Design and Analysis of Four Architectures for FPGA-Based Cellular Computing

Kenneth J. Morgan

Thesis submitted to the faculty of the
Virginia Polytechnic Institute and State University
in partial fulfillment of the requirements for the degree of

Master of Science
in
Electrical Engineering

James R. Armstrong, Chair
Peter M. Athanas
Mark T. Jones

October 19, 2004
Blacksburg, Virginia

Keywords: FPGA, Single-Chip Computer, Parallel Computer,
Bit-Serial, Booth Algorithm, Cellular Computing

Design and Analysis of Four Architectures for FPGA-Based Cellular Computing

Kenneth J. Morgan

Abstract

The computational abilities of today's parallel supercomputers are often quite impressive, but these machines can be impractical for some researchers due to prohibitive costs and limited availability. These researchers might be better served by a more personal solution such as a "hardware acceleration" peripheral for a PC. FPGAs are the ideal device for the task: their configurability allows a problem to be translated directly into hardware, and their reconfigurability allows the same chip to be reprogrammed for a different problem.

Efficient FPGA computation of parallel problems calls for cellular computing, which uses an array of independent, locally connected processing elements, or cells, that compute a problem in parallel. The architecture of the computing cells determines the performance of the FPGA-based computer in terms of the cell density possible and the speedup over conventional single-processor computation.

This thesis presents the design and performance results of four computing-cell architectures. MULTIPLE performs all operations in one cycle, which takes the least amount of time but requires the most chip area. BIT performs all operations bit-serially, which takes a long time but allows a large cell density. The two other architectures, SINGLE and BOOTH, lie within these two extremes of the area/time spectrum.

The performance results show that MULTIPLE provides the greatest speedup over common calculation software, but its usefulness is limited by its small cell density. Thus, the best architecture for a particular problem depends on the number of computing cells required. The results also show that with further research, next-generation FPGAs can be expected to accelerate single-processor computations as much as 22,000 times.

Acknowledgments

This work exists because of Dr. James Armstrong's vision, and he is responsible for the ideas behind the different computing architectures. I thank him for suggesting this work as a thesis topic, for becoming my advisor and guiding this research, and for providing me access to all the tools necessary for completing this project.

I would like to thank Dr. Jones for serving on my committee and taking the time to review this thesis. I would also like to thank Dr. Athanas for joining my committee on such short notice and for reviewing this thesis.

This work began as a project during the fall 2003 semester of ECE 5514. I would like to thank the fine members of team "Beta" for the professional job they did and for helping to lay the groundwork for this research.

I owe my education to my father who made a lot of sacrifices to pay for the college of my choice. I will always be indebted to him for the incredible gift of knowledge.

Finally, I would like to express my gratitude to my wonderful wife Erin for taking care of me and everything else during our time in Blacksburg.

Table of Contents

List of Figures

List of Tables

Glossary

ADS The ARM Developer Suite (ADS) is a set of software tools that target ARM hardware.

AFS The ARM Firmware Suite (AFS) is a set of libraries and source code for software that runs on ARM hardware.

AHB The ARM High-performance Bus (AHB) is a simple bus standard developed by ARM.

ASIC An Application Specific Integrated Circuit (ASIC) is a chip that is designed and manufactured to perform a specific function.

CA Cellular Automata (CA) are systems that evolve discrete cells according to simple rules.

CAD Computer-Aided Design.

CLB A Configurable Logic Block (CLB) is a common array element in FPGAs and contains programmable logic.

DIMM A Dual In-line Memory Module (DIMM) is a standard RAM form factor in computer hardware.

DIP A Dual In-line Package (DIP) is a standard shape for electronic components.

DRAM A Dynamic RAM (DRAM) is digital storage that must be refreshed to maintain its storage.

EDIF The Electronic Data Interchange Format (EDIF) is a standard file format for hardware netlists.

FF A Flip-Flop (FF) is a one-bit digital storage element.

Flash Flash memory is non-volatile digital storage on a chip.

FPGA A Field-Programmable Gate Array (FPGA) is a chip that can be programmed to perform a specific logical function.

FSM A Finite State Machine (FSM) is an abstract machine that transitions among a set of states that produce a set of outputs.

GUI A Graphical User Interface (GUI) is a visual front end to a computer program.

IDE An Integrated Development Environment (IDE) is a software application that includes a suite of development tools.

I/O Input/Output.

LC A Logic Cell (LC) is part of a CLB and contains combinational logic followed by a flip-flop.

LED A Light-Emitting Diode (LED) is an electronic visual indicator.

LM A Logic Module (LM) is an ARM development board that contains a large FPGA.

LUT A Look-Up Table (LUT) is a small memory used in FPGAs for generating an arbitrary logic function.

MAC A Multiply-Accumulate (MAC) unit is hardware that performs a multiplication followed by an addition and is often used in digital signal processors.

MPGA A Mask Programmable Gate Array (MPGA) is a chip with a regular array of transistors and custom wire connections.

PAR Place And Route (PAR) is a step during the translation of a hardware description to an FPGA configuration file.

PC Personal Computer.

PCI The Peripheral Component Interconnect (PCI) is a bus specification that is commonly implemented on computer motherboards.

RAM A Random Access Memory (RAM) is digital storage that does not have to be accessed sequentially.

RISC Reduced Instruction Set Computing (RISC) is a processor architecture that uses simplified instruction encoding and execution.

ROM A Read-Only Memory (ROM) is digital storage that is meant to be read and not written.

RTR Run-Time Reconfiguration (RTR) is a method for dynamically changing an FPGA's configuration.

SDRAM Synchronous DRAM (SDRAM) is read and written on clock edges rather than asynchronously.

SRAM A Static RAM (SRAM) is digital storage that retains its contents as long as power is applied.

SSRAM Synchronous SRAM (SSRAM) is read and written on clock edges rather than asynchronously.

VHDL The **V**ery High Speed Integrated Circuit **H**ardware **D**escription **L**anguage (VHDL) is a textual language for describing electronic hardware.

Chapter 1
Introduction

In his book *A New Kind of Science*, Stephen Wolfram demonstrates a fluid flow simulation that is computed using cellular automata, a system in which the state of each of its discrete cells is updated according to a few simple rules. Remarkably, the simulation shows eddies and complex patterns that are seen in actual streamline experiments. Wolfram goes on to suggest that most real-world systems can be modeled successfully only through methods like cellular automata, thus underscoring the importance of these methods and the machinery used to compute them efficiently. This fluid flow example and cellular automata in general are part of an important class of computational problems that can be discretized into cells that can be computed simultaneously. Other common problems in this class include finite element analysis and digital image processing.

Computing these kinds of parallel problems on a single-processor machine can quickly become impractical as the problems become large and execution times increase. The traditional approach to reducing execution time has been to compute the problem with an array of general-purpose processors connected together with a high-speed interconnection network. A notable example of this kind of parallel machine at Virginia Tech is the "X" terascale cluster [36]. Built using 1,100 Power Mac G5s and an InfiniBand network, this supercomputer cost $5.2 million, occupied 280 square meters, and achieved a computation rate of 10 trillion operations per second.

While supercomputers like the Virginia Tech "X" are able to offer impressive parallel performance, they are costly, they are usually quite large, and they often can only be used on a time-shared basis making them inappropriate for some researchers working on parallel problems like those discussed above. The computational needs of these researchers could be better met with a more personal solution such as a "hardware acceleration" expansion card that could be inserted into the researcher's personal computer and called upon to handle any cellular parallel problem.

Contrary to traditional parallel computers that use general-purpose processors combined with software to solve a problem, such a device would need its hardware to be configured specifically for a particular problem in order to minimize size and maximize performance. Additionally, the device hardware would need to be reconfigurable to allow the user to use the same device for a different problem. Such a parallel computing device is realizable because of today's high-performance, high-density field-programmable gate arrays (FPGAs). Using the programmable logic in an FPGA, a cell in a cellular parallel problem can effectively become a small-scale processor, and when cells are replicated throughout the chip, an FPGA can effectively become a parallel computer. The topic of this thesis involves the analysis of a prototype FPGA-based cellular computing system.

The purpose of this thesis is to present the design and analysis of four different computing-cell architectures in an FPGA. The architectures vary in the degree of parallelism within a cell, and they vary with respect to the methods used for basic arithmetic operations. The goal of the analysis is to determine how the architectures compare in terms of the speedup they provide over single-processor machines and the number of computing cells they allow in a single chip. Since the target user of the system is a researcher who may not be familiar with lower-level programming languages like C, the speedups compare the FPGA to higher-level languages, namely Matlab and Mathematica. The following list is an overview of the four cell architectures considered.

- *MULTIPLE*: All cell operations[1] are performed in a single clock cycle. It is referred to as MULTIPLE because each cell instantiates multiple adders and multipliers. It takes the least number of clock cycles to complete an iteration, but each cell consumes a large area on the chip.
- *SINGLE*: All cell operations are performed using a single adder and a single multiplier. It is referred to as SINGLE for that reason. It takes more clock cycles than MULTIPLE to complete an iteration because the single adder/multiplier resource must be time-shared among all operations.

[1] In this list, "cell operations" means additions and multiplications.

- *BOOTH*: All cell operations are performed using a single full-width adder. Multiplies are performed by following the Booth algorithm, and it is referred to as BOOTH for that reason.
- *BIT*: All cell operations are performed using a single bit-serial adder. It is referred to as BIT because all operations are executed one bit at a time. Like BOOTH, it uses the Booth algorithm to perform multiplies, but the algorithm is bitwise in this case. Each cell consumes a very small chip area, but it takes the most number of clock cycles to complete an iteration.

The contributions made by the thesis author include a synthesizable VHDL description of each of the architectures described above, a synthesizable VHDL interface between the FPGA and the development platform used for this research, and a Matlab/C interface for directing computed results from the FPGA to a host computer over serial links. Also, a particularly interesting achievement was the implementation of a bit-serial comparator for two's complement integers.

Background

This section gives a brief overview of FPGAs and an explanation of the numerical representation used in the architecture designs. More in-depth information on FPGAs can be found in [45] and [11].

Overview of FPGAs

Field-programmable gate arrays are chips that can be programmed to perform virtually any logic operation. They can be used in place of multiple smaller components such as glue logic, or they can contain large designs such as processors or graphics controllers. Many FPGAs can be reconfigured any number of times making them ideal for design prototyping, and they have recently been replacing ASICs and MPGAs in low-volume productions due to the high initial cost and long turnaround time of these custom manufactured chips. Their high logic capacity and abundance of flip-flops distinguish FPGAs from other kinds of programmable logic devices.

Many different architectures exist, but Figure 1-1 shows the basic structure of a typical FPGA: a matrix of configurable logic blocks (CLBs) and interconnection resources surrounded by I/O blocks. The CLBs are often complex but are likely to contain one or more function generators followed by flip-flops. Made using either look-up tables (LUTs) or multiplexers, function generators are capable of producing any *k*-input Boolean function where *k* is usually four. LUTs are 1-bit wide memories and essentially store the truth table of the Boolean function they generate. They often can be used for general storage when not acting as a function generator. The output of a function generator can serve as part of combinational logic or can be directed to a flip-flop to create a latched signal. Interconnection resources are composed of horizontal and vertical wires that can form connections with each other through the programmable switches. There are also programmable switches that connect wires to CLBs. I/O blocks can be programmed to allow their associated pin to operate as either an input or an output. Current FPGAs often include additional components such as clock managers, RAM, and dedicated circuitry for common arithmetic operations.

