
  time_t start_time, end_time;
  simulation_data *sim_data = NULL ;
  // optimization variables //
  int number_of_cell_layers = 0, *number_of_cells_in_layer = NULL ;
  QCADCell ***sorted_cells = NULL ;
  float clock_shift = (float)((options->clock_high + options->clock_low)
      /2 + options->clock_shift);
  float clock_prefactor = (float)(options->clock_high - options->
      clock_low) * options->clock_amplitude_factor;
  float four_pi_over_number_samples = 4.0 * (float)PI / (float) options
      ->number_of_samples;
  float two_pi_over_number_samples = 2.0 * (float)PI / (float) options->
      number_of_samples;
  int idxMasterBitOrder = -1 ;
  int max_iterations_per_sample = ((bistable_OP *)options)->
      max_iterations_per_sample;
  BUS_LAYOUT_ITER bli ;
#ifdef REDUCE_DEREF
  // For dereference reduction
  int sim_data_number_samples = 0, pvt_vectors_icUsed = 0,
    design_bus_layout_outputs_icUsed = 0,
        design_bus_layout_inputs_icUsed = 0, pvt_inputs_icUsed = 0 ;
  int number_of_cells_in_current_layer = 0 ;
  EXP_ARRAY *pvt_inputs = NULL ;
  EXP_ARRAY *pvt_vectors = NULL ;
  EXP_ARRAY *design_bus_layout_inputs = NULL ;
  EXP_ARRAY *design_bus_layout_outputs = NULL ;
  BUS_LAYOUT *design_bus_layout = NULL ;
#endif
  // -- these used to be inside run_bistable_iteration -- //
  int Nix;
  int q, iteration, calculatedCellNum= 0;
  int stable = FALSE;
  float old_polarization;
  float new_polarization;
  float tolerance = (float)((bistable_OP *)options)->
      convergence_tolerance;
  float temp; //just used to move data formats between DWORDS and floats
  bistable_model *current_cell_model = NULL ;
  QCADCell *cell;

  //wildcard initialization
  WC_RetCode rc = WC_SUCCESS;
  WC_TestInfo TestInfo;
  char **TestLoc = NULL;
  DWORD control[8];
  WC4_DmaHandle hDMASource;
  WC4_DmaHandle hDMADestination;
  DWORD dGrantedDwords;
  DWORD dSourceHardwareAddress;
    DWORD dDestinationHardwareAddress;
     float *DMABufSource;
     float *DMABufDest;
  boolean intStatus;
  DWORD debugReg[8];
    DWORD maxNeighbours_DMA = 0x400; //max number of neighbours
        supported, used to allocate source buffer (1024)
    DWORD numCells_DMA = 0; //number of cells to be read from the PE;
        sets DMA destination buffer; calculated later
    DWORD totem = 0xBF000000; //totem value used to signal end of things

  TestInfo.bVerbose      = DEFAULT_VERBOSITY;
  TestInfo.DeviceNum     = DEFAULT_SLOT_NUMBER;
  TestInfo.dIterations   = DEFAULT_ITERATIONS;
  TestInfo.fClkFreq      = DEFAULT_FREQUENCY;

  STOP_SIMULATION = FALSE;

  // -- get the starting time for the simulation -- //
  if((start_time = time (NULL)) < 0)
    fprintf(stderr, "Could not get start time\n");
  // Create per-layer cell arrays to be used by the engine
  simulation_inproc_data_new (design, &number_of_cell_layers, &
      number_of_cells_in_layer, &sorted_cells) ;
  for(i = 0; i < number_of_cell_layers; i++)
    {
#ifdef REDUCE_DEREF
    number_of_cells_in_current_layer = number_of_cells_in_layer[i] ;
    for(j = 0; j < number_of_cells_in_current_layer ; j++)
#else
    for(j = 0; j < number_of_cells_in_layer[i] ; j++)
#endif
      {
      // attach the model parameters to each of the simulation cells //
      current_cell_model = g_malloc0 (sizeof(bistable_model)) ;
      sorted_cells[i][j]->cell_model = current_cell_model;

      // -- Clear the model pointers so they are not dangling -- //
      current_cell_model->neighbours = NULL;
      current_cell_model->Ek = NULL;

      // -- set polarization in cell model for fixed cells since they
          are set with actual dot charges by the user -- //
      if(QCAD_CELL_FIXED == sorted_cells[i][j]->cell_function)
       current_cell_model->polarization =
           qcad_cell_calculate_polarization(sorted_cells[i][j]);

      total_cells++;

    if (!((QCAD_CELL_INPUT == sorted_cells[i][j]->cell_function)||(
        QCAD_CELL_FIXED == sorted_cells[i][j]->cell_function)))
      numCells_DMA++; //how many cells will be calculated by the
          hardware
      }
    }

  // if we are performing a vector table simulation we consider only the
      activated inputs //
  if(SIMULATION_TYPE == VECTOR_TABLE)
   {
   for (Nix = 0 ; Nix < pvt->inputs->icUsed ; Nix++)
     if (!exp_array_index_1d (pvt->inputs, VT_INPUT, Nix).active_flag)
       exp_array_index_1d (pvt->inputs, VT_INPUT, Nix).input->
           cell_function = QCAD_CELL_NORMAL ;
    }

  // write message to the command history window //
  command_history_message (_("Simulation found %d inputs %d outputs %d
      total cells\n"), design->bus_layout->inputs->icUsed, design->
      bus_layout->outputs->icUsed, total_cells) ;

  command_history_message(_("Starting initialization\n"));
  command_history_message(_("Opening card.."));
  rc = WC_Open (TestInfo.DeviceNum, 0);
  if (rc != WC_SUCCESS) {
    command_history_message("Failed to open card.\n");
    return NULL;
  }


    command_history_message(_("Initializing hardware..."));
  //initialize and reset the PE
  rc = wc4_qca_init( &TestInfo);
  if (rc != WC_SUCCESS) {
    command_history_message("initialization failed\n");
    return NULL;
  }
  //CHECK_RC(rc);

  command_history_message(_("resetting hardware..."));
    rc = WC_PeReset (TestInfo.DeviceNum, TRUE);
  if (rc != WC_SUCCESS) {
    command_history_message("failed\n");
    return NULL;
  }
    //CHECK_RC(rc);

    rc = WC_PeReset (TestInfo.DeviceNum, FALSE);
  if (rc != WC_SUCCESS) {
    command_history_message("failed\n");
    return NULL;
  }
  //CHECK_RC (rc);
    command_history_message("Done\n");

  set_progress_bar_visible (TRUE) ;
  set_progress_bar_label (_("Bistable simulation:")) ;

  // -- Initialize the simualtion data structure -- //
  sim_data = g_malloc0 (sizeof(simulation_data));
```

```
sim_data->number_of_traces = design->bus_layout->inputs->icUsed +
    design->bus_layout->outputs->icUsed;
sim_data->number_samples = options->number_of_samples;
sim_data->trace = g_malloc0 (sizeof (struct TRACEDATA) * sim_data->
    number_of_traces);

// create and initialize the inputs into the sim data structure //
for (i = 0; i < design->bus_layout->inputs->icUsed; i++)
  {
  sim_data->trace[i].data_labels = g_strdup (qcad_cell_get_label (
      QCAD_CELL (exp_array_index_1d (design->bus_layout->inputs,
      BUS_LAYOUT_CELL, i).cell))) ;
  sim_data->trace[i].drawtrace = TRUE;
  sim_data->trace[i].trace_function = QCAD_CELL_INPUT;
  sim_data->trace[i].data = g_malloc0 (sizeof (double) * sim_data->
      number_samples);
  }

// create and initialize the outputs into the sim data structure //
for (i = 0; i < design->bus_layout->outputs->icUsed; i++)
  {
  sim_data->trace[i + design->bus_layout->inputs->icUsed].data_labels
      = g_strdup (qcad_cell_get_label(QCAD_CELL(exp_array_index_1d (
      design->bus_layout->outputs, BUS_LAYOUT_CELL, i).cell))) ;
  sim_data->trace[i + design->bus_layout->inputs->icUsed].drawtrace =
      TRUE;
  sim_data->trace[i + design->bus_layout->inputs->icUsed].
      trace_function = QCAD_CELL_OUTPUT;
  sim_data->trace[i + design->bus_layout->inputs->icUsed].data =
      g_malloc0 (sizeof (double) * sim_data->number_samples);
  }