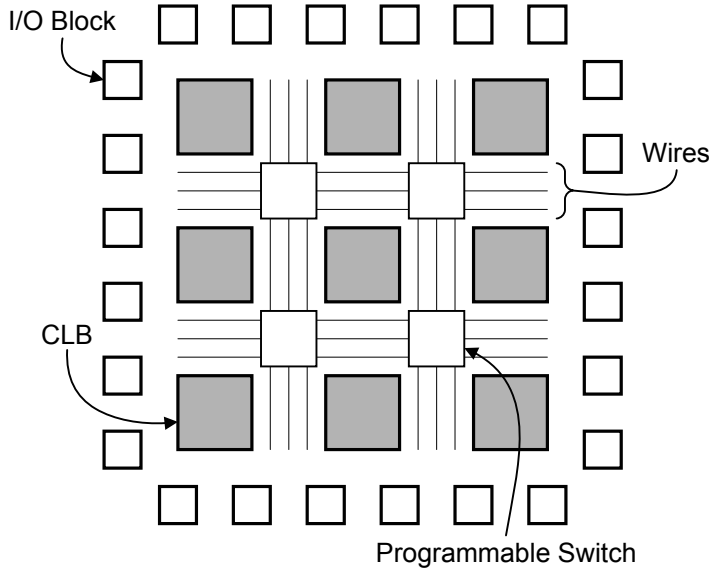

Figure 1-1. Typical FPGA structure

Due to the size and complexity of FPGAs, CAD tools must be used to take a design from its initial description to a bit stream that can be programmed on a device. Three processing steps are required: mapping, placement, and routing. During mapping,

a design's description in terms of logic gates is translated to a form suitable for function generators. During placement, CLB sites on the chip are chosen, and during routing, the necessary connections between CLBs are formed. A programming file can be generated and downloaded into the device once these steps are completed, which can take several hours for large, highly-constrained designs. The physical mechanism for programmability varies for different FPGAs with the main methods being static RAMs and antifuses.

Numerical Representation

The purpose of the FPGA-based system discussed in this thesis is to aid researchers working on scientific problems, so it must support real numbers that have both integer and fractional parts. Floating-point hardware could be used, but a fixed-point number representation is used instead in order to conserve chip area. In an effort to make each computing cell as lightweight as possible, the fixed-point numbers are scaled so that the arithmetic hardware need only deal with integers.

Given a W-bit number in this fixed-point system, a certain number of bits, S, are reserved for the integer part and sign bit, and the remaining bits, $W - S$, make up the number's fractional part. Using the scaled integer scheme, a real number $-2^{S-1} \le x < 2^{S-1}$ can be represented to a resolution of $\varepsilon = 2^{-(W-S)}$ by a two's complement integer $-2^{W-1} \le y \le 2^{W-1} - 1$, which can be converted back to a discrete real number $-2^{S-1} + \varepsilon/2 \le z \le 2^{S-1} - \varepsilon/2$. The formulas used for converting to and from fixed-point scaled integers are

$\lfloor x \cdot 2^{W-S} \rfloor$ to convert a real number x to a scaled integer, and

$\frac{y}{2^{W-S}} + \frac{\varepsilon}{2}$ to convert a scaled integer y to a real number.

The conversion to a scaled integer simply shifts all fractional bits to the left of the binary point and rounds down. The conversion back to a real number shifts the fractional bits back to their original positions. Although not strictly necessary, the $\varepsilon/2$ addition keeps the range of discrete real numbers centered on zero and guarantees that the

maximum quantization error is $\varepsilon/2$ instead of ε. An example of a conversion to and from scaled integer form is shown in Figure 1-2 for the number 2.57823 assuming $W = 8$ and $S = 4$. The figure shows how continuous ranges are mapped to single integers and how the conversion process adds quantization error.

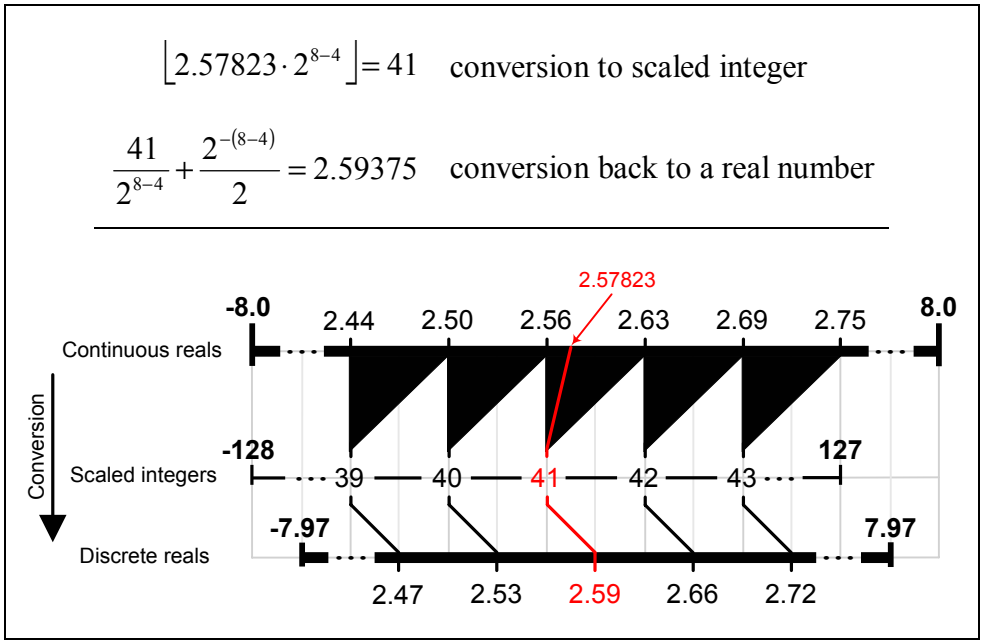

Figure 1-2. Fixed-point scaled integer conversion example with $W = 8$ and $S = 4$

There is a caveat associated with how multiplies are performed with scaled integers: If two real numbers *x* and *y* are scaled and multiplied[2], the result is

$$\left(x \cdot 2^{W-S}\right)\left(y \cdot 2^{W-S}\right) = xy \cdot 2^{2(W-S)},$$

which is no longer a correctly scaled number because the scale factor's exponent is doubled. Thus, for each multiply operation that is performed, one scale factor must be eliminated. Doing so gives a correctly scaled result:

$$\frac{\left(x \cdot 2^{W-S}\right)\left(y \cdot 2^{W-S}\right)}{2^{W-S}} = xy \cdot 2^{W-S}.$$

[2] In the following derivations, the floor operation is omitted for clarity.

It turns out that this correction is easily applied in the VHDL design descriptions and adds virtually no extra hardware. There is no such correction needed for additions since

$$x \cdot 2^{W-S} + y \cdot 2^{W-S} = (x + y) \cdot 2^{W-S}$$

gives a correctly scaled number.

There are at least two disadvantages to this fixed-point scaled integer scheme. First, a fixed-point representation does not allow numbers to vary over a wide range like floating-point does, which may prevent the computation of certain scientific problems. Second, choosing the number of integer bits, *S*, presupposes knowledge of the range of numbers that will occur in all intermediate calculations, but this information may not be available before the calculations have been performed. It is assumed, however, that most researchers can provide an accurate estimate of the range of values that occur for a given problem. *S* can also be given 1 or 2 extra "buffer bits" to avoid calculation overflow, but this comes at the expense of precision in the fractional part.

Related Work

Sipper gives an excellent introduction to cellular computing in his 1999 article [31] in IEEE Computer. He defines the three principles of cellular computing as simplicity, vast parallelism, and locality, and he outlines several application areas including fast solutions to NP-complete problems and high-quality random number generators.

Armstrong, Vick, and Scott recently published a paper [8] that included results obtained using an earlier version of the BOOTH architecture discussed in this thesis. Since the publication, the BOOTH model's calculation rate and cell density have both improved. Also referenced in [8] are results of models similar to the MULTIPLE and SINGLE architectures discussed in this thesis, but those models were not integrated into the development platform in the same way as MULTIPLE and SINGLE. Additionally, MULTIPLE and SINGLE have been redesigned from scratch using design styles similar

to those used for BOOTH and BIT so that the four architectures can be equitably compared.

Although not an FPGA-based system, an interesting example of a single-chip parallel computer at Virginia Tech is the Single-Chip Message-Passing (SCMP) [9] computer. In this architecture, RISC[3] processors are tiled in a regular array throughout the chip and communicate with each other by sending messages to neighboring nodes. Messages are forwarded using wormhole routing until they reach their intended recipients. The advantage of this setup is that wire lengths are kept short, allowing clock frequencies to increase beyond what is possible using longer, higher resistance wires. Also, the architecture tries to exploit thread-level parallelism by giving each node its own set of 16 thread contexts that can be quickly switched.

Researchers have been using FPGAs to accelerate discretized parallel problems, although the usual approach has been to use the FPGA to solve systems of equations instead of using explicit solutions as discussed in this thesis. Frank et al. [15] suggested an FPGA implementation to provide real-time virtual reality simulation of soft tissue models. Vuilleman [37] discusses the computation of problems in heat transfer, high-energy physics, and RSA cryptography using reconfigurable systems, which combine a standard processor with an FPGA and SRAM. Ramachandran [29] implemented an FPGA-based computer similar to the system discussed in this thesis but used floating-point multiply-accumulate or MAC units to compute a matrix multiplication solution. The size and complexity of the MAC units as well as the size of the FPGAs available at the time limited the system to only a few processing elements. Paar [27] implemented a multi-FPGA cellular system for simulating heat transfer with one FPGA per cell. The cell architecture in the system was similar to the SINGLE architecture used in this research, but it used floating-point rather than fixed-point arithmetic. Schneider et al. [30] demonstrated the transfer of a computationally intensive finite-difference time-domain algorithm used in electromagnetics onto an FPGA-based computing system. Similar to the BIT architecture for this research, they used bit-serial integer arithmetic and were able to achieve a substantial speedup over single-processor computers.

[3] Reduced Instruction Set Computing

Researchers have also used FPGAs for computing cellular automata (CA). D'Antone [13] discusses the use of CA in FPGAs for random test-pattern generation, a useful component of built-in self-test for complex chip designs, and Hartka [17] used CA in FPGAs for structural analysis. Miwa et al. [26] used genetic algorithms and neural networks to model the function of the human cerebellum, and they achieved significant speedup over a conventional processor by using an FPGA implementation.

Thesis Outline

Chapter 2 describes the test problem that the FPGA system computes and explains how its features make it representative of a typical scientific problem.

Chapter 3 gives an overview of the main steps in the design flow of the architectures as well as all of the supporting design needed to make the FPGA computer an accessible system. This chapter also describes the development platform on which the system is implemented.

Chapter 4 gives a general description of the four architecture designs: MULTIPLE, SINGLE, BOOTH, and BIT. Also described is the FPGA's interface to the development board.

Chapter 5 presents the performance of each architecture as determined by testing of an actual implementation of the system. Also discussed are some of the alternative designs that were considered.

Chapter 6 suggests some possibilities for further research and ways to improve the system and extend it to other problems. It concludes with a summary of this thesis.

An Appendix is included to provide a more detailed view of the steps required for implementing the system. It also discusses some details of the VHDL descriptions.

Chapter 2
Case Study: Heat Transfer in a Matrix with Embedded Particles

An objective of this research is to design a system that computes a problem by translating it to machine hardware, so a prerequisite of the design is to select a test problem. Ideally, this problem should be representative of a typical real-life problem so that the usefulness of the system can be demonstrated. The test problem used for this design is, in fact, a real-life problem that researchers are currently studying. This chapter discusses the problem, the derivation of its model, the steps required for FPGA implementation, and some of the useful things it demonstrates.