// create and initialize the clock data //
sim_data->clock_data = g_malloc0 (sizeof (struct TRACEDATA) * 4);

for (i = 0; i < 4; i++)
  {
  sim_data->clock_data[i].data_labels = g_strdup_printf ("CLOCK %d", i
      );
  sim_data->clock_data[i].drawtrace = 1;
  sim_data->clock_data[i].trace_function = QCAD_CELL_FIXED; // Abusing
       the notation here

  sim_data->clock_data[i].data = g_malloc0 (sizeof (double) * sim_data
      ->number_samples);

    if (SIMULATION_TYPE == EXHAUSTIVE_VERIFICATION)
      for (j = 0; j < sim_data->number_samples; j++)
        {
        sim_data->clock_data[i].data[j] = clock_prefactor * cos (((double)(1 << design->bus_layout->inputs->icUsed)) * (double)j * four_pi_over_number_samples - PI * i / 2) + clock_shift ;
        sim_data->clock_data[i].data[j] = CLAMP (sim_data->clock_data[i].data[j], options->clock_low, options->clock_high) ;
        }
    else
//    if (SIMULATION_TYPE == VECTOR_TABLE)
      for (j = 0; j < sim_data->number_samples; j++)
        {
        sim_data->clock_data[i].data[j] = clock_prefactor * cos (((double)pvt->vectors->icUsed) * (double)j * two_pi_over_number_samples - PI * i / 2) + clock_shift ;
        sim_data->clock_data[i].data[j] = CLAMP (sim_data->clock_data[i].data[j], options->clock_low, options->clock_high) ;
        }
    }

  command_history_message("Allocating DMA buffers...");
  rc = WC4_DmaMemAlloc (TestInfo.DeviceNum, &hDMASource, maxNeighbours_DMA, (DWORD **)&DMABufSource);
  if (rc != WC_SUCCESS) {
    command_history_message("\nError Allocating DMA source buffer\n");
    WC4_DmaMemFree(TestInfo.DeviceNum, hDMASource);
    WC_PeDeprogram(TestInfo.DeviceNum);
    WC4_PeInitiateAutoProgram (TestInfo.DeviceNum);
    WC_Close (TestInfo.DeviceNum);
    return NULL;
  }
  rc = WC4_DmaMemAlloc (TestInfo.DeviceNum, &hDMADestination, numCells_DMA, (DWORD **)&DMABufDest);
  if (rc != WC_SUCCESS) {
    command_history_message("\nError Allocating DMA destination buffer\n" );
    WC4_DmaMemFree(TestInfo.DeviceNum, hDMASource);
    WC4_DmaMemFree(TestInfo.DeviceNum, hDMADestination);
    WC_PeDeprogram(TestInfo.DeviceNum);
    WC4_PeInitiateAutoProgram (TestInfo.DeviceNum);
    WC_Close (TestInfo.DeviceNum);
    return NULL;
  }
  printf("DONE\n");
```

```

    printf("Binding DMA buffers\t\t\t");
    rc = WC4_DmaBind      (TestInfo.DeviceNum, hDMASource, (DWORD *)
        DMABufSource, maxNeighbours_DMA,  &dGrantedDwords,  &
        dSourceHardwareAddress );
    if (rc != WC_SUCCESS) {
      printf ("\nError binding DMA source buffer\n" );
      WC4_DmaMemFree (TestInfo.DeviceNum, hDMASource);
      WC4_DmaMemFree (TestInfo.DeviceNum, hDMADestination);
    WC_PeDeprogram(TestInfo.DeviceNum);
    WC4_PeInitiateAutoProgram (TestInfo.DeviceNum);
    WC_Close (TestInfo.DeviceNum);
    return NULL;
    } else if (maxNeighbours_DMA != dGrantedDwords) {
      printf ("\nError could not bind full source buffer!\n");
      WC4_DmaMemFree (TestInfo.DeviceNum, hDMASource);
      WC4_DmaMemFree (TestInfo.DeviceNum, hDMADestination);
    WC_PeDeprogram(TestInfo.DeviceNum);
    WC4_PeInitiateAutoProgram (TestInfo.DeviceNum);
    WC_Close (TestInfo.DeviceNum);
      return NULL;
    }

    rc = WC4_DmaBind (TestInfo.DeviceNum, hDMADestination, (DWORD *)
        DMABufDest, numCells_DMA,  &dGrantedDwords,  &
        dDestinationHardwareAddress );
    if (rc != WC_SUCCESS) {
      printf("\nError binding DMA destination buffer\n");
      WC4_DmaUnbind (TestInfo.DeviceNum, hDMASource);
      WC4_DmaMemFree (TestInfo.DeviceNum, hDMADestination);
      WC4_DmaMemFree (TestInfo.DeviceNum, hDMASource);
    WC_PeDeprogram(TestInfo.DeviceNum);
    WC4_PeInitiateAutoProgram (TestInfo.DeviceNum);
    WC_Close (TestInfo.DeviceNum);
      return NULL;
    } else if (numCells_DMA != dGrantedDwords) {
      printf("\nError could not bind full destination buffer!\n");
      WC4_DmaUnbind (TestInfo.DeviceNum, hDMASource);
      WC4_DmaMemFree (TestInfo.DeviceNum, hDMADestination);
      WC4_DmaMemFree (TestInfo.DeviceNum, hDMASource);
    WC_PeDeprogram(TestInfo.DeviceNum);
    WC4_PeInitiateAutoProgram (TestInfo.DeviceNum);
    WC_Close (TestInfo.DeviceNum);
      return NULL;
    }
    printf("DONE\n");

    //send DMA control data to PE
    control[0] = dDestinationHardwareAddress;
    control[1] = dSourceHardwareAddress;
    control[2] = numCells_DMA;
  command_history_message("Sending DMA control data\n");
    rc = WC_PeRegWrite(TestInfo.DeviceNum, CTRL_REG_BASE, 3, control);
    if (rc != WC_SUCCESS) {
      command_history_message("Register transfer failed\n");
    wildcard_cleanup(TestInfo.DeviceNum, &hDMASource, &hDMADestination);
    return NULL;
  }
  WC_PeRegRead(TestInfo.DeviceNum, CTRL_REG_BASE+4,4, debugReg);
  command_history_message("state: %x\n", debugReg[0]);
  command_history_message("Calculating kink energies\n");
  command_history_message("DMA To PE Addr: %x\n",dSourceHardwareAddress);
  // -- refresh all the kink energies to all the cells neighbours within the radius of effect -- //
  bistable_refresh_all_Ek_hardware (number_of_cell_layers, number_of_cells_in_layer, sorted_cells, options, DMABufSource, TestInfo.DeviceNum);
  //Using TestInfo.DeviceNum instead of TestInfo->DeviceNum. Don't know why it has to be like this. Make sure this doesn't cause issues
  command_history_message("done calculating  kink energies, waiting for interrupt\n");
  WC_IntQueryStatus(TestInfo.DeviceNum, &intStatus);
  command_history_message("interrupt status: %d\n",intStatus);
  rc=WC_IntWait(TestInfo.DeviceNum, INT_TIMEOUT_MS); //wait for interrupt for the last set of Eks to be confirmed as received
  //might want to find a different way of testing for the interrupt, because this seems cumbersome.
  if(rc != WC_SUCCESS) {
    WC_PeRegRead(TestInfo.DeviceNum, CTRL_REG_BASE, 8, debugReg);
    command_history_message("Interrupt timed out; int Counter, numNeighbours, dram_addr, dma_incount, state, in_fifo_out, DRAM data, DMA_outCount: %x %x %x %x %x %x %x %x\n",
      debugReg[0], debugReg[1], debugReg[2], debugReg[3], debugReg[4], debugReg[5], debugReg[6], debugReg[7]);
    wildcard_cleanup(TestInfo.DeviceNum, &hDMASource, &hDMADestination);
    return NULL;
  }
  command_history_message("interrupt received\n");
  rc=WC_IntEnable(TestInfo.DeviceNum, FALSE); //reset interrupt
  rc=WC_IntReset(TestInfo.DeviceNum);
  rc=WC_IntEnable(TestInfo.DeviceNum, TRUE);
  command_history_message("interrupt reset\n");
  WC_PeRegWrite(TestInfo.DeviceNum, CTRL_REG_BASE+3, 1, &totem); //send
      a totem value to tell the PE that all Ek data has been sent

  // -- get and print the total initialization time -- //
  if((end_time = time (NULL)) < 0)
      fprintf(stderr, "Could not get end time\n");

  //there was a cell randomization sequence here, but the hardware
      accelerator makes it unnecessary

  command_history_message("Total initialization time: %g s\n", (double)(
      end_time - start_time));

  command_history_message("Starting Simulation\n");
  set_progress_bar_fraction (0.0) ;

  // perform the iterations over all samples //
#ifdef REDUCE_DEREF
  // Dereference some structures now so we don't do it over and over in
      the loop
  sim_data_number_samples = sim_data->number_samples ;
  pvt_inputs = pvt->inputs ;
  pvt_inputs_icUsed = pvt_inputs->icUsed ;
  pvt_vectors = pvt->vectors ;
  pvt_vectors_icUsed = pvt->vectors->icUsed ;
  design_bus_layout = design->bus_layout ;
  design_bus_layout_inputs = design_bus_layout->inputs ;
  design_bus_layout_inputs_icUsed = design_bus_layout_inputs->icUsed ;
  design_bus_layout_outputs = design_bus_layout->outputs ;
  design_bus_layout_outputs_icUsed = design_bus_layout_outputs->icUsed ;
#else
  #define sim_data_number_samples sim_data->number_samples
  #define pvt_inputs pvt->inputs
  #define pvt_inputs_icUsed pvt_inputs->icUsed
  #define pvt_vectors pvt->vectors
  #define pvt_vectors_icUsed pvt->vectors->icUsed
  #define design_bus_layout design->bus_layout
  #define design_bus_layout_inputs design_bus_layout->inputs
  #define design_bus_layout_inputs_icUsed design_bus_layout_inputs->
      icUsed
  #define design_bus_layout_outputs design_bus_layout->outputs
  #define design_bus_layout_outputs_icUsed design_bus_layout_outputs->
      icUsed
#endif
  for (j = 0; j < sim_data_number_samples ; j++)
  {
  if (j % 100 == 0)
    {
    // write the completion percentage to the command history window
        //
    set_progress_bar_fraction ((float) j / (float)
        sim_data_number_samples) ;
    // redraw the design if the user wants it to appear animated //
    }