The problem used as a case study is based on work presented by Vick and Scott in their paper [35] on heat transfer in a heterogeneous material. In this paper, they give a thermal model for a solid material with a uniform density of particles of a different material. The heat capacity of the particle material can be chosen such that the particle temperature lags the matrix material temperature (Figure 2-1). One possible use for such a material is in protective suits used in firefighting where the thermal lag effect can keep the inside cooler than the outside, which may be exposed to extreme heat.

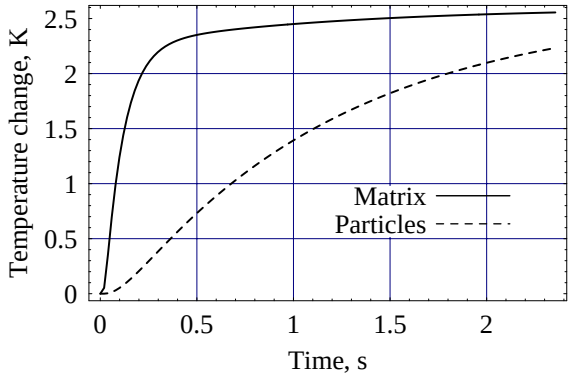

Figure 2-1. Thermal lag effect using the model in [35] with an applied temperature of +2.6 K[1]

[1] This figure was generated using the thermal model in MULTIPLE. Temperatures of cell 25 of 50 are shown for 12,000 time steps. A temperature of +2.6 is applied to both ends of the material, and $\Gamma = 0$.

A derivation of the thermal model begins by considering a volume of a bulk material containing a uniform density of particles of another material, as shown in Figure 2-2. The behavior of interest is the dynamics of T_m, the matrix material temperature, and T_p, the particle temperature. The analysis is considered one-dimensional, meaning that heat flows only in the *x* dimension. A temperature gradient, $\partial T_m / \partial x$, at the volume slice shown in Figure 2-2 will cause heat to flow at that point, and a change in the temperature gradient, $\partial^2 T_m / \partial x^2$, due to the heat flow indicates a change in the overall temperature of the slice. Thus, heat flow into the slice is given by

$$K \frac{\partial^2 T_m}{\partial x^2}, \tag{2-1}$$

where the thermal conductivity, *K*, quantifies the ability of heat to flow through the matrix material. In general, the matrix material will have a different temperature than the particles, and this difference, $T_m - T_p$, causes heat to flow from the material into the particles. The ability of heat to cross the interface between the matrix and the particles is characterized by a heat transfer coefficient, *H*, and this heat flow is given by

$$H\left(T_m - T_p\right). \tag{2-2}$$

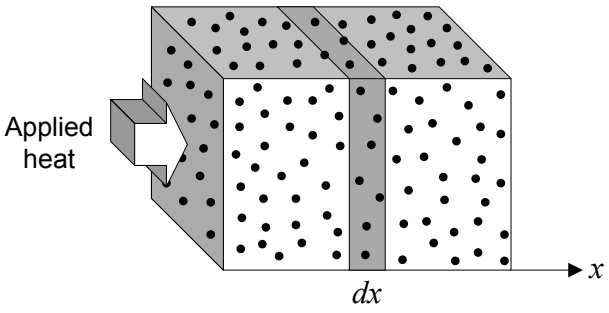

Figure 2-2. Idealized sectional used for analysis

The thermal model in [35] has been extended for this analysis to include the effects of curing, a heat-induced chemical change that alters the material's thermal properties. A model [8] of the heat generated by this curing process is given by

$$\Gamma \frac{\partial \alpha}{\partial t}, \tag{2-3}$$

where

$$\frac{\partial \alpha}{\partial t} = (K_1 + K_2 \alpha)(1 - \alpha) \text{ and} \tag{2-4}$$

$$K_i = A_i e^{-E_i/(\bar{R}T_m)}, \ i = 1, 2. \tag{2-5}$$

As an example of the effect of curing on the temperature dynamics of a material, Figure 2-3 uses the same setup as Figure 2-1 but includes curing. It can be seen that curing causes a heat spike to occur once the matrix material has reached a certain temperature. The spike peaks and then quickly returns to a stable temperature.

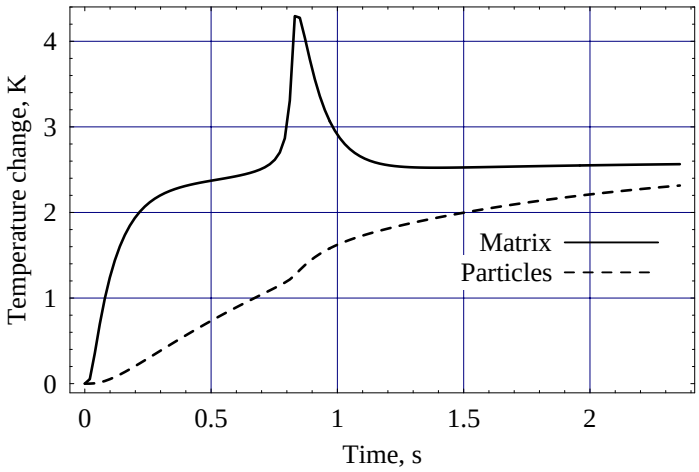

Figure 2-3. Same setup as Figure 2-1 but with the addition of the curing effect[2]

All the heat sources and sinks with respect to the matrix material add together to give

$$C_m \frac{\partial T_m}{\partial t}, \tag{2-6}$$

the rate of heat storage in that part of the material, where the heat capacity, C_m, is the ability of the matrix material to store heat energy. Invoking the law of conservation of energy, expressions (2-1), (2-2), (2-3), and (2-6) can be combined to give

[2] This figure has $\Gamma = 2.6$.

$$C_m \frac{\partial T_m}{\partial t} = K \frac{\partial^2 T_m}{\partial x^2} - H\left(T_m - T_p\right) + \Gamma \frac{\partial \alpha}{\partial t}. \tag{2-7}$$

Equation (2-7) is the basic model for determining how the matrix temperature changes over time. Heat is added to the matrix material through an applied temperature and through the curing process, and heat is removed from the material by flowing into the particles. The particles are considered small enough that heat flow within them can be ignored, and it is assumed that the curing effect does not occur for the particle material. Thus, the particle temperature changes only due to the heat flowing through the matrix/particle interface, suggesting that the particle temperature dynamics can be characterized by

$$C_p \frac{\partial T_p}{\partial t} = H\left(T_m - T_p\right), \tag{2-8}$$

where C_p is the heat capacity of the particle material.

Explicit Model

Equation (2-7) cannot be solved analytically, but a numerical solution can be obtained by using the finite difference method described by Patankar [28]. The approach is to divide the material into *N* discrete cells and to divide the duration under analysis into *T* discrete time elements, as depicted in Figure 2-4. The differential elements of time and space in the equations can be approximated by Δt and Δx, respectively, allowing forms like $\partial T_m / \partial t$ to be written as

$$\frac{T_{m,j}^{i} - T_{m,j}^{i-1}}{\Delta t},$$

where the *j* subscript specifies the cell and the *i* superscript specifies the time step.

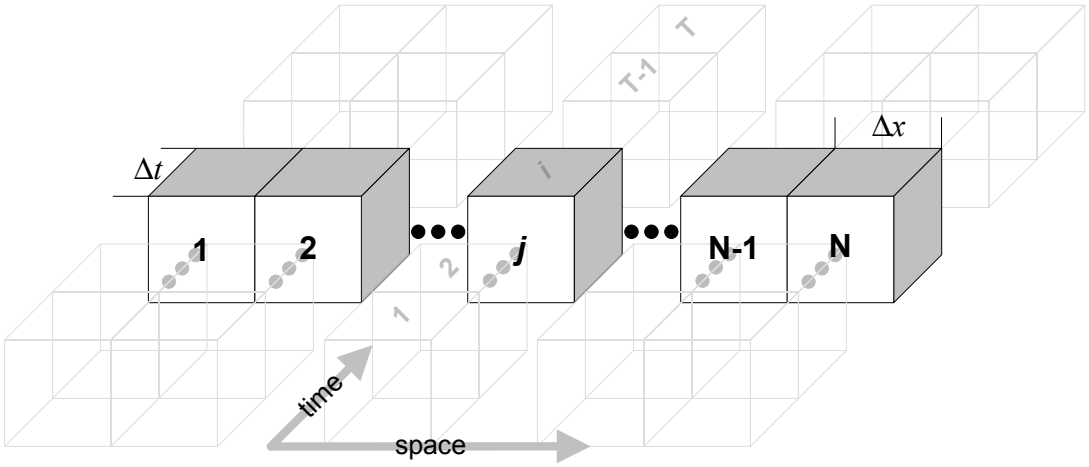

Figure 2-4. Discretization of space and time for numerical analysis

In discrete form, the second derivative in (2-1) represents a difference of differences of adjoining cell temperatures where each cell, *j*, considers the temperatures of its left and right neighbors. There is a question, however, as to the time step at which these differences are taken. An implicit method uses the current time step, *i*, and an explicit method uses the previous time step, $i-1$. Equations (2-4) and (2-8) are the same when rewritten using either method and are given by

$$\frac{\alpha_j^i - \alpha_j^{i-1}}{\Delta t} = \left(K_{1,j}^{i-1} + K_{2,j}^{i-1}\alpha_j^{i-1}\right)\left(1-\alpha_j^{i-1}\right) \text{ and}$$

$$C_p \frac{T_{p,j}^i - T_{p,j}^{i-1}}{\Delta t} = H\left(T_{m,j}^{i-1} - T_{p,j}^{i-1}\right),$$

which can be rearranged to give

$$\alpha_j^i = \Delta t\left(K_{1,j}^{i-1} + K_{2,j}^{i-1}\alpha_j^{i-1}\right)\left(1-\alpha_j^{i-1}\right) + \alpha_j^{i-1} \text{ and} \tag{2-9}$$

$$T_{p,j}^i = T_{p,j}^{i-1}\left(1 - \frac{H\Delta t}{C_p}\right) + \frac{H\Delta t}{C_p} T_{m,j}^{i-1}. \tag{2-10}$$

Rewriting equation (2-7) using the implicit method gives

$$C_m \frac{T_{m,j}^i - T_{m,j}^{i-1}}{\Delta t} = \frac{K}{\Delta x}\left(\frac{T_{m,j+1}^i - T_{m,j}^i}{\Delta x} - \frac{T_{m,j}^i - T_{m,j-1}^i}{\Delta x}\right) - H\left(T_{m,j}^{i-1} - T_{p,j}^{i-1}\right) + \Gamma\frac{\alpha_j^i - \alpha_j^{i-1}}{\Delta t},$$

which can be rearranged to give

$$aT_{m,j}^{i} + b\left(T_{m,j-1}^{i} + T_{m,j+1}^{i}\right) - \Gamma\alpha_{j}^{i} = c_{j}^{i}, \tag{2-11}$$

where

$$a = C_m - 2b,$$

$$b = -\frac{K\Delta t}{\Delta x^2}, \text{ and}$$

$$c_{j}^{i} = C_m T_{m,j}^{i-1} - H\Delta t\left(T_{m,j}^{i-1} - T_{p,j}^{i-1}\right) - \Gamma\alpha_{j}^{i-1}.$$

Equation (2-11) sets up a system of j simultaneous equations that does not render itself to FPGA-based cellular computing because its solution requires expensive matrix multiplication, and the complexity of the computation increases with the number of cells. Rewriting equation (2-7) using the explicit method gives

$$C_m \frac{T_{m,j}^{i} - T_{m,j}^{i-1}}{\Delta t} = \frac{K}{\Delta x}\left(\frac{T_{m,j+1}^{i-1} - T_{m,j}^{i-1}}{\Delta x} - \frac{T_{m,j}^{i-1} - T_{m,j-1}^{i-1}}{\Delta x}\right) - H\left(T_{m,j}^{i-1} - T_{p,j}^{i-1}\right) + \Gamma\frac{\alpha_{j}^{i} - \alpha_{j}^{i-1}}{\Delta t},$$

which can be rearranged to give

$$T_{m,j}^{i} = a\,T_{m,j}^{i-1} + b\left(T_{m,j-1}^{i-1} + T_{m,j+1}^{i-1}\right) + c_{j}^{i}, \tag{2-12}$$

where

$$a = -2b,$$

$$b = \frac{K\Delta t}{C_m \Delta x^2}, \text{ and}$$

$$c_{j}^{i} = T_{m,j}^{i-1} - \frac{H\Delta t}{C_m}\left(T_{m,j}^{i-1} - T_{p,j}^{i-1}\right) + \frac{\Gamma}{C_m}\left(\alpha_{j}^{i} - \alpha_{j}^{i-1}\right).$$

Equation (2-12) gives an explicit solution for the matrix material temperature that can be calculated from previously computed values, and the entire set of cells can be updated in parallel. This makes the explicit solution ideal for an FPGA-based cellular computing system and is used in the design discussed in this thesis.