  // -- for each of the (VECTOR_TABLE => active?) inputs -- //
  if (EXHAUSTIVE_VERIFICATION == SIMULATION_TYPE)
    for (idxMasterBitOrder = 0, design_bus_layout_iter_first (
        design_bus_layout, &bli, QCAD_CELL_INPUT, &i) ; i > -1 ;
        design_bus_layout_iter_next (&bli, &i), idxMasterBitOrder++)
      ((bistable_model *)exp_array_index_1d (design_bus_layout_inputs,
          BUS_LAYOUT_CELL, i).cell->cell_model)->polarization =
       sim_data->trace[i].data[j] = (-1 * sin (((double)(1 <<
          idxMasterBitOrder)) * (double)j * FOUR_PI / (double)
          sim_data_number_samples) > 0) ? 1 : -1 ;
  else
//    if (VECTOR_TABLE == SIMULATION_TYPE)
    for (design_bus_layout_iter_first (design_bus_layout, &bli,
        QCAD_CELL_INPUT, &i) ; i > -1 ; design_bus_layout_iter_next (&
        bli, &i))
      if (exp_array_index_1d (pvt_inputs, VT_INPUT, i).active_flag)
        ((bistable_model *)exp_array_index_1d (pvt_inputs, VT_INPUT, i
          ).input->cell_model)->polarization =
         sim_data->trace[i].data[j] = exp_array_index_2d (pvt_vectors
            , gboolean, (j * pvt_vectors_icUsed) /
            sim_data_number_samples, i) ? 1 : -1 ;

  // -- run the iteration with the given clock value -- //
  // -- iterate until the entire design has stabalized -- //
  iteration = 0;
  stable = FALSE;
  while (!stable && iteration < max_iterations_per_sample)
    {
    iteration++;
    // -- assume that the circuit is stable -- //
    stable = TRUE;

    for (icLayers = 0; icLayers < number_of_cell_layers; icLayers++)
      {
#ifdef REDUCE_DEREF
```

```
        number_of_cells_in_current_layer = number_of_cells_in_layer[icLayers] ;
        for (icCellsInLayer = 0 ; icCellsInLayer < number_of_cells_in_current_layer ; icCellsInLayer++)
#else
        for (icCellsInLayer = 0 ; icCellsInLayer < number_of_cells_in_layer[icLayers] ; icCellsInLayer++)
#endif
          {
          cell = sorted_cells[icLayers][icCellsInLayer] ;

          if (!((QCAD_CELL_INPUT == cell->cell_function)||
                (QCAD_CELL_FIXED == cell->cell_function)))
            {
            current_cell_model = ((bistable_model *)cell->cell_model) ;

            for (q = 0; q < current_cell_model->number_of_neighbours; q++)
      {
        DMABufSource[q] = ((bistable_model *)current_cell_model->neighbours[q]->cell_model)->polarization;
      }

      command_history_message("setting gamma...");
        temp = (float)(sim_data->clock_data[cell->cell_options.clock].data[j]);
      control[0] = *(DWORD *)&temp; //casting necessary to put a float into a DWORD unaltered
      command_history_message("setting neighbours...\n");
        control[1] = (DWORD)(current_cell_model->number_of_neighbours);
      command_history_message("set\n");
        if(icCellsInLayer != 0) //no interrupt comes for the first cell
        //we may have to add a condition for the first layer as well...
      { command_history_message("waiting for calculation\n");
        rc=WC_IntWait(TestInfo.DeviceNum, INT_TIMEOUT_MS); //wait for interrupt, signifying current calculation is done
        if(rc != WC_SUCCESS) { //debug output
          WC_PeRegRead(TestInfo.DeviceNum, CTRL_REG_BASE, 8, debugReg);
          command_history_message("Interrupt timed out; int Counter, numNeighbours, dram_addr, dma_incount, state, in_fifo_out, DRAM data, DMA_outCount: %x %x %x %x %x %x %x %x\n",
            debugReg[0], debugReg[1], debugReg[2], debugReg[3], debugReg[4], debugReg[5], debugReg[6], debugReg[7]);
          wildcard_cleanup(TestInfo.DeviceNum, &hDMASource, &hDMADestination);
          return NULL;
        }
        command_history_message("interrupt received\n"); //debugging
            information
        rc=WC_IntEnable(TestInfo.DeviceNum, FALSE); //reset interrupt
        command_history_message("interrupt disabled\n");
        rc=WC_IntReset(TestInfo.DeviceNum);
        command_history_message("interrupt reset\n");
        rc=WC_IntEnable(TestInfo.DeviceNum, TRUE);
        command_history_message("interrupt enabled\n");
      }
      if(j==0 && iteration == 1)
        control[1] = 0x80000000 | (DWORD)(current_cell_model->
            number_of_neighbours); //tell the PE this is not the first
            sample

      command_history_message("control data: %x %x\n", (int)control[0],
          (int)control[1]);
      command_history_message("sending polarizations\n");
      //get debugging data
        WC_PeRegRead(TestInfo.DeviceNum, CTRL_REG_BASE, 8, debugReg);
        command_history_message("Ek in, read count, write count, gamma
            in, state, in_fifo_out, DRAM fifo data, out_fifo_in: %x %x %
            x %x %x %x %x %x\n",
          debugReg[0], debugReg[2], debugReg[7], debugReg[1], debugReg
              [4], debugReg[5], debugReg[6], debugReg[3]);
      rc = WC_PeRegWrite(TestInfo.DeviceNum, CTRL_REG_BASE+3, 2, control
          ); //send gamma and numNeighbours
      if (rc != WC_SUCCESS) {
        command_history_message("Register transfer failed\n");
        wildcard_cleanup(TestInfo.DeviceNum, &hDMASource, &
            hDMADestination);
        return NULL;
      }

      command_history_message("waiting for transfer\n");
      rc=WC_IntWait(TestInfo.DeviceNum, INT_TIMEOUT_MS); //wait for
          interrupt, signifying all polarizations have been received.
          Loop around to begin gathering next set
      if(rc != WC_SUCCESS) {
        WC_PeRegRead(TestInfo.DeviceNum, CTRL_REG_BASE, 8, debugReg);
        command_history_message("Interrupt timed out; int Counter,
            numNeighbours, dram_addr, dma_incount, state, in_fifo_out,
            DRAM data, DMA_outCount: %x %x %x %x %x %x %x %x\n",
          debugReg[0], debugReg[1], debugReg[2], debugReg[3], debugReg
              [4], debugReg[5], debugReg[6], debugReg[7]);
        wildcard_cleanup(TestInfo.DeviceNum, &hDMASource, &hDMADestination);
        return NULL;
      }
      command_history_message("interrupt received\n");
      rc=WC_IntEnable(TestInfo.DeviceNum, FALSE); //reset interrupt
      command_history_message("interrupt disabled\n");
      rc=WC_IntReset(TestInfo.DeviceNum);
      command_history_message("interrupt reset\n");
      rc=WC_IntEnable(TestInfo.DeviceNum, TRUE);
      command_history_message("interrupt enabled\n");
      //go on to next cell (continue with loop):
      }
      }

      command_history_message("waiting for calculation (after loop)\n");
      rc=WC_IntWait(TestInfo.DeviceNum, INT_TIMEOUT_MS); //wait for interrupt, signifying last polarization has been calculated
      if(rc != WC_SUCCESS) {
        WC_PeRegRead(TestInfo.DeviceNum, CTRL_REG_BASE, 8, debugReg);
        command_history_message("Interrupt timed out; int Counter, numNeighbours, dram_addr, dma_incount, state, in_fifo_out, DRAM data, DMA_outCount: %x %x %x %x %x %x %x %x\n",
          debugReg[0], debugReg[1], debugReg[2], debugReg[3], debugReg[4], debugReg[5], debugReg[6], debugReg[7]);
        wildcard_cleanup(TestInfo.DeviceNum, &hDMASource, &hDMADestination);
        return NULL;
      }
      rc=WC_IntEnable(TestInfo.DeviceNum, FALSE); //reset interrupt
      rc=WC_IntReset(TestInfo.DeviceNum);
      rc=WC_IntEnable(TestInfo.DeviceNum, TRUE);
      WC_PeRegWrite(TestInfo.DeviceNum, CTRL_REG_BASE+3, 1, &totem); //send a totem value to tell the PE that all cells have been calculated
      if (rc != WC_SUCCESS) {
        command_history_message("Register transfer failed\n");
        wildcard_cleanup(TestInfo.DeviceNum, &hDMASource, &hDMADestination);
        return NULL;
      }

      calculatedCellNum = 0; //this variable is used to generate the index for the relevant data in DMABufDest
      //since it is a smaller size than number_of_cells_in_layer
      command_history_message("sorting out results\n");
```

```
          for (icCellsInLayer = 0 ; icCellsInLayer <
              number_of_cells_in_layer[icLayers] ; icCellsInLayer++)
      { cell = sorted_cells[icLayers][icCellsInLayer] ;
        if (!((QCAD_CELL_INPUT == cell->cell_function)||(QCAD_CELL_FIXED
            == cell->cell_function)))
      { //this condition must be here, as it was for these lines in the
          original version
        //tolerance testing could be done in hardware fairly easily, but
            doing alongside the data replacement makes sense
        current_cell_model = ((bistable_model *)cell->cell_model) ;
        old_polarization = current_cell_model->polarization;
        current_cell_model->polarization = DMABufDest[calculatedCellNum
            ]; //eliminated need for *(float *)& casting by changing
            buffer's type
        command_history_message("new polarization: %e\n", DMABufDest[
            calculatedCellNum]);
        stable = (fabs(DMABufDest[calculatedCellNum] - old_polarization)
            <= tolerance) && stable;
        DMABufDest[calculatedCellNum] = 1;
        calculatedCellNum++;
      }
      }
        }
      }//WHILE !STABLE

    if (VECTOR_TABLE == SIMULATION_TYPE)
      for (design_bus_layout_iter_first (design_bus_layout, &bli,
          QCAD_CELL_INPUT, &i) ; i > -1 ; design_bus_layout_iter_next (&
          bli, &i))
        if (!exp_array_index_1d (pvt_inputs, VT_INPUT, i).active_flag)
          sim_data->trace[i].data[j] = ((bistable_model *)
              exp_array_index_1d (pvt_inputs, VT_INPUT, i).input->
              cell_model)->polarization;

    // -- collect all the output data from the simulation -- //
    for (design_bus_layout_iter_first (design_bus_layout, &bli,
        QCAD_CELL_OUTPUT, &i) ; i > -1 ; design_bus_layout_iter_next (&
        bli, &i))
      sim_data->trace[design_bus_layout_inputs_icUsed + i].data[j] = ((
          bistable_model *)exp_array_index_1d (design_bus_layout_outputs
          , BUS_LAYOUT_CELL, i).cell->cell_model)->polarization;

    // -- if the user wants to stop the simulation then exit. -- //
    if(TRUE == STOP_SIMULATION)
      j = sim_data_number_samples ;

```

```
  }//for number of samples

  // Free the neigbours and Ek array introduced by this simulation//
  for (k = 0; k < number_of_cell_layers; k++)
    {
#ifdef REDUCE_DEREF
    number_of_cells_in_current_layer = number_of_cells_in_layer[k] ;
    for (l = 0 ; l < number_of_cells_in_current_layer ; l++)
#else
    for (l = 0 ; l < number_of_cells_in_layer[k] ; l++)
#endif
      {
      g_free(((bistable_model *)sorted_cells[k][l]->cell_model)->
          neighbours);
      g_free(((bistable_model *)sorted_cells[k][l]->cell_model)->
          neighbour_layer);
      g_free(((bistable_model *)sorted_cells[k][l]->cell_model)->Ek);
      }
    }

  simulation_inproc_data_free (&number_of_cell_layers, &
      number_of_cells_in_layer, &sorted_cells) ;

  wildcard_cleanup(TestInfo.DeviceNum, &hDMASource, &hDMADestination);

// Restore the input flag for the inactive inputs
  if (VECTOR_TABLE == SIMULATION_TYPE)
    for (i = 0 ; i < pvt_inputs_icUsed ; i++)
      exp_array_index_1d (pvt_inputs, VT_INPUT, i).input->cell_function
          = QCAD_CELL_INPUT ;

// -- get and print the total simulation time -- //
  if ((end_time = time (NULL)) < 0)
    fprintf(stderr, "Could not get end time\n");

  command_history_message ("Total simulation time: %g s\n", (double)(
      end_time - start_time));
  set_progress_bar_visible (FALSE) ;

#ifndef REDUCE_DEREF
  #undef sim_data_number_samples
  #undef pvt_inputs
  #undef pvt_inputs_icUsed
  #undef pvt_vectors
  #undef pvt_vectors_icUsed
  #undef design_bus_layout
  #undef design_bus_layout_inputs
  #undef design_bus_layout_inputs_icUsed
  #undef design_bus_layout_outputs
  #undef design_bus_layout_outputs_icUsed
#endif

  return sim_data;
  }//run_bistable_hardware
```

Listing B.1: Bistable simulation engine software code with hardware interaction

Appendix C

qcad VHDL Code Listing

The following is the code for the top module of the QCA simulating hardware module.

```
library IEEE;
use IEEE.std_logic_arith.all;
use IEEE.std_logic_1164.all;
use IEEE.std_logic_unsigned.all;

library WILDCARD4_LIB;
use WILDCARD4_LIB.pe_package.all;
use WILDCARD4_LIB.lad_tools_package.all;

library UNISIM;
use UNISIM.vcomponents.all;

architecture qcad of pe is

--component declarations
component p_math
    Port ( pol : in  STD_LOGIC_VECTOR (31 downto 0);
           gamma : in  STD_LOGIC_VECTOR (31 downto 0);
           Ek : in  STD_LOGIC_VECTOR (31 downto 0);
           start : in  STD_LOGIC;
           clk : in  STD_LOGIC;
        ce : in std_logic;
        new_cell : in std_logic;
           reset : in  STD_LOGIC;
           new_pol : out  STD_LOGIC_VECTOR (31 downto 0);
           done : out  STD_LOGIC);
end component;

component fifo
  port (
  clk: IN std_logic;
  din: IN std_logic_VECTOR(31 downto 0);
  rd_en: IN std_logic;
  rst: IN std_logic;
  wr_en: IN std_logic;
  dout: OUT std_logic_VECTOR(31 downto 0);
  empty: OUT std_logic;
  full: OUT std_logic);
end component;

component out_fifo
  port (
  clk: IN std_logic;
  din: IN std_logic_VECTOR(31 downto 0);
  rd_en: IN std_logic;
  rst: IN std_logic;
  wr_en: IN std_logic;
  dout: OUT std_logic_VECTOR(31 downto 0);
  empty: OUT std_logic;
  full: OUT std_logic);
end component;

component DRAM_fifo
  port (
  din: IN std_logic_VECTOR(127 downto 0);
  rd_clk: IN std_logic;
  rd_en: IN std_logic;
  rst: IN std_logic;
  wr_clk: IN std_logic;
  wr_en: IN std_logic;
  dout: OUT std_logic_VECTOR(127 downto 0);
  empty: OUT std_logic;
  full: OUT std_logic;
  prog_empty: OUT std_logic);
end component;

--wildcard interface signals
constant STARTUP_WAIT         : natural                         := 8000;
                    -- Use for synthesis
--constant STARTUP_WAIT           : natural                          := 16;
                        -- Use for simulation
signal reset                  : std_logic := '0';
signal clocks_in              : clocks_in_type;
signal clocks_out             : clocks_out_type;
signal lad_in                 : lad_in_type;
signal lad_out                : lad_out_type;

```

```
--data input/output for p_math unit
signal Ek_in, gamma_in : std_logic_vector(31 downto 0);

--control signals for various aspects
signal p_math_nd, cell_done, p_math_con, p_math_en : std_logic;

--state machine for core operation
type core_states is (idle, setup_Ek_DMA, Ek_DMA_init1, Ek_DMA_init2,
    Ek_to_DRAM,
    next_Ek_DMA, last_Ek_to_DRAM, get_regs, totem_wait, setup_pol_DMA,
        pol_DMA_init1,
   pol_DMA_init2, DRAM_in_start, dumbState, calculate1, calculate2,
       next_cell, nextCell_int,
   setup_deliver, deliver_init);
signal core_state : core_states;

--DRAM signals
signal dram_in                   : dram_in_type;
signal dram_out                  : dram_out_type;
signal dram_read, dram_read_sel  : std_logic;
signal dram_write                : std_logic;
signal dram_buf_write, dram_super_write : std_logic; --write control for
    dram input buffer
signal dram_addr                 : std_logic_vector(24 downto 0);
signal dram_write_addr, dram_read_addr : std_logic_vector(22 downto 0);
signal dram_data_out             : std_logic_vector(127 downto 0);
signal dram_data_in_valid        : std_logic;
signal dram_data_in              : std_logic_vector(127 downto 0);
signal dram_write_rdy            : std_logic;
signal dram_read_rdy             : std_logic;

signal dram_fifo_rd_cnt : std_logic_vector(8 downto 0);

--state machine control signals
signal smBus_req, smDMA_init, DMAfromPE : std_logic;
signal smLAD_data_out : std_logic_vector(31 downto 0);
signal DMA_From_Pe_Addr    : std_logic_vector(29 downto 0)     := (others
    => '0');
signal DMA_To_Pe_Addr      : std_logic_vector(29 downto 0)     := (others
    => '0');
signal DMA_inCount, DMA_outCount, numCells_DMA, numNeighbours_DMA :
    std_logic_vector(31 downto 0);
signal DMA_inCount_clr, DMA_outCount_clr : std_logic;
signal wait_count : std_logic;
signal wait_counter : std_logic_vector(5 downto 0);

signal dram_addr_inc, dram_addr_clr, dram_realAddr_inc : std_logic;
signal dram_tailAddr_inc : std_logic;
signal pN_delay : std_logic_vector(3 downto 0); --perNeighbour delay
signal pN_count, pN_clr : std_logic;
signal i_clock_prev : std_logic; --maybe bad practice, but used to
    detect i_clock edges
--signal sram_addr_inc, sram_addr_clr : std_logic;

--other control signals
type lad_register_vector is array (natural range<>) of std_logic_vector
    (31 downto 0);
signal control_register : lad_register_vector(0 to 7);
signal debug_reg : lad_register_vector(0 to 7);
signal reg_strobe_out : std_logic;
signal reg_LAD_out : std_logic_vector(31 downto 0);
signal Reg_Index : natural := 0;
signal Start_DMA, gamma_set : std_logic;
signal generate_interrupt, pci_rdy, Interrupt, Interrupt_Done :
    std_logic;
signal clear_interrupt : std_logic;
signal int_counter : std_logic_vector(4 downto 0);