One caveat with the explicit solution that does not exist for the implicit case is a restriction on the size of Δt. The temperature solutions given in equations (2-10) and (2-12) will remain numerically stable only if the coefficients of $T_{m,j}^{i-1}$ and $T_{p,j}^{i-1}$ are greater than or equal to zero, or

$$\Delta t \le \min\left(\frac{C_m \Delta x^2}{2K + \Delta x^2 H}, \frac{C_p}{H}\right).$$

The analysis is simplified for this research by assuming that $K_1 = K_2$, $K = H = C_m = C_p = 1$, and $\Delta x = 1/N$. With these assumptions, equations (2-9), (2-10), and (2-12) become

$$\alpha_j^i = \Delta t \cdot K_j^{i-1}\left(1+\alpha_j^{i-1}\right)\left(1-\alpha_j^{i-1}\right)+\alpha_j^{i-1}, \tag{2-13}$$

$$T_{p,j}^i = T_{p,j}^{i-1}\left(1-\Delta t\right)+\Delta t \cdot T_{m,j}^{i-1}, \text{ and} \tag{2-14}$$

$$T_{m,j}^i = aT_{m,j}^{i-1} + b\left(T_{m,j-1}^{i-1} + T_{m,j+1}^{i-1}\right)+c_j^i, \tag{2-15}$$

where

$$b = \frac{\Delta t}{\Delta x^2},$$

$$c_j^i = T_{m,j}^{i-1} - \Delta t\left(T_{m,j}^{i-1} - T_{p,j}^{i-1}\right)+\Gamma\left(\alpha_j^i - \alpha_j^{i-1}\right),$$

and a is the same as for the explicit case above. The time step constraint becomes

$$\Delta t \le \frac{1}{2/\Delta x^2 + 1},$$

and for this analysis,

$$\Delta t = \frac{0.99}{2/\Delta x^2 + 1}.$$

Arrhenius Approximation

The Arrhenius equation (2-5) involves an exponential function, which must be approximated in some way so that it can be computed using the basic arithmetic

operations available in an FPGA. A common approach to approximating a complicated function is to use some number of terms from the beginning of its Taylor series expansion, which works well for exponentials in the form of e^{-x}. The Arrhenius form, however, involves an exponential in the form of $e^{-1/x}$ with its independent variable in the denominator of the exponent, and an accurate Taylor series approximation can require a large number of terms, which in turn requires a large number of costly multiplications.

A better approach is a parabolic approximation, which can be used in this case because the matrix temperature is not expected to span a large range of values. Equation (2-5) with values used in this analysis is $K_{1,2} = 10{,}000\, e^{-20/T_m}$ and is shown in Figure 2-5 over the range of expected temperatures. It can be seen that over this range, the function is close to zero for temperatures less than about two, so for temperatures below this value, the function is taken to be zero, and for higher temperatures, the function is approximated with a parabola. Determined by using a least-squares fit[3], this parabolic approximation is given by

$$K_{1,2,\,approx} = \begin{cases} 25.99\left(T_{m,j}^{i-1} - 2.37\right)^2, & T_{m,j}^{i-1} \geq 2.37 \\ 0, & T_{m,j}^{i-1} < 2.37 \end{cases} \tag{2-16}$$

and is shown along with the original function in Figure 2-5. The parabolic approximation parameters will be referred to as

$$C_a = 25.99 \text{ and}$$

$$C_b = 2.37\,.$$

Each of the four cell architectures generates new temperatures by computing equations (2-13), (2-14), (2-15), and (2-16). The operations required for this calculation are depicted in the dataflow diagram shown in Figure 2-6, and the only variables in the computation are α, T_m, and T_p.

[3] Least-squares fit to 2632 data points between 2.37 and 5

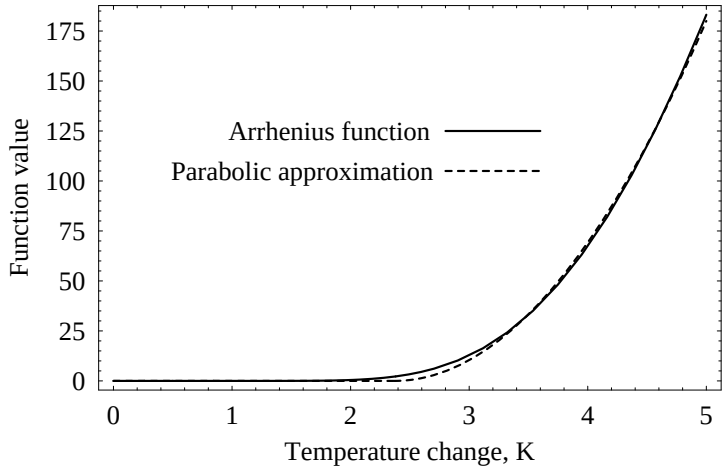

Figure 2-5. Parabolic approximation of an exponential function

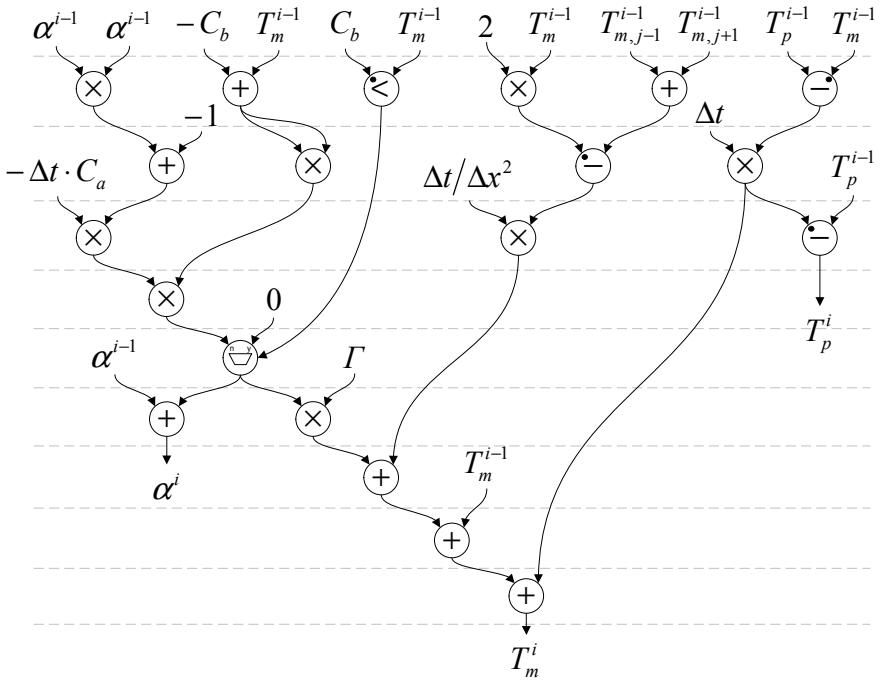

Figure 2-6. Dataflow diagram showing the heat flow computation

What this Test Case Demonstrates

The heat flow test case is useful not only because it demonstrates the application of the FPGA system to a real-life problem, but for other reasons as well. The variable coefficients like those in (2-12) are sometimes constant values that will not change, but

they are more often parameters that a researcher would like to adjust in order to see their effect on the model's behavior. In order to demonstrate the ability to quickly change a parameter without requiring an FPGA reconfiguration, the coefficient of the curing term, Γ, in (2-3) is set up to be sent as an input parameter to the FPGA. Additionally, the externally applied temperatures are input parameters that can be changed without reconfiguration.

Many problems require more complex operations than additions and multiplications, and an example of this in the heat flow test case is the less-than test required for the parabolic approximation in (2-16). To address this need, additional hardware is included in the design to realize a less-than test and its inclusion is a useful example of the implementation of a complex operation.

Chapter 3
Design Flow and Development Tools

The design work required for this research includes describing the four architectures in a hardware description language, transforming the description into an FPGA-suitable form, defining interfaces between the FPGA and the development platform, and, finally, defining an interface between the development platform and a host PC where a user interacts with the system. This chapter gives an overview of the main development steps performed and the tools used throughout the design process.

The development process can be divided into two major steps: the FPGA design and the supporting design. These steps are shown in the flow diagram of Figure 3-1 along with the major sub-steps.

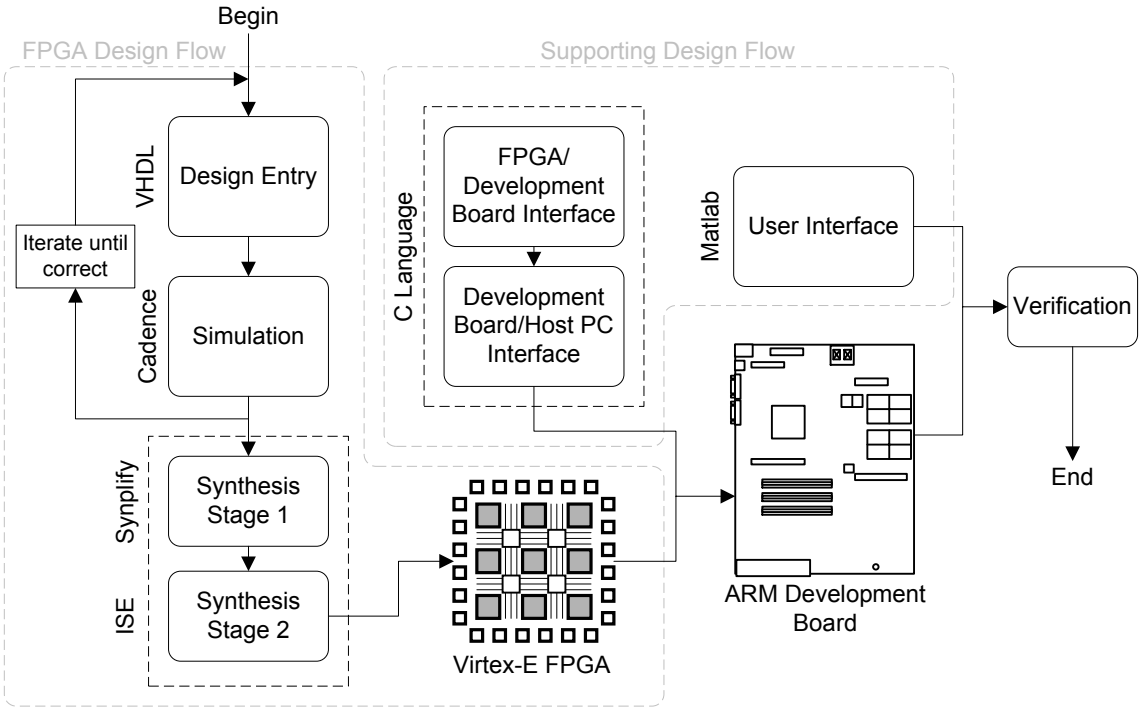

Figure 3-1. Flow diagram of the major design development steps

FPGA Design Flow

This section discusses the design flow required to realize the initial concept in an FPGA, and it gives an overview of the particular FPGA used in this research.