--controls to and from the fifo units
signal in_fifo_rd, in_fifo_empty, in_fifo_full : std_logic;
signal in_fifo_out : std_logic_vector(31 downto 0);
signal out_fifo_rd, out_fifo_empty, out_fifo_full, out_fifo_prog_full :
    std_logic;
signal out_fifo_out, out_fifo_in : std_logic_vector(31 downto 0);
signal dram_fifo_rd, dram_fifo_empty, dram_fifo_prog_empty,
    dram_fifo_full : std_logic;
signal dram_fifo_clr : std_logic;
signal dram_fifo_out : std_logic_vector(127 downto 0);

alias p_clock          : std_logic is clocks_in.p_clock.clock;
alias i_clock          : std_logic is lad_in.clock_in;

constant REG_BASE       : std_logic_vector(15 downto 0) := x"0100";
constant REG_MASK       : std_logic_vector(15 downto 0) := x"FFFF";
constant REG_BASE_ADDRESS    : std_logic_vector(15 downto 0)     := x
    "1000";
constant REG_BASE_MASK       : std_logic_vector(15 downto 0)     := x"
    FFF8";

begin

polarization_math : p_math
```

```
Port map( pol => in_fifo_out ,
    gamma => gamma_in, --from register
    Ek => Ek_in , --from DRAM buffer thing
    start => p_math_nd,
    clk => i_clock ,
    ce => p_math_en ,
    new_cell => p_math_con ,
    reset => reset ,
    new_pol => out_fifo_in ,
    done => cell_done);

in_fifo : fifo --fifo storage for incoming data; size is 32x1024
    port map (
      clk => i_clock ,
      din => lad_in.data_in ,
      rd_en => in_fifo_rd ,
      rst => reset ,
      wr_en => lad_in.DMA_strobe,
      dout => in_fifo_out , --goes to DRAM buffer thing (sort out which
         part of the 128 bit word to use)
      empty => in_fifo_empty ,
      full => in_fifo_full);

away : out_fifo --fifo for outgoing data; size is 32x256
--size was enlarged from 32x32 to deal with laggy full signals and
   prevent data overflow
    port map (
      clk => i_clock ,
      din => out_fifo_in ,
      rd_en => out_fifo_rd ,
      rst => reset ,
      wr_en => cell_done ,
      dout => out_fifo_out ,
      empty => out_fifo_empty ,
      full => out_fifo_full);

out_fifo_rd <= (not out_fifo_empty) and lad_in.bus_gnt and lad_in.
   pci_rdy;

DRAM_output_fifo : DRAM_fifo --fifo for DRAM output; size is 128x256
    port map (
      din => dram_data_in ,
      rd_clk => i_clock ,
      rd_en => dram_fifo_rd , --controlled from state machine when data
         is needed
```

```
        rst => dram_fifo_clr , --controlled in the state machine so empty
            flag resets
        wr_clk => p_clock ,
        wr_en => dram_data_in_valid ,
        dout => dram_fifo_out , --goes to DRAM buffer thing (the thing that
             sorts out which part of the 128 bit word to use)
        empty => dram_fifo_empty ,
        full => dram_fifo_full ,
        prog_empty => dram_fifo_prog_empty); --set to turn on when less
            than 50 data are present
        --this is to ensure this fifo communicates effectively with the
            DRAM control
        --to ensure data does not over- or underflow
        --because data must always be present to meet timing
        --and the 173 cycle delay on DRAM read means
        --there must be a sizable buffer zone between stopping the read
            and the fifo being full

--DRAM output buffer thing (between DRAM output fifo and p_math input):
--governing which part of the DRAM word goe to the p_math core based on
    the address bits
--this may need to be modified if we are using two p_math cores
Ek_in <= dram_fifo_out(127 downto 96) when (dram_addr(1 downto 0) =
    "11") else
        dram_fifo_out(95 downto 64) when (dram_addr(1 downto 0) = "10")
            else
        dram_fifo_out(63 downto 32) when (dram_addr(1 downto 0) = "01")
            else
        dram_fifo_out(31 downto 0) when (dram_addr(1 downto 0) = "00")
            else
        (others => '0');

--dram input buffer thing: DRAM write only happens once at the beginning
    , in sequence,
--so every time the last two address bits are 11, the previous three
    data slots have been filled ,
--and we have a 128 bit word to write.
--the last set of data will need to have a special case to be sure it's
    written
--this also means that real address 0 will have no data.
DRAM_input_buffer: process(i_clock , reset)
begin
  if(reset = '1') then
    dram_data_out <= (others => '0');
    dram_write <= '0';
  else
```

```
    if rising_edge(i_clock) then
      if(dram_buf_write = '1') then
        case dram_addr(1 downto 0) is
        when "00" =>
          dram_data_out(31 downto 0) <= in_fifo_out;
        when "01" =>
          dram_data_out(63 downto 32) <= in_fifo_out;
        when "10" =>
          dram_data_out(95 downto 64) <= in_fifo_out;
        when "11" =>
          dram_data_out(127 downto 96) <= in_fifo_out;
        when others =>
          dram_data_out <= dram_data_out;
        end case;

        if((dram_addr(1 downto 0) = "11") or (dram_super_write = '1'))
            then
          dram_write <= '1';
        else
          dram_write <= '0';
        end if;
      else
        dram_write <= '0';
        dram_data_out <= dram_data_out;
      end if;
    end if;
  end if;
end process;

----------------------------------------------------------------
--DMA control stuff
----------------------------------------------------------------
lad_reg_control_proc: process (reset, i_clock) is
begin
  if (reset = '1') then
    Start_DMA <= '0';
    gamma_set <= '0';
    control_register <= (others => (others => '0'));
    reg_LAD_out <= (others => '0');
    reg_strobe_out <= '0';
  elsif rising_edge (i_clock) then --make sure this is the correct clock
      to use
    if (lad_in.reg_strobe = '1') then
      if(lad_in.write = '1') then
        control_register(Reg_Index) <= lad_in.data_in;
        if(Reg_Index = 4) then
```

```
          Start_DMA <= '1'; --start DMA on third transfer
        elsif(Reg_Index = 3) then
          gamma_set <= '1';
        end if;
      else
        reg_LAD_out <= debug_reg(Reg_Index);
        reg_strobe_out <= '1';
      end if;
    else
      gamma_set <= '0';
      start_DMA <= '0';
      reg_strobe_out <= '0';
      reg_LAD_out <= (others => '0');
    end if;
  end if;
end process lad_reg_control_proc;

Reg_Index    <= CONV_INTEGER('0' & lad_in.Addr(2 downto 0));

DMA_From_PE_Addr  <= control_register(0)(31 downto 2);
DMA_To_PE_Addr    <= control_register(1)(31 downto 2);
numCells_DMA <= control_register(2); --DMA from PE count; the number of cells
gamma_in   <= control_register(3);
numNeighbours_DMA <= X"7fffffff" and control_register(4); --DMA to PE count; the number of neighbours a cell has
--the gamma and neighbours registers are arranged in this order to streamline the transfer
--of gamma followed by the number of neighbours for starting a polarization transfer
--ANDed with this value because of dumb things. (see comment for dumbState in the transitions process)

--debug_reg(2) <= dram_fifo_empty & control_register(4)(31) & gamma_set & start_DMA & "000" & dram_addr;
debug_reg(4) <= X"00000000" when core_state = idle else
  X"00000001" when core_state = setup_Ek_DMA else
  X"00000002" when core_state = Ek_DMA_init1 else
  X"00000003" when core_state = Ek_DMA_init2 else
  X"00000004" when core_state = Ek_to_DRAM else
  X"00000005" when core_state = next_Ek_DMA else
  X"00000006" when core_state = last_Ek_to_DRAM else
  X"00000007" when core_state = get_regs else
  X"00000008" when core_state = totem_wait else
  X"00000009" when core_state = setup_pol_DMA else
  X"0000000a" when core_state = pol_DMA_init1 else
  X"0000000b" when core_state = pol_DMA_init2 else
  X"0000000c" when core_state = DRAM_in_start else
  X"0000000d" when core_state = dumbState else
  X"0000000e" when core_state = calculate1 else
  X"0000000f" when core_state = calculate2 else
  X"00000010" when core_state = next_cell else
  X"00000011" when core_state = nextCell_int else
  X"00000012" when core_state = setup_deliver else
  X"00000013" when core_state = deliver_init else
  X"ffffffff";

debug_out : process(reset, i_clock) is
begin
  if (reset = '1') then
    debug_reg(0) <= (others => '0');
    debug_reg(1) <= (others => '0');
    debug_reg(5) <= (others => '0');
    debug_reg(3) <= (others => '0');
    debug_reg(2) <= (others => '0');
    debug_reg(7) <= (others => '0');

    dram_fifo_rd_cnt <= (others => '0');
  elsif(rising_edge(i_clock)) then
    if(core_state = calculate1) then
      debug_reg(0) <= Ek_in;
      debug_reg(2) <= "0000000000000000000000" & dram_fifo_rd_cnt;
      debug_reg(7) <= (others => '0');
      debug_reg(1) <= gamma_in;
      debug_reg(5) <= in_fifo_out;
      debug_reg(6) <= dram_fifo_out(127 downto 96);
    end if;

    if(core_state = nextCell_int) then
      debug_reg(3) <= out_fifo_in;
    end if;

    if(dram_fifo_rd = '1') then
      dram_fifo_rd_cnt <= dram_fifo_rd_cnt + 1;
    elsif(dram_fifo_clr = '1') then
      dram_fifo_rd_cnt <= (others => '0');
    end if;
  end if;
end process;