Design entry was done using VHDL [19], a textual language for describing digital hardware. Although a VHDL description is mainly used for simulation, it can also be used for synthesis, but a subset of the language [20] must be used if a design is meant to be synthesized. Synthesis is the process of taking an input form like VHDL to a device-specific form such as a programming file.

A design must be verified for correctness before it is synthesized because the synthesis process can take a very long time. Verification of this design was done by simulating waveforms using Cadence tools, which was preceded by the construction of a VHDL testbench. VHDL simulation requires three steps: analysis, elaboration, and simulation. Analysis, done using `ncvhdl`, parses the text input and places design units in libraries. Elaboration, done using `ncelab`, creates a set of processes that can be executed to simulate the hardware. Simulation, done using `ncsim`, executes the processes and allows signals to be viewed as a waveform. These tools were used from inside the NCDesktop v03.20 IDE.

A design is ready for synthesis once it simulates correctly. All synthesis steps can be done using software from Xilinx, the manufacturer of the FPGA used in this design, but the initial compilation and mapping steps were done using Synplify Pro 7.1 because it is capable of giving better results for large designs [12]. During this initial synthesis step, the VHDL description is interpreted and logic is assembled and mapped to FPGA components.

With the required components defined, they can be assigned to actual chip locations and connections can be made between them. Because of the interconnection architecture in the device used for this research, these place and route steps can take several hours to complete as the software tries to find routes that meet timing constraints. Once the design is placed and routed, a binary file can be generated that contains all the information needed for programming the FPGA with the design. For this research, placement, routing, and programming-file-generation were all done using ISE 5.1i.

There are generally two ways to program an FPGA with a binary file: directly or indirectly through flash memory. For the Xilinx FPGA, either the iMPACT software along with the Parallel Cable IV from Xilinx or Multi-ICE can be used to program the chip directly through the Multi-ICE connector on the ARM development board, but this method is unfavorable because the design remains programmed only while the FPGA is powered. A better method is to store the binary programming file in a flash memory that is set up to download the programming information into the FPGA on power-up. Using Multi-ICE 2.2 and the `progcards` utility from ARM, the flash memory programming method was used for this research.

The particular FPGA used for this research was a Xilinx Virtex-E XCV2000E. This chip is made up of an array of 80×120 CLBs surrounded by I/O blocks and interconnected with various routing resources similar to the structure shown in Figure 1-1, and configuration is achieved by loading static RAM cells. Each CLB (Figure 3-2) has 2 slices, each of which has two logic cells (LCs). An LC contains a 4-input LUT-based function generator followed by a flip-flop (FF). Additional logic is included in the CLB to allow function generators to be combined to produce higher-input functions. A LUT can be used as a 16×1-bit synchronous RAM and can be combined with other LUTs to produce larger memories, or it can be used as a shift register.

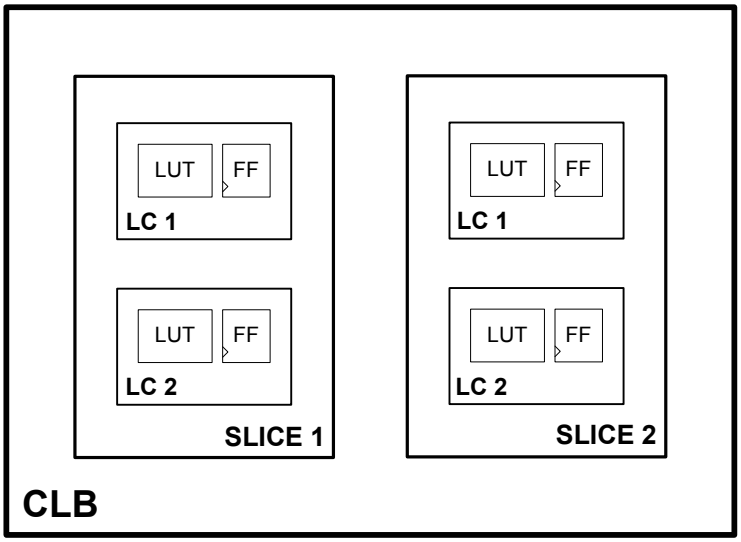

Figure 3-2. Structure of a Virtex-E CLB

Horizontal and vertical routing channels run between the rows and columns of CLBs, and at the intersection of these channels are routing switches. These switches allow CLBs to interconnect by interfacing with the CLBs as shown in Figure 3-3. CLBs are also able to form direct connections to their left and right neighbors so that delays incurred by going through the programmable switches can be avoided. Global clocks must avoid switch delays, so they are distributed throughout the chip using special resources that minimize clock skew.

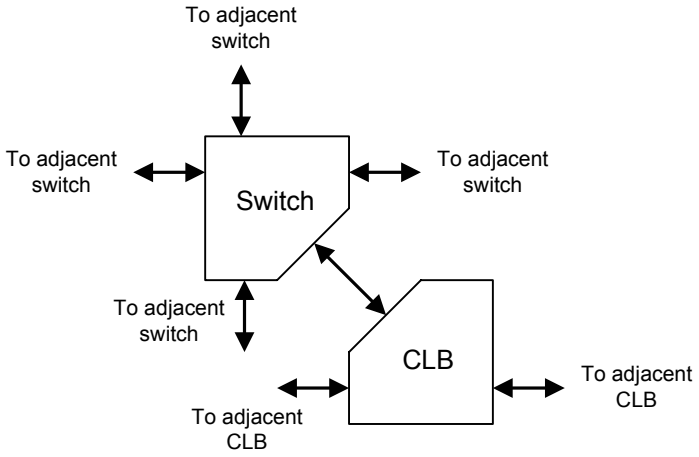

Figure 3-3. CLB interface with the routing matrix[1]

The Virtex-E XCV2000E FPGA also has extra features to support designs that use the chip. In addition to the distributed RAM contained in LUTs, the chip has 160 512-byte block RAMs spread across the chip for a total of 80KB of extra memory. The chip has 8 digital delay-locked loops that allow clocks to be multiplied or divided, or they can be used to eliminate skew on the clock lines. The horizontal routing channels contain dedicated lines that allow for tri-state busses. There is also additional arithmetic logic in the CLBs that helps to speed up add and multiply operations. The synthesis software uses many of these resources automatically whenever a design can benefit from them.

[1] Adapted from a figure in [44]

Supporting Design and Development Platform

This section introduces the development platform on which the design was implemented and discusses the design flow of those parts of the design not specifically included in the FPGA design flow.

For the FPGA-based computing system to be useful, there should be a convenient way to communicate with the chip and retrieve the results it generates. An ideal arrangement would be the FPGA attached to a PC where computing software such as Matlab or Mathematica could accelerate the computation of certain problems by sending them to the FPGA. The ARM Integrator/AP platform provides a good environment for developing a PC/FPGA setup. As shown in Figure 3-4, the Integrator/AP is a motherboard that provides basic system resources such as memory and I/O to modules that can be attached to the system. Up to five core or logic modules can be added to the system by stacking them on one another, but only one of each module was used for this research. Core modules contain a processor and logic modules contain a configurable FPGA, and all modules can communicate over a common bus. The motherboard has a simple boot monitor that can configure the system and execute images stored in on-board flash memory, but switch settings allow flash images containing user programs to execute on power-up. I/O resources include serial ports, PCI slots, and keyboard and mouse connectors.

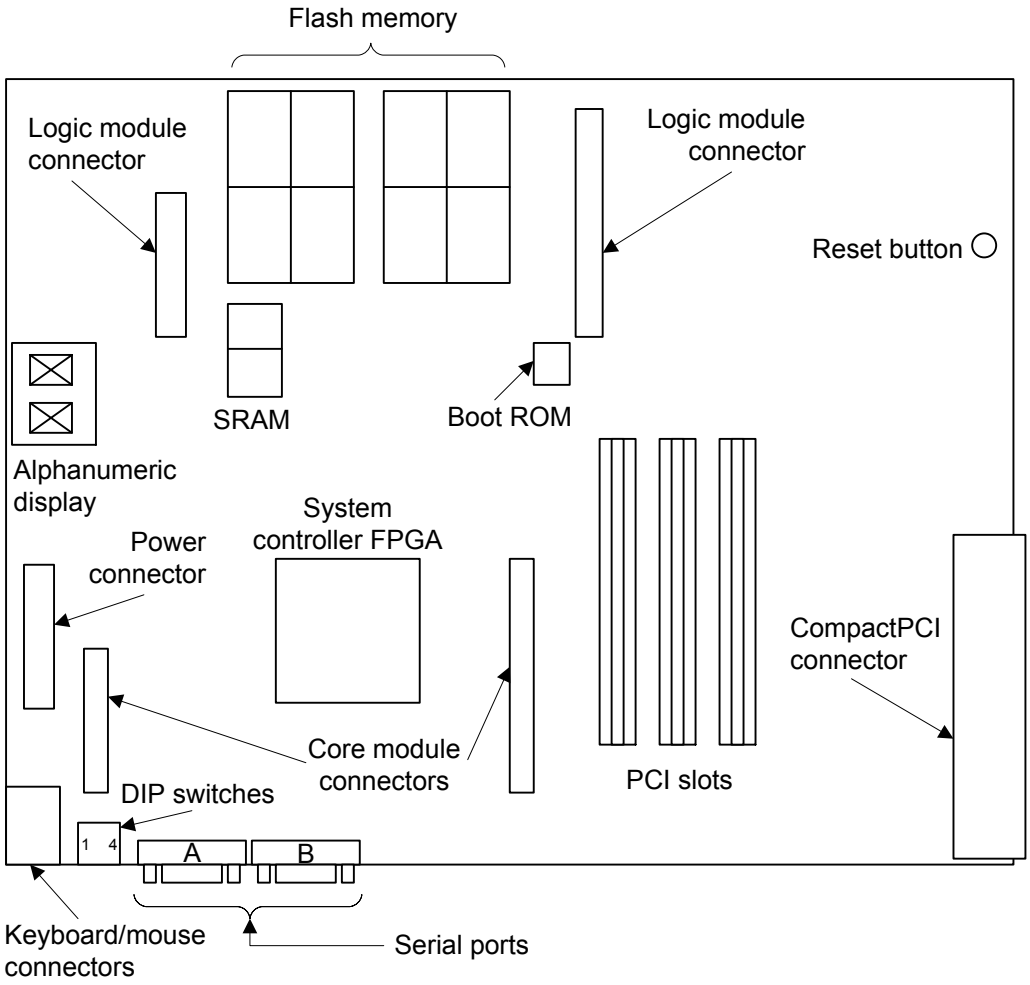

Figure 3-4. ARM Integrator/AP motherboard layout[2]

The core module used for this design, CM720T, has an ARM720T chip, which is a 32-bit RISC processor that executes the ARM and Thumb instruction sets. The module has a DIMM slot for up to 256MB of SDRAM that can be accessed by other modules on the motherboard, and it has 256KB of local SSRAM. The logic module used for this design, LM-XCV600E+ (Figure 3-5), has a Xilinx XCV2000E FPGA, 4MB of local flash memory used for storing FPGA configurations, and 1MB of SSRAM for general-purpose memory. The module also has general-purpose LEDs and input switches as well as switches for selecting the configuration data to be loaded into the FPGA on power-up.