--state machine transitions
core_transitions : process (i_clock, reset)
```

```
begin
  if(reset = '1') then
    core_state <= idle;
  elsif rising_edge(i_clock) then
    case core_state is
    when idle => --this state should only happen once, when the core is
        reset
      --wait for signal to start DMA transfer of Ek data
      if(Start_DMA = '1') then
        core_state <= setup_Ek_DMA;
      end if;
    when setup_Ek_DMA => --this is the state we should loop back to for
        repeating the DMA transfer
      if (lad_in.bus_gnt = '1') then
        core_state      <= Ek_DMA_init1;
      end if;
    when Ek_DMA_init1 =>
      core_state <= Ek_DMA_init2;
    when Ek_DMA_init2 =>
      core_state <= Ek_to_DRAM;
    when Ek_to_DRAM =>
    --shove Ek values from the FIFO into DRAM
      --Max address limit is irrelevant because the amount of data is
          never that high
      --get number of neighbours from the host in the RegWrite that
          triggers things
      --no check for dram_write_rdy here because in_fifo is read only
          when that signal is high
      --so in_fifo_empty will activate after the complete set of *
          successful* DRAM writes
      if((DMA_inCount >= numNeighbours_DMA) and (in_fifo_empty = '1'))
          then
        core_state <= next_Ek_DMA; --start new DMA for more Ek data...
      end if;
    when next_Ek_DMA =>
      --will need to send an interrupt here
      if(Start_DMA = '1') then
        core_state <= setup_Ek_DMA;
      elsif(gamma_set = '1') then
        core_state <= last_Ek_to_DRAM;
      end if;
    when last_Ek_to_DRAM =>
      if(dram_write_rdy = '1') then --if dram_write_rdy is 1, then the
          thing has happened and we can move on
        core_state <= setup_deliver;
      end if;
    when setup_deliver =>
    --open the channel to deliver data back to host (DMA from the SRAM)
      if(lad_in.bus_gnt = '1') then
        core_state <= deliver_init;
      end if;
    when deliver_init =>
      core_state <= totem_wait;
    when totem_wait => --a wait state for between iterations
      if(gamma_in /= X"BF000000" and wait_counter >= X"4") then
        core_state <= get_regs;
      end if;
    when get_regs => --get gamma and neighbour values from host for
        current cell in register transfer
    --this is the state we start from every time after we have Ek data
      --wait until we have gamma and neighbours
      if(DMA_outCount = X"00000000" or start_DMA = '1') then
        core_state <= setup_pol_DMA;
      elsif(gamma_set = '1' and gamma_in = X"BF000000") then
      --test here for whether the last cell is done or not
      --test must be done in the first transfer or it will cause a false
          positive
        core_state <= setup_deliver;
      end if;
    when setup_pol_DMA => --this is for v1 of the hardware
      --wait for the bus grant, like in the Ek DMA setup
      if (lad_in.bus_gnt = '1') then
        core_state      <= Pol_DMA_init1;
      end if;
    --it would be nice to keep fetching more polarization data as soon
        as all the current stuff goes in
    --but I think it would require extracting the DMA control into a
        seperate machine
    when pol_DMA_init1 =>
      core_state <= pol_DMA_init2;
    when pol_DMA_init2 =>
      core_state <= DRAM_in_start;
    when DRAM_in_start => --we have all Ek, gamma, and Polarizations are
         on their way, so we can start calculating the new polarization
    --this state should request data from DRAM, then move to a wait
        state
    --DRAM address in this and subsequent states should be incrementing
        continuously until some limit...
      if(dram_fifo_empty = '0' and interrupt_done = '1') then
      --make sure DRAM data is ready and that polarization data has come
          through the fifo
      --the particulars of the logic at play here mean that the in_fifo
          must be at least
      --as large as the largest DMA input transaction
        if(control_register(4)(31) = '1' or DMA_outCount /= X"0") then
        --do dumb things on the first cell of any sample but the first.
          core_state <= calculate1;
        else
          core_state <= dumbState;
        end if;
      end if;
    when dumbState =>
    --there is a dumb thing that happens where the DRAM doesn't actually
         read the address it's told to read...
    --Upon doing read(1) on anything but the first sample, the first
        data_in_valid would be associated
    --with all zeroes on the data line (presumably read(0), but I don't
        know).
    --in order to compensate for this dumb thing that happens, this dumb
         state was added
    --to flush that first worthless word out of the dram_fifo when
        needed.
    --The need to do this is indicated by having a zero in the MSB of
        control_register(4) when sent
    --from the host. This means also that the MSB of control_register(4)
        (which recieves numNeighbours)
    --must be anded with 7fffffff in order to make numNeighbours a
        normal value for use in
    --DMA initialization
    --Anyway this is dumb. I don't know why it needs to happen like this
        , I can't explain the simulated
    --behaviour of the DRAM, and it seems to happen the same in hardware
        ...
    --it may have something to do with the write.
      core_state <= calculate1;
    when calculate1 =>
    --first data ready from DRAM fifo, start calculation
      --if(dram_fifo_empty = '0' and in_fifo_empty = '0') then --make
          sure data is received before moving on
      --the functionality of the p_math core is dependent upon precise
          and reliable timing
      --ZERE CAN BE NO VAITINK!
      --(*ahem* there can be no waiting) Although extra waiting around
          between cells is acceptable
      --the question of the moment is: is it safe to assume the DRAM
          will reliabley have new data ready?
      core_state <= calculate2;
```

```
      --end if;
    when calculate2 =>
    --this is the wait state; it should have two cycles
      if(wait_counter = X"1") then
        --how to determine when to move on to the next cell...
        if(in_fifo_empty = '1') then
          core_state <= next_cell;
        else
          core_state <= calculate1;
        end if;
      end if;
    when next_cell =>
    --p_math core prepped for next stuff...
    --new_pol from p_math should be automatically sent when core is
        finished,
    --rather than being controlled explicitly here
      if(cell_done = '1') then --wait around until calculation is done
        core_state <= nextCell_int;
      end if;
    when nextCell_int => --state for generating interrupt
      if(interrupt_done = '1' and start_DMA = '0') then
        core_state <= get_regs;
      end if;
    when others =>
      core_state <= idle;
    end case;
  end if;
end process;

core_control : process (core_state, DMA_To_Pe_Addr, numNeighbours_DMA,
  dram_write_rdy, in_fifo_empty, dram_addr, dram_read_rdy,
      dram_fifo_full,
  DMA_inCount, DMA_From_Pe_Addr, DMA_outCount, numCells_DMA,
      out_fifo_full,
  out_fifo_empty, cell_done, dram_fifo_prog_empty)
begin
  dram_addr_inc <= '0';
  dram_realAddr_inc <= '0';
  dram_tailAddr_inc <= '0';
  dram_addr_clr <= '0';
  dram_buf_write <= '0';
  dram_super_write <= '0';
  dram_read <= '0';
  dram_fifo_rd <= '0';
  dram_fifo_clr <= '0';
  p_math_nd <= '0';
```

```
p_math_en <= '0';
p_math_con <= '0';
smbus_req <= '0';
smdma_init <= '0';
smLAD_Data_out <= (others => '0');
DMA_inCount_clr <= '0';
DMA_outCount_clr <= '0';
in_fifo_rd <= '0';
pn_clr <= '0';
pn_count <= '0';
generate_interrupt <= '0';
clear_interrupt <= '0';
wait_count <= '0';
dram_read_sel <= '0';

case core_state is
when idle =>
  clear_interrupt <= '1';
  DMA_inCount_clr <= '1';
  DMA_outCount_clr <= '1';
  dram_fifo_clr <= '1';
when setup_Ek_DMA =>
  smBus_Req <= '1'; --request LAD bus for DMA initialization
when Ek_DMA_init1 =>
  smBus_Req          <= '1';
  smdma_init         <= '1';
  smLAD_Data_Out     <= DMA_TO_PE_Addr & "00";
when Ek_DMA_init2 =>
  smbus_req <= '1';
  smdma_init <= '1';
  smLAD_Data_Out <= numNeighbours_DMA;
  DMA_inCount_clr <= '1';
  clear_interrupt <= '1';
when Ek_to_DRAM => --shove Ek values from the FIFO into DRAM
--we could extract this out to be an autonomous control structure, so
    that it could continue
--running while new data is being fetched, but "in theory" new data
    should be stored immediately
--so there should be no waiting
  --data has to be written 128 bits at a time, this is handled by the
      DRAM input buffer
  if (DRAM_write_rdy = '1' and in_fifo_empty = '0') then
    dram_buf_write <= '1'; --control sent to the DRAM input buffer
    in_fifo_rd <= '1';
  end if;
when next_Ek_DMA => --wait for next totem, and fire interrupt
```

```
    generate_interrupt <= '1';
  when last_Ek_to_DRAM => --this is for making sure the last set of data
      in the DRAM input buffer gets written
  --because if it does not end on a "11" it will otherwise just stay in
     the buffer and not get written
    if(dram_addr(1 downto 0) /= "00" and DRAM_write_rdy = '1') then
    --indicating that the last address was not ending in "11"
      dram_super_write <= '1';
      dram_buf_write <= '1';
      dram_realAddr_inc <= '1';
      in_fifo_rd <= '1';
    end if;
  when setup_deliver =>
  --deliver data back to host (DMA from the SRAM)
    smBus_req <= '1';
    clear_interrupt <= '1';
    dram_read_sel <= '1';
  when deliver_init =>
      smBus_Req           <= '1';
      smdma_init          <= '1';
      smLAD_Data_Out      <= DMA_From_PE_Addr & "01";
    dram_read_sel <= '1';
  when totem_wait =>
    wait_count <= '1';
    DMA_outCount_clr <= '1';
    dram_fifo_clr <= '1';
    dram_addr_clr <= '1'; --all data has been put in DRAM, reset the
        address for reading
    dram_read_sel <= '1';
  when get_regs =>
    p_math_en <= '1';
    p_math_con <= '1';
    dram_read_sel <= '1';
  when setup_pol_DMA =>
    smBus_Req <= '1'; --request LAD bus for DMA initialization
    p_math_en <= '1';
    dram_read_sel <= '1';
  when pol_DMA_init1 =>
    smBus_Req           <= '1';
    smdma_init          <= '1';
    smLAD_Data_Out      <= DMA_TO_PE_Addr & "00";
    p_math_en <= '1';
    dram_read_sel <= '1';
  when pol_DMA_init2 =>
    smbus_req <= '1';
    smdma_init <= '1';
    smLAD_Data_Out <= numNeighbours_DMA;
    DMA_inCount_clr <= '1';
    p_math_en <= '1';
    clear_interrupt <= '1';
    dram_read_sel <= '1';
  when DRAM_in_start => --we have all Ek, gamma, and Polarizations, so we can start calculating the new polarization
  --this state should request data from DRAM, then move to a wait state
  --DRAM address in this and subsequent states should be incrementing continuously until some limit...
    if (DRAM_read_rdy = '1' and dram_fifo_prog_empty = '1') then
      dram_read <= '1';
    end if;
    p_math_en <= '1';