[2] Adapted from a figure in [6]

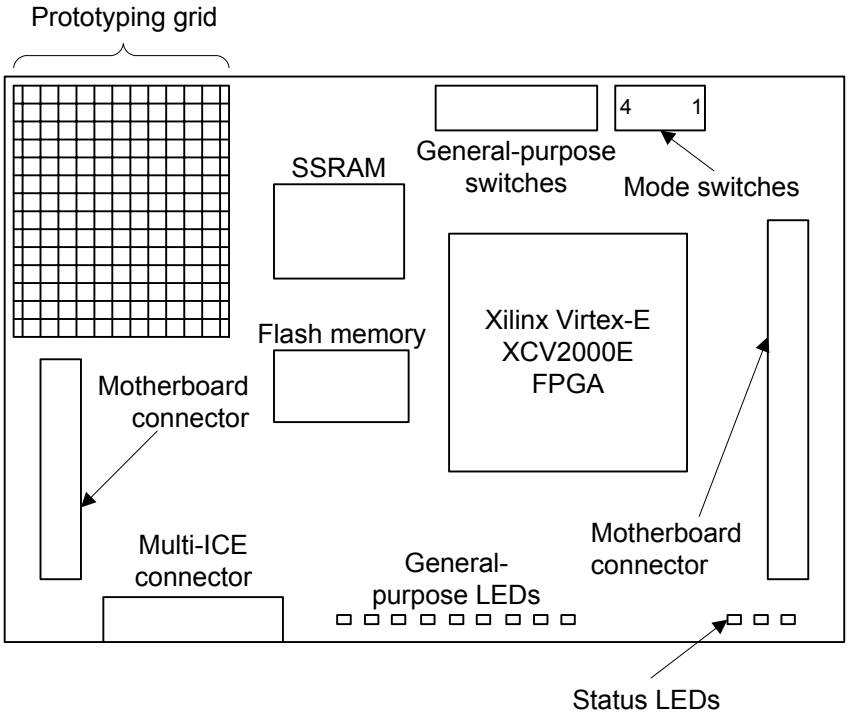

Figure 3-5. ARM LM-XCV600E+ logic module layout[3]

The Integrator/AP motherboard is not set up to support full-scale operating systems that can run calculation software like Matlab or Mathematica, so to gain this functionality, the motherboard is connected to a host PC where FPGA-computed results can be manipulated. This presents the problem of how to quickly transfer FPGA-computed results that are stored on the logic module to the PC's calculation software, since a goal of this research is to demonstrate that the FPGA system can give results more quickly than a typical PC system. While not an ideal solution to the data transfer problem, an adequate solution for this research has been to use both Integrator/AP serial links at full speed. These serial links are also used for problem setup, which includes sending input parameters to the FPGA.

A core module attached to the Integrator/AP motherboard acts as an ideal intermediary between the host PC and the FPGA since it frees the FPGA from requiring serial communication hardware and leaves more space for the computing cells. Thus, the host PC's link to the FPGA system is set up as shown in Figure 3-6. The interface program that runs on the core module's processor was compiled from C code using

[3] Adapted from a figure in [7]

CodeWarrior for the ARM Developer Suite (ADS) v1.2 in conjunction with libraries from the ARM Firmware Suite (AFS) v1.4.1. The AXD Debugger was used to download the program's binary into the motherboard's flash memory.

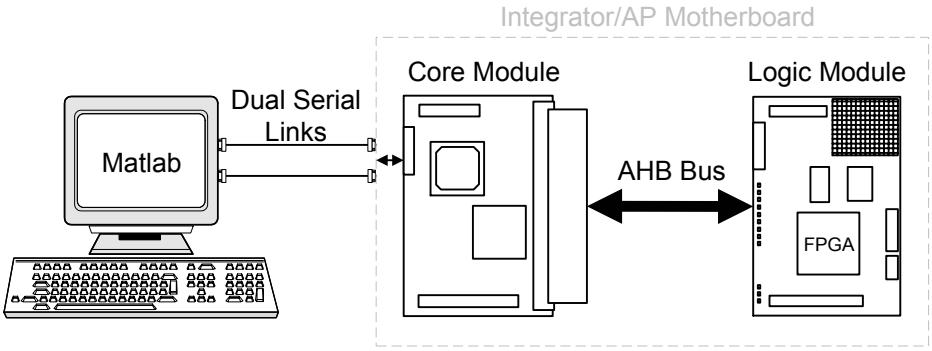

Figure 3-6. Host PC's link to the FPGA system

As a demonstration of the FPGA system's use from common calculation software, a Matlab interface was written to allow a user to provide input parameters to and receive results from the FPGA. A similar interface could be written for Mathematica or another calculation system.

Chapter 4
Four Architecture Designs

The main purpose of this chapter is to describe the four computing-cell architectures implemented for this research, but this chapter also describes the common top-level FPGA hardware used by each architecture to facilitate interaction with the surrounding system. The four architectures MULTIPLE, SINGLE, BOOTH, and BIT differ in the way they compute the test problem by using different degrees of parallelism and by performing arithmetic operations differently. The hardware is designed specifically for the test problem, so each architecture includes only the minimum number of arithmetic units required to compute the problem. Therefore, each arithmetic unit[1] is used in every clock cycle making the duty cycle of each architecture 100%. The logic required for system interaction is kept to a minimum in an effort to reserve as much chip area as possible for the computing cells. Except for the logic that controls the FPGA's clock frequency, the system interface logic is identical for all four architectures.

Common Top-Level Design

This section describes the tasks performed by the FPGA and the hardware implemented to carry out these tasks. Only the parts of the FPGA design common to each architecture are discussed in this section.

The steps performed by the FPGA during a complete calculation cycle are shown in Figure 4-1. Immediately after power-up, configuration data from the Logic Module (LM) flash memory is downloaded to the FPGA. At the same time, the Core Module (CM) initializes the Integrator/AP motherboard and then waits for input parameters to be sent from the host PC. After the CM receives these parameters, it sends them over the motherboard's Advanced High-performance Bus (AHB) to the LM. At this time, writes

[1] Adders and multipliers

to the LM address space are forwarded to the LM SSRAM by the FPGA. Before the FPGA begins computing, it expects all input parameters to reside at the beginning of the LM SSRAM in the following order: *iterations*, *result cell*, and the rest of the input parameters. *Iterations* is the number of iterations to calculate, which are time steps for the heat flow test case. If the FPGA is configured with *N* computing cells, then *result cell* is a number between *1* and *N* and is the cell number whose data will be stored in SSRAM. For the heat flow problem, the cell data stored is the matrix temperature at each time step. The remaining input parameters are problem-specific, and for the heat flow problem, they are the *temperature applied to Cell 1*, the *temperature applied to Cell N*, and the curing parameter Γ. Because of the number scheme used in this design, these input parameters must be given as fixed-point scaled integers. After the CM sends these parameters, it sends the FPGA a *start* signal indicating that the calculation is ready to proceed. After the FPGA receives this signal, it gains control of the LM SSRAM and reads the input parameters described above. With the input parameters in place, the row of computing cells is instructed to begin calculating, and when an iteration has completed, the result in *result cell* is written to SSRAM starting immediately after the input parameters at the beginning of the memory space. Iteration results continue to be written to SSRAM until *iteration* results have been stored. At this point, the FPGA releases control of the SSRAM and allows the CM to read the computed results over the AHB bus. Initial values are then reset, and the FPGA is ready to begin a new calculation cycle. If a second calculation cycle is started, the FPGA overwrites the stored results from the first cycle.

The VHDL that implements the procedure described above is set up to be extensible to any cellular parallel problem that is similar to the heat flow test case. There can be any number of input parameters following *iterations* and *result cell*, and the number of cells is limited only by the available space in the FPGA. Changes to the number of input parameters or the number of cells requires resynthesis of the design and reconfiguration of the FPGA, but the number of iterations calculated is limited only by the size of the SSRAM where results are stored.

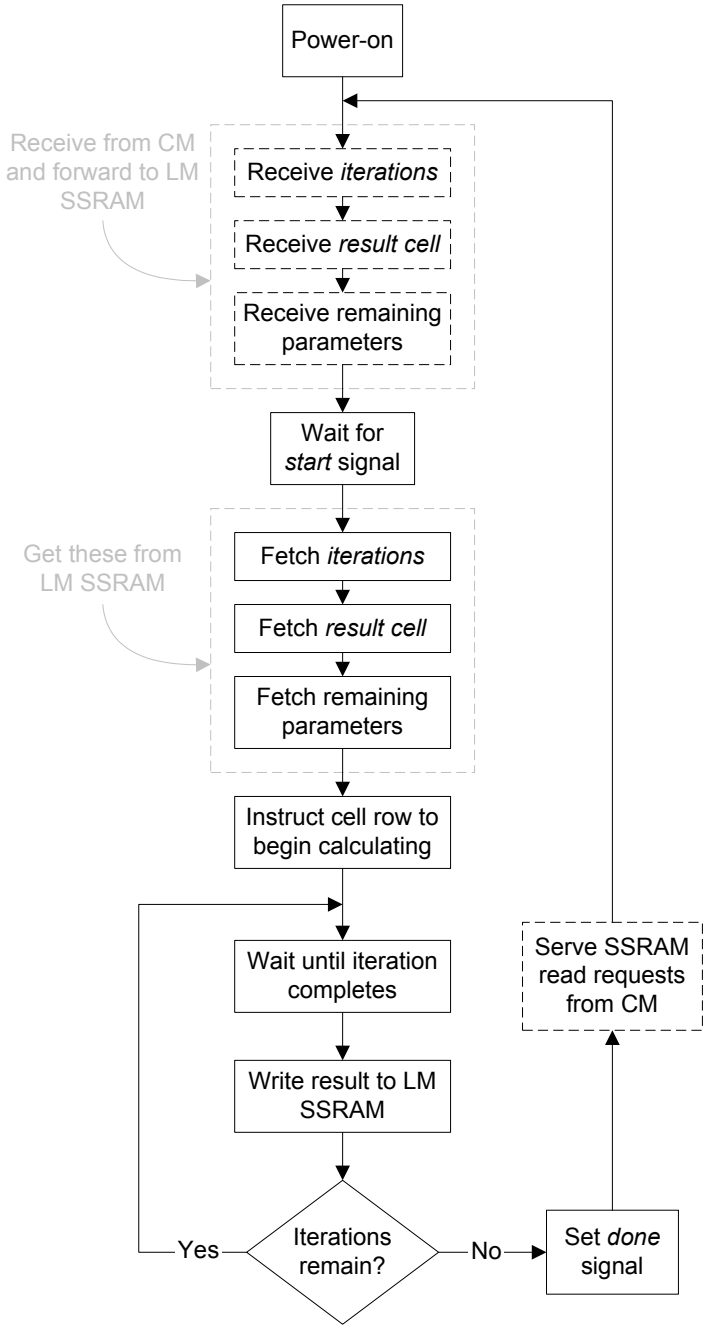

Figure 4-1. Tasks performed by the FPGA design; dashed tasks are not required for a calculation cycle

Figure 4-2 shows the structure of the top-level FPGA design that implements the procedure in Figure 4-1. The cell row controller is a Finite State Machine (FSM) that waits for a *start* signal, reads SSRAM input parameters, starts the row computation and sends results to SSRAM, and keeps track of the number of iterations that have completed. A single cell-controller FSM controls all cells in the row and is different for each

architecture. The SSRAM controller provides the memory chip's control signals and facilitates the timing of reads and writes. Control and data lines from both the AHB bus and the row controller are multiplexed onto the SSRAM, and data from the row controller and the SSRAM are multiplexed onto the AHB bus. Although not shown in Figure 4-2, there is also logic that decodes addresses from the AHB bus. All addresses in the LM address space except one will address the SSRAM. A write from the CM to the single address `0x0DDC0DE` in the LM's address space writes the FPGA's *start* signal, and a read of the same address reads the row controller's *done* signal, which indicates when the CM can begin reading the calculated results from SSRAM. Also not shown in the figure is logic that controls the FPGA's programmable clock and the LM's indicator LEDs.

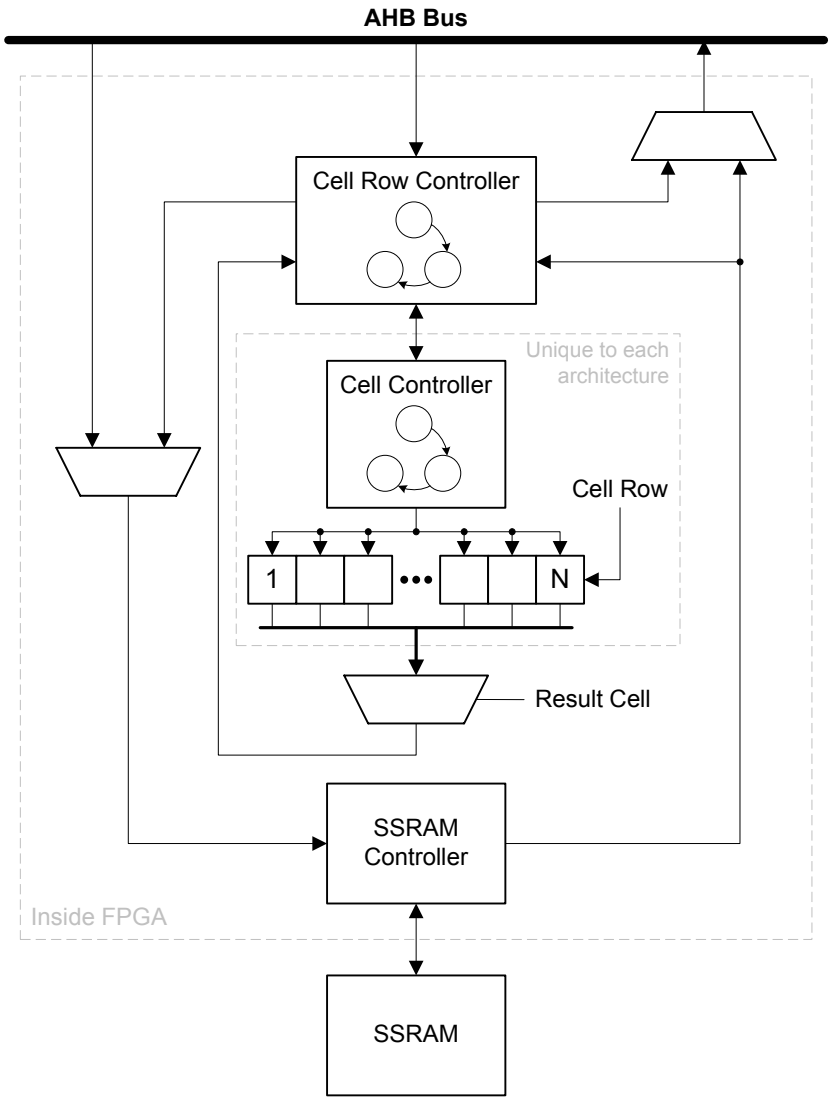

Figure 4-2. Structure of the top-level FPGA design

The computing cells in the FPGA run on a separate clock from the AHB bus clock so that they are not constrained to the bus frequency. Data must be transferred between these two asynchronous clock domains, however, so a handshaking sequence is performed to ensure proper data delivery. The handshaking is set up to allow any combination of bus and cell clock frequencies.

A detailed view of the cell row structure is shown in Figure 4-3. The explicit solution given in equation (2-15) dictates this structure, and it requires that each cell connect to its left and right neighbors and that it have access to the values computed in its previous iteration. Each cell computes equations (2-13), (2-14), (2-15), and (2-16) and stores the matrix and particle temperatures for a single time step. Collectively, the values stored represent the temperature distribution across the section of material partitioned by the cells. All cells are updated in parallel, and the speed of the computation does not depend on the number of cells, *N*. An effort is made to make the computing cells as small as possible by moving most of the control outside of the cell despite the fact that a centralized controller is contrary to the cellular computing model described in [31]. These cells are replicated until the entire FPGA area is utilized.

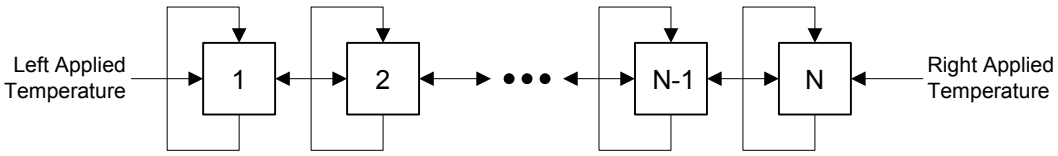

Figure 4-3. Cell row structure

MULTIPLE

This section discusses the cell controller and computing cell structures of the MULTIPLE architecture.

The MULTIPLE architecture is given this name because each computing cell instantiates multiple adders and multipliers: one for each add and multiply that occurs in the underlying equations being computed. The computation is done completely with combinational logic, so new values could be generated every clock cycle. However, due to the handshaking needed to write results to SSRAM, each iteration takes two clock

cycles. Thus, the cell controller is an extremely simple FSM that toggles between two states. During one state, a value is computed, and during the other state, the value is stored in memory. The MULTIPLE architecture consumes a large amount of chip area, so its usefulness is mainly as a basis of comparison for the other architectures.

VHDL supports standard arithmetic operations, so the equations that need to be computed can almost be written directly in the hardware description. However, it is desirable to first put the equations into forms that minimize the number of multiplies, since multiply hardware consumes more area than addition. Thus, the heat flow equations for the MULTIPLE architecture are implemented as shown in Figure 4-4. The temporary variables t_1, t_2, and $\Delta\alpha$ are used to ensure that the synthesis software does not instantiate unnecessary logic. To implement the less-than test, the MULTIPLE architecture instantiates full-width comparator hardware.

$$t_1 = T_{m,j}^{i-1} + \left(-C_b\right)$$

$$\Delta\alpha = \begin{cases} 0, & T_{m,j}^{i-1} < -\left(-C_b\right) \\ t_1 \times t_1 \times \left(-\Delta t \cdot C_a\right) \times \left(\alpha_j^{i-1} \times \alpha_j^{i-1} + (-1)\right), & T_{m,j}^{i-1} \geq -\left(-C_b\right) \end{cases}$$

$$t_2 = \Delta t \times \left(T_{p,j}^{i-1} - T_{m,j}^{i-1}\right)$$

$$\alpha_j^i = \alpha_j^{i-1} + \Delta\alpha$$

$$T_{m,j}^i = T_{m,j}^{i-1} + t_2 + \frac{\Delta t}{\Delta x^2} \times \left(T_{m,j-1}^{i-1} + T_{m,j+1}^{i-1} - 2T_{m,j}^{i-1}\right) + \Gamma \times \Delta\alpha$$

$$T_{p,j}^i = T_{p,j}^{i-1} - t_2$$

Figure 4-4. Heat flow equations reformulated for the MULTIPLE architecture

All variables and parameters in Figure 4-4 are fixed-point scaled *W*-bit integers, so results of multiply operations must be scaled accordingly. First, a standard VHDL multiply is performed, which produces a 2*W*-bit result. The scaled integer result of the multiply operation resides within this 2*W*-bit result as shown in Figure 4-5 assuming *S*

integer bits. For this and the other architectures, $W = 25$ and $S = 5$. W was chosen by overlaying temperature plots made using floating-point and fixed-point calculations, and W was increased until the fixed-point plot "looked" accurate.

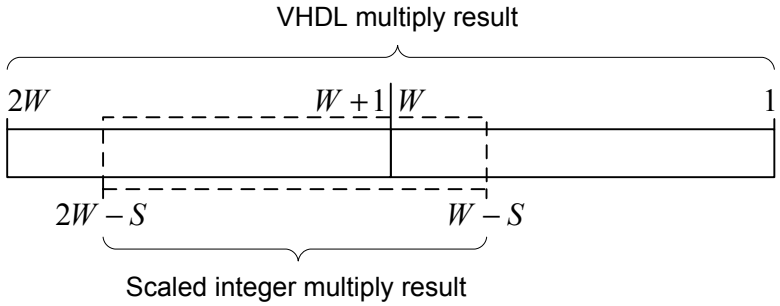

Figure 4-5. Scaled integer multiplies

The cell structure for the MULTIPLE architecture is shown in Figure 4-6. The number of add and multiply units in the figure shows why each cell consumes so much chip area. Each multiply unit performs the operation shown in Figure 4-5. The *enable* signal allows new results to latch into the registers and is generated by the 2-state FSM in the cell controller.

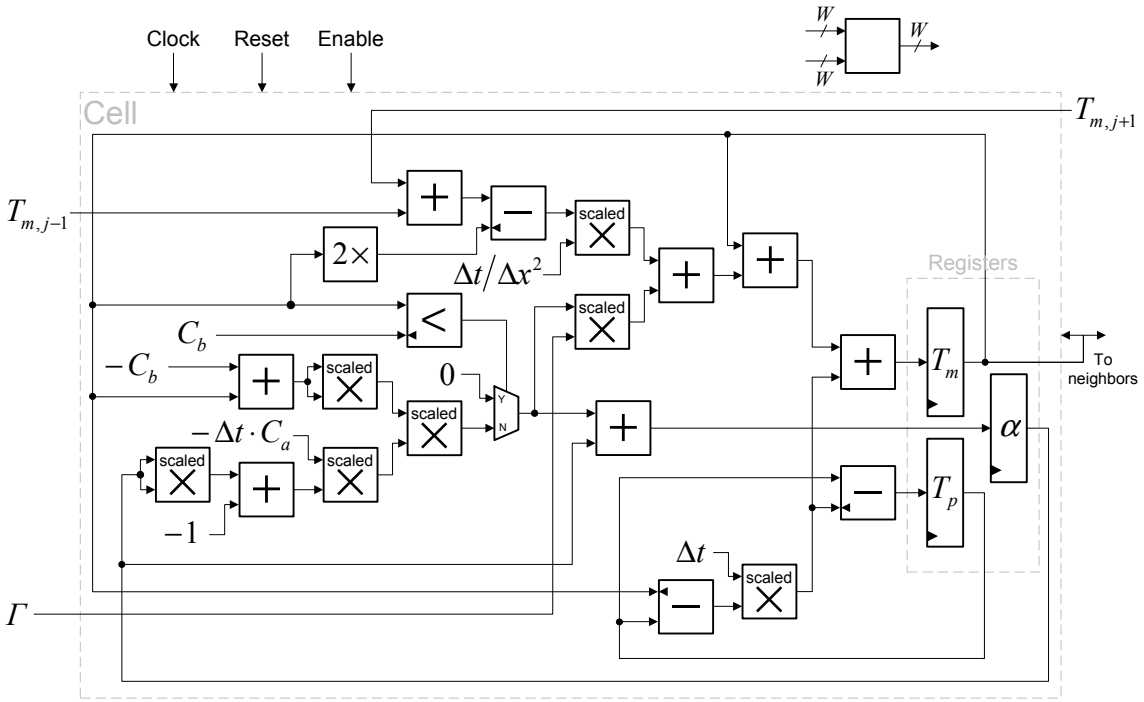

Figure 4-6. Cell structure for the MULTIPLE architecture

SINGLE

This section discusses the cell controller and computing cell structures of the SINGLE architecture.

The SINGLE architecture is given this name because each cell time-shares a single multiplier and adder among a sequence of operations that implement the heat flow equations. Thus, it uses less area than MULTIPLE but more clock cycles, although the clock rate may be able to be higher for SINGLE because signals only have one multiplier level to travel through instead of the multiple levels in MULTIPLE. Each single-cycle operation consists of a consecutive multiply and add. Time-sharing the arithmetic resources is achieved through a many-state FSM, so the cell controller for SINGLE is much more complex than the controller for MULTIPLE. The SINGLE architecture is useful as a first attempt at decreasing area at the expense of time.

If an add does not need to be performed during an operation, then the adder resource is idle and is effectively wasted for that clock cycle. A similar situation holds whenever a multiply is not needed. Thus, it is desirable to partition the equations into sub-operations that include both a multiply and an add, as shown in Figure 4-7. The temporary variables t_1 and t_2 are essential and add to the number of register bits required by each cell, although this is not usually a problem for flip-flop-rich FPGAs. Some special operations are done in order to maximize cell performance. Lines 5 and 11 in Figure 4-7 make two concurrent assignments for the purpose of saving one clock cycle. Two numbers A and B can be subtracted by performing an operation such as $-1\times B + A = A - B$, but the hardware does not support direct subtraction as in $A\times B - C$.

$$
\begin{array}{ll}
1 & t_1 = 1 \times T_{m,j}^{i-1} + \left(-C_b\right) \\
2 & t_1 = t_1 \times t_1 + 0 \\
3 & t_1 = \begin{cases} 0, & T_{m,j}^{i-1} < -\left(-C_b\right) \\ \left(-\Delta t \cdot C_a\right) \times t_1 + 0, & T_{m,j}^{i-1} \geq -\left(-C_b\right) \end{cases} \\
4 & t_2 = \alpha_j^{i-1} \times \alpha_j^{i-1} + \left(-1\right) \\
5 & \alpha_j^i = t_1 \times t_2 + \alpha_j^{i-1}, \;\; t_1 = t_1 \times t_2 \\
6 & t_1 = \Gamma \times t_1 + 0 \\
7 & t_1 = \Delta t \times T_{p,j}^{i-1} + t_1 \\
8 & t_2 = 1 \times T_{m,j-1}^{i-1} + T_{m,j+1}^{i-1} \\
9 & t_1 = \frac{\Delta t}{\Delta x^2} \times t_2 + t_1 \\
10 & t_1 = \left(1 - 2\frac{\Delta t}{\Delta x^2} - \Delta t\right) \times T_{m,j}^{i-1} + t_1 \\
11 & t_1 = \Delta t \times T_{m,j}^{i-1} + 0, \;\; T_{m,j}^i = t_1 \\
12 & T_{p,j}^i = \left(1 - \Delta t\right) \times T_{p,j}^{i-1} + t_1
\end{array}
$$

Figure 4-7. Heat flow equations partitioned for the SINGLE architecture

The cell and controller structures for the SINGLE architecture are shown in Figure 4-8. The add and multiply units are the same as those used by MULTIPLE, but there is only one of each for this architecture. Each state in the cell controller's 12-state FSM determines the inputs applied to the adder and multiplier as well as the destination register that will latch the result of the multiply/add operation. The controller also distributes constant parameter inputs to all cells. Although not shown in Figure 4-8, each cell includes the same comparator hardware as MULTIPLE.

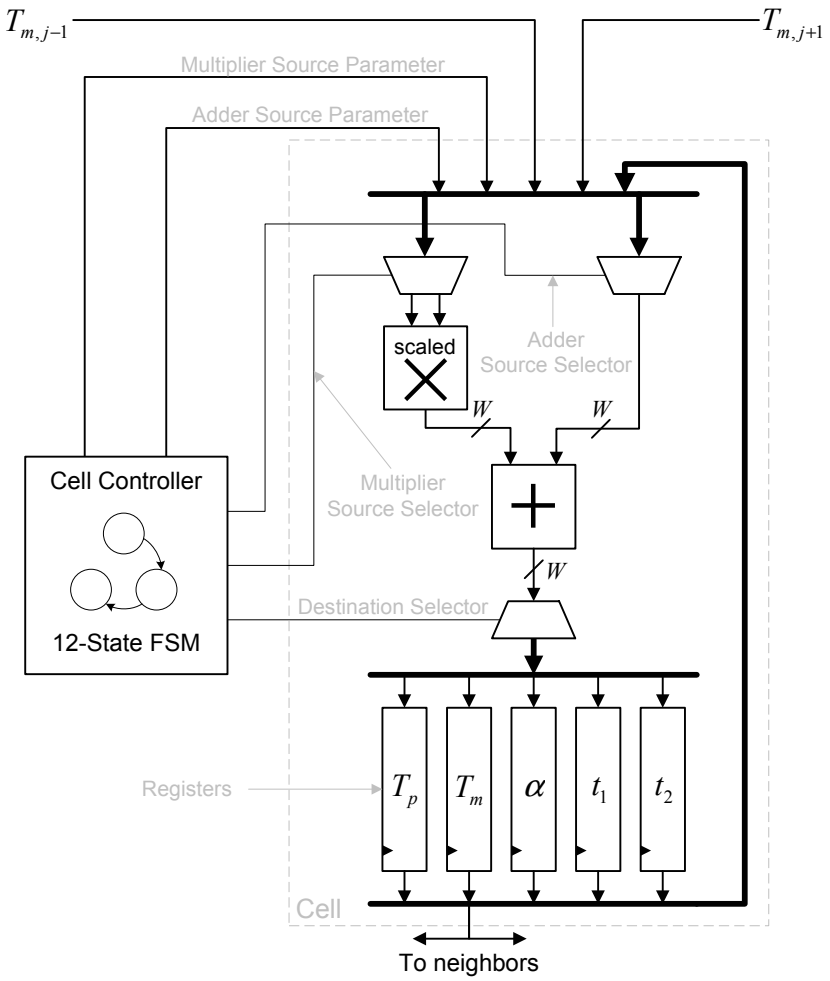

Figure 4-8. Cell and controller structures for the SINGLE architecture

BOOTH

This section discusses the cell controller and computing cell structures of the BOOTH architecture.

The strategy of decreasing size at the expense of time can be taken further by eliminating the large multiplier unit in SINGLE and doing all operations with a single *W*-bit adder. One way to perform signed-number multiply operations using a single adder is the Booth algorithm [10], which is the method used for this architecture and the reason for the name BOOTH. Using this method, each multiply operation takes *W* clock cycles, but full-width add operations can still be done in one cycle. The clock frequency may be able to be higher than for SINGLE since signals in the BOOTH architecture go through

one adder rather than a multiplier followed by an adder. Like SINGLE, the BOOTH architecture requires complex control in order to spread operations out over time in the correct order.

A string of W bits $a_{W-1}a_{W-2}\ldots a_1a_0$ represents a two's complement integer by

$$A = -a_{W-1}\cdot 2^{W-1} + \sum_{i=0}^{W-2} a_i \cdot 2^i = \sum_{i=0}^{W-1} \left(a_{i-1} - a_i\right)\cdot 2^i$$

if $a_{-1} = 0$ [38]. Therefore, if A is multiplied by another two's complement integer B, then

$$A \cdot B = \sum_{i=0}^{W-1} \left(a_{i-1} - a_i\right)\cdot 2^i \cdot B\,,$$

where each addend effectively adds B, subtracts B, or adds 0 to each partial sum term depending on if $a_{i-1}a_i$ is 10, 01, or 00/11, respectively. Booth's algorithm uses this fact and the fact that multiplication by 2^i can be done by shifting a bit sequence. The Booth algorithm and the required hardware are shown in Figure 4-10 and Figure 4-9, respectively. A and B are the two operands to the multiplication, and C accumulates the partial sums. The multiplication by 2^i is done by shifting C and A to the right. The adder contains additional hardware for subtracting B and for adding 0. The usual result of the multiplication is a two's complement integer that resides in the concatenation of registers C and A, but since scaled integers are being used, the result lies within C and A at the position shown in Figure 4-5.

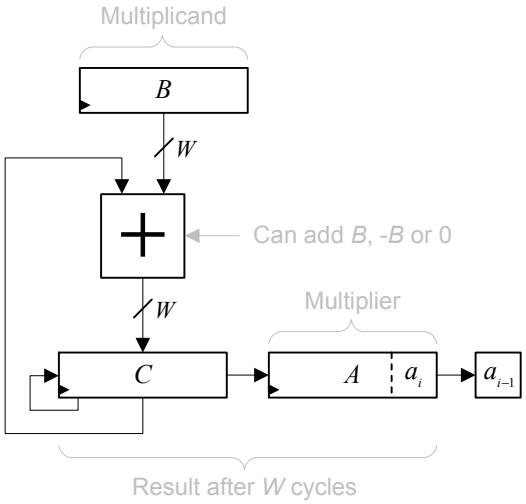

Figure 4-9. Hardware for Booth's algorithm

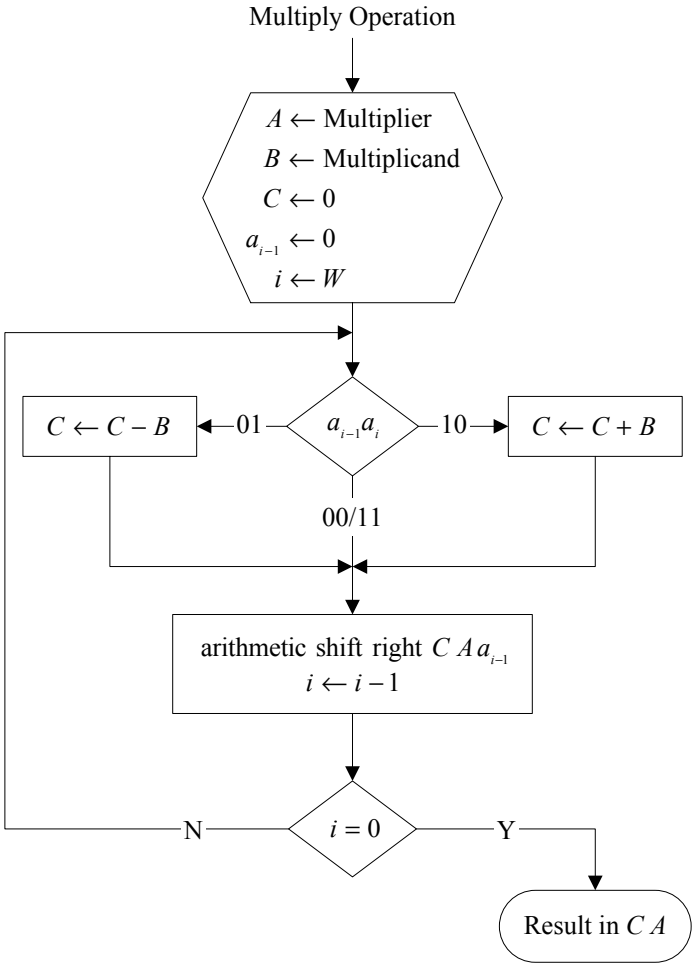

Figure 4-10. Booth's algorithm for two's complement multiplication

Like the MULTIPLE architecture, it is advantageous to compute the heat flow equations using the least number of multiply operations, but unlike MULTIPLE, the reason is not because of size but because of time: multiply operations are W times longer than add operations in the BOOTH architecture. Thus, the heat flow equations for the BOOTH architecture are partitioned as shown in Figure 4-11. Lines 11 and 12 implement $-2T_{m,j}^{i-1}$ and are done using two single-cycle add operations rather than a multiply, which would take W clock cycles. As with SINGLE, temporary variables t_1 and t_2 are essential and add to the register count of each cell.

1	$t_1 = T_{m,j}^{i-1} + \left(-C_b\right)$
2	$t_1 = \begin{cases} 0, & T