    --fire the interrupt here, allowing the host to gather the next set of polarizations
    --while the calculation is being done.
    --need to fire interrupt again when cell calculation is done to tell host
    --it is ok to send new gamma and neighbours
    if(DMA_inCount >= numNeighbours_DMA) then --this condition is used in a few places...
      generate_interrupt <= '1';
    end if;
    dram_read_sel <= '1';
  when dumbState =>
    if (DRAM_read_rdy = '1' and dram_fifo_prog_empty = '1') then
      dram_read        <= '1';
    end if;
    dram_fifo_rd <= '1';
    p_math_en <= '1';
    dram_read_sel <= '1';
  when calculate1 =>
  --first data ready from DRAM fifo, start calculation
    if (DRAM_read_rdy = '1' and dram_fifo_prog_empty = '1') then --be careful with this read_rdy signal and p_math_nd....
      dram_read <= '1';
    end if;
    p_math_nd <= '1';
    p_math_en <= '1';
    dram_tailAddr_inc <= '1';
    in_fifo_rd <= '1';
    if(dram_addr(1 downto 0) = "11") then --when the fourth 32 bit word has been read, get next 128 bit word
      dram_fifo_rd <= '1';
    end if;
    dram_read_sel <= '1';
  when calculate2 =>
  --currently the p_math unit is configured for a 2-latency accumulator
     ...
  --this is the downcycle
    if (DRAM_read_rdy = '1' and dram_fifo_prog_empty = '1') then --be
        careful with this read_rdy signal and p_math_nd....
      dram_read        <= '1';
    end if;
    p_math_en <= '1';
    wait_count <= '1';
    dram_read_sel <= '1';
  when next_cell =>
    if (DRAM_read_rdy = '1' and dram_fifo_prog_empty = '1') then --be
        careful with this read_rdy signal and p_math_nd....
      dram_read        <= '1';
    end if;
    p_math_en <= '1';
    clear_interrupt <= '1';
    dram_read_sel <= '1';
  when nextCell_int => --interrupt to inform that the cell has been
      calculated
    if (DRAM_read_rdy = '1' and dram_fifo_prog_empty = '1') then --this
        is just running all the time to make sure data is present
    --stopping the DRAM transfer only after fifo_full is detected, may
        cause data loss...
    --BECAUSE TIMING IS SUPER CRITICAL WE MUST HAVE DATA
      dram_read        <= '1';
    end if;
    generate_interrupt <= '1';
    p_math_en <= '1';
    dram_read_sel <= '1';
  end case;
end process;

process (reset, p_clock, i_clock) is
begin
  if(reset = '1') then
    dram_read_addr <= (others => '0');
    dram_write_addr <= (others => '0');
    dram_addr(1 downto 0) <= "00";
  else
    if rising_edge(p_clock) then
      if(dram_read = '1') then
        dram_read_addr <= dram_read_addr + 1;
      end if;

      if(dram_addr_clr = '1') then
        --this is the address where the first data is, and it is only
            set after DRAM is filled
        --because the nature of how the DRAM input buffer works means
            that address zero is not written
        --so we just skip over it when we go to reads.
        --(the global reset is enough to set the initial writing address
            to 0.)
        dram_read_addr <= "0000000000000000000000001";
      end if;
    end if;

    if rising_edge(i_clock) then
      if (dram_buf_write = '1' and dram_addr(1 downto 0) = "11") then
        dram_write_addr <= dram_write_addr + 1;
      end if;

      if(dram_addr_clr = '1') then
        dram_write_addr <= "0000000000000000000000001";
      end if;

      --it is necessary to separate these two out at some point, because
          while streaming the DRAM output
      --we must keep the real address going up as fast as possible to
          ensure presense of data,
      --in order to keep up with the timing concerns of the p_math core.
      --but then the two LSBs get out of sync from where we want them,
          which is pointing to
      --the next data coming out of the DRAM fifo
      if(dram_tailAddr_inc = '1' or dram_buf_write = '1') then
        dram_addr(1 downto 0) <= dram_addr(1 downto 0) + 1;
      end if;

      if(dram_addr_clr = '1') then
        dram_addr(1 downto 0) <= "00";
      end if;
    end if;
  end if;
end process;

--this is so we can have the address controlled by two different clocks
dram_addr(24 downto 2) <= dram_read_addr when dram_read_sel = '1' else
                  dram_write_addr;

```

```
--be wary of clocks....
wait_counter_proc: process (reset, i_clock) is
begin
  if (reset = '1') then
    DMA_inCount <= (others => '0');
    DMA_outCount <= (others => '0');
    wait_counter <= (others => '0');
  elsif rising_edge (i_clock) then
  --this is synchronized to i_clock because they are tracking DMA
      transactions over the LAD bus
    if(lad_in.DMA_strobe = '1') then
      DMA_inCount <= DMA_inCount + 1;
    elsif(DMA_inCount_clr = '1') then
      DMA_inCount <= (others => '0');
    end if;

    if(DMAfromPE = '1') then
      DMA_outCount <= DMA_outCount + 1;
    elsif(DMA_outCount_clr = '1') then
      DMA_outCount <= (others => '0');
    end if;

    if(wait_count = '1') then
      wait_counter <= wait_counter + 1;
    else
      wait_counter <= (others => '0');
    end if;

    DMAfromPE <= out_fifo_rd; --similar structure as the DMA example for
        outputting data
  end if;
end process wait_counter_proc;

------------------------------------------------------------
--DRAM signal registration
------------------------------------------------------------
register_signals_proc: process (p_clock, reset) is
begin
  if (reset = '1') then
    dram_out.read       <= '0';
    dram_out.write      <= '0';
    dram_out.addr       <= (others => '0');
    dram_out.data_out   <= (others => '0');
    dram_data_in        <= (others => '0');
    dram_data_in_valid  <= '0';
    dram_write_rdy      <= '0';
    dram_read_rdy       <= '0';
  elsif rising_edge (p_clock) then
    dram_out.read       <= dram_read;
    dram_out.write      <= dram_write;
    dram_out.addr       <= dram_addr(24 downto 2); --lower two bits are
       used to decide which 32 bit word to select
    dram_out.data_out   <= dram_data_out;
    dram_data_in        <= dram_in.data_in;
    dram_data_in_valid  <= dram_in.data_in_valid;
    dram_write_rdy      <= dram_in.write_rdy;
    dram_read_rdy       <= dram_in.read_rdy;
  end if;
end process register_signals_proc;

lad_out_proc: process (reset, i_clock) is
begin
  if (reset = '1') then
    lad_out.Data_Out        <= (others => '0');
    lad_out.Strobe_Out      <= '0';
    lad_out.dma_init        <= '0';
    lad_out.bus_req          <= '0';
    lad_out.pe_rdy           <= '0';
    lad_out.int_req          <= '0';
    pci_rdy                  <= '0';
  elsif rising_edge (i_clock) then
    if (DMAFromPe = '1') then --DMAfromPe will need to be controlled
       somewhere, figure that out
      lad_out.Data_Out <= out_fifo_out; --SRAM data output goes here,
         somehow
    elsif (reg_strobe_out = '1') then
      lad_out.Data_out <= Reg_LAD_out;
    else
      lad_out.Data_Out <= smLAD_Data_Out;
    end if;
    ------------------------------------------------
    -- Generate a strobe on the LAD bus for any of the following 2 cases
       : --
    -- Regs_Strobe_Out = '1' : Register read from the control register
       --
    -- DMAFromPe = '1'          : DMA-From-PE transaction data strobe
         --
    ------------------------------------------------
    lad_out.Strobe_Out <= DMAFromPe or Reg_strobe_out;
    ------------------------------------------------
    --  Allow the state machine to drive the dma_init line when
       necessary. --
    ------------------------------------------------
    lad_out.dma_init <= smdma_init;

    lad_out.bus_req <= ((not out_fifo_empty) and lad_in.pci_rdy) or
        DMAFromPE or smBus_Req;

    --  In addition to performing the DMA-to-PE initialization
        transactions   --
    --  the PE also must inform the PCI controller when it is able to
        accept  --
    --  DMA data.  This is done through the lad_out.pe_rdy line.
    lad_out.pe_rdy <= not in_fifo_full;

    lad_out.int_req <= Interrupt;

    -- Delay PCI ready by one clock to make better timing.
    pci_rdy <= lad_in.pci_rdy;
  end if;
end process lad_out_proc;

------------------------------------------------
--Interrupt Control
------------------------------------------------
int_proc: process(i_clock, reset)
begin
  if (reset = '1') then
    Int_Counter     <= (others => '0');
  elsif rising_edge (i_clock) then
    if (Generate_Interrupt = '1' and Interrupt_Done = '0' and lad_in.
        dma_in_progress = '0') then
       Int_Counter <= Int_Counter + 1;
    end if;

    if(clear_interrupt = '1') then
      int_counter <= (others => '0');
    end if;
  end if;
end process int_proc;

Interrupt_Done  <= Int_Counter(4);
Interrupt       <= Int_Counter(4) or Int_Counter(3) or Int_Counter(2) or
    Int_Counter(1) or Int_Counter(0);

-- Tie off unused Clock signals
clocks_out.p_clock.reset    <= '0';

```

```
--------------------------------------------
--instantiate control interfaces
--------------------------------------------
dram_interface_inst: dram_interface
generic map (
  STARTUP_WAIT      => STARTUP_WAIT
) port map (
  pads              => pads.dram,
  reset             => reset,
  p_clock           => clocks_in.p_clock,
  lad_clock         => i_clock,
  user_in           => dram_in,
  user_out          => dram_out
);

default_interface_inst: default_interface
port map (
  pads          => pads.default,
  reset_out     => reset,
  clocks_in     => clocks_in,
  clocks_out    => clocks_out,
  lad_in        => lad_in,
  lad_out       => lad_out
);
--------------------------------------------
--end instantiate control interfaces
--------------------------------------------


-- Do not delete/modify this line
pads    <= init_pe_pads;

end qcad;
```

Listing C.1: QCA hardware simulator code

Appendix D

p_math VHDL Code Listing

The listing that follows is the VHDL code for the main mathematical core.

```vhdl
-- use instructions: first cycle - turn on ce, provide gamma value,
    provide Ek and Pol values, pulse start signal for one cycle
--              leave start signal off for one cycle
--              provide next Ek and Pol values and pulse start signal for
    one cycle
--              leave start signal off for one cycle
--              repeat until all Ek and Pol values have been sent in
--              after pulsing start signal for the last time,
--              wait for (mult + add + 1) latency, then pulse new_cell to
    start a new cell
--              repeat instructions from first cycle
--              wait until done signal activates, retrieve data.

library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;

entity p_math is
    Port ( pol : in  STD_LOGIC_VECTOR (31 downto 0); --parallel data
        inputs for polarization and Ek
           gamma : in  STD_LOGIC_VECTOR (31 downto 0);
           Ek : in  STD_LOGIC_VECTOR (31 downto 0);
           start : in  STD_LOGIC; --pulse this with each new set of data
               input
           clk : in  STD_LOGIC;
        ce : in std_logic; --clock enable, stays on until "done"
           triggers
        new_cell : in std_logic; --pulse this when ready to send in new
           train of cell data
           reset : in  STD_LOGIC;
```

```
            new_pol : out  STD_LOGIC_VECTOR (31 downto 0); --result
               output
            done : out  STD_LOGIC); --computation complete signal
end p_math;

architecture Behavioral of p_math is

--some of these floating point units could have latencies reduced to
    free up hardware
component multiply
  port (
  a: IN std_logic_VECTOR(31 downto 0);
  b: IN std_logic_VECTOR(31 downto 0);
  operation_nd: IN std_logic;
  clk: IN std_logic;
  sclr: IN std_logic;
  ce: IN std_logic;
  result: OUT std_logic_VECTOR(31 downto 0);
  rdy: OUT std_logic);
end component;

component add
  port (
  a: IN std_logic_VECTOR(31 downto 0);
  b: IN std_logic_VECTOR(31 downto 0);
  operation_nd: IN std_logic;
  clk: IN std_logic;
  sclr: IN std_logic;
  ce: IN std_logic;
  result: OUT std_logic_VECTOR(31 downto 0);
  rdy: OUT std_logic);
end component;

component fastadder --floating point adder with latency of 1 for low-
    latency accumulation
  port (
  a: IN std_logic_VECTOR(31 downto 0);
  b: IN std_logic_VECTOR(31 downto 0);
  operation_nd: IN std_logic;
  clk: IN std_logic;
  sclr: IN std_logic;
  ce: IN std_logic;
  result: OUT std_logic_VECTOR(31 downto 0);
  rdy: OUT std_logic);
end component;

```

```
component sqrt
  port (
  a: IN std_logic_VECTOR(31 downto 0);
  operation_nd: IN std_logic;
  clk: IN std_logic;
  sclr: IN std_logic;
  ce: IN std_logic;
  result: OUT std_logic_VECTOR(31 downto 0);
  rdy: OUT std_logic);
end component;

component divider
  port (
  a: IN std_logic_VECTOR(31 downto 0);
  b: IN std_logic_VECTOR(31 downto 0);
  operation_nd: IN std_logic;
  clk: IN std_logic;
  sclr: IN std_logic;
  ce: IN std_logic;
  result: OUT std_logic_VECTOR(31 downto 0);
  rdy: OUT std_logic);
end component;

component shift_reg --32 bit shift register for use as a delay line
   Generic ( depth : integer := 50);
    Port ( data_in : in  STD_LOGIC_VECTOR (31 downto 0);
           data_out : out  STD_LOGIC_VECTOR (31 downto 0);
           clk : in  STD_LOGIC);
end component;

component onebit_shift --1 bit delay line
   Generic ( depth : integer := 10);
    Port ( data_in : in  STD_LOGIC;
           data_out : out  STD_LOGIC;
           clk : in  STD_LOGIC);
end component;

signal PxEk, sum : std_logic_vector(31 downto 0);
signal two_gamma, pol_math, p_math_squared : std_logic_vector(31 downto
   0);
signal pol_math_delayed : std_logic_vector(31 downto 0);
signal square_plus_1, SoP, den : std_logic_vector(31 downto 0);
signal in_counter, calc_counter : std_logic_vector(9 downto 0);

signal add_done, pol_math_rdy, square_done, plus1_rdy : std_logic;
signal den_rdy, accum_rdy, initial, mpy_done : std_logic;
```

```

begin

PxEk_mult : multiply --this latency must be greater than adder+1 in order for everything to work
    port map (
      a => pol, --pol and Ek from main inputs
      b => Ek,
      operation_nd => start, --signal from main inputs, should be pulsed with each new pol/Ek
      clk => clk,
      sclr => reset,
      ce => ce,
      result => PxEk,
      rdy => mpy_done);

--adder latency is reduced to 1 cycles
accum : add --accumulator, stage 1b (multiplier above is stage 1a)
    port map (
      a => PxEk,
      b => SoP,
      operation_nd => mpy_done,
      clk => clk,
      sclr => reset,
      ce => ce,
      result => sum,
      rdy => add_done);

process(clk, reset, new_cell)
begin

--here the accumulation process is handled by careful timing and control
if(reset = '1' or new_cell = '1') then
  Sop <= (others => '0');
  calc_counter <= (others => '0');
  initial <= '1'; --this signal is an indicator to whether we are at the beginning

  accum_rdy <= '0';
elsif(clk'event and clk = '1') then
  if(initial = '1' and mpy_done = '1') then
    initial <= '0';
  elsif(add_done = '1') then
    SoP <= sum; --running total updated manually, easier to manage than a simple loop
    calc_counter <= calc_counter + 1;
  elsif ((calc_counter = in_counter) and (initial = '0')) then
  --needed to keep accum_rdy from going into a permanent high
    calc_counter <= calc_counter + 1;
  end if;

  --this is where "initial" is helpful, because the two counters are
     also equal at the beginning
  if ((calc_counter = in_counter) and (initial = '0')) then
    accum_rdy <= '1';
  else
    accum_rdy <= '0';
  end if;
end if;

end process;

--this process controls "in_counter" which counts how many sets of pol/
   Ek come in
--"in_counter" is compared to "calc_counter", controlled above, to
   determine when
--the accumulation is finished.
process(clk, reset, new_cell)
begin

if(reset = '1' or new_cell = '1') then
  in_counter <= (others => '0');
elsif(clk'event and clk = '1') then
  if(start = '1') then
    in_counter <= in_counter + 1;
  end if;
end if;

end process;

--trick to double a floating point number, to avoid using an extra
   multiplier
--this method will only cause issue when gamma > 1.7e38, which it never
   is.
--saves about 200 slices
twoGamma : process(clk, reset) --gamma doubling, stage 1c
begin
  if (reset = '1') then
    two_gamma <= (others => '0');
  elsif (clk'event and clk = '1') then
    if (start = '1') then --add 1 to exponent
```

```
      two_gamma <= gamma(31) & (gamma(30 downto 23)+1) & gamma(22 downto 0);
    end if;
  end if;
end process;

divvy : divider --sum of products divided by gamma, stage 2
    port map (
      a => SoP, --result from accumulation
      b => two_gamma,
      operation_nd => accum_rdy, --signal from accumulator control to say it's done
      clk => clk,
      sclr => reset,
      ce => ce,
      result => pol_math,
      rdy => pol_math_rdy);

pol_mathDelay : shift_reg --delay line from p_math to final divider
   Generic map( depth => 6) --depth is the combined latency of parts on other path (multiplier, adder, sqrt)
    Port map( data_in => pol_math,
           data_out => pol_math_delayed,
           clk => clk);

square : multiply --square the divider result, stage 3
    port map (
      a => pol_math,
      b => pol_math,
      operation_nd => pol_math_rdy,
      clk => clk,
      sclr => reset,
      ce => ce,
      result => p_math_squared,
      rdy => square_done);

plus_one : add --add one to squared pol_math, stage 4
    port map (
      a => p_math_squared,
      b => X"3F800000", --floating point 1
      operation_nd => square_done, --_nd lines dominoing from rdy lines ...
      clk => clk,
      sclr => reset,
      ce => ce,
      result => square_plus_1,
      rdy => plus1_rdy);

make_den : sqrt --square root function for denominator, stage 5
    port map (
      a => square_plus_1,
      operation_nd => plus1_rdy,
      clk => clk,
      sclr => reset,
      ce => ce,
      result => den,
      rdy => den_rdy);

final : divider --give final result, stage 6, depends on results from
   sqrt (stage 5) and divvy (stage 2)
    port map (
      a => pol_math_delayed,
      b => den,
      operation_nd => den_rdy,
      clk => clk,
      sclr => reset,
      ce => ce,
      result => new_pol,
      rdy => done); --connected to main output

end Behavioral;
```

Listing D.1: Polarization math arithmetic core hardware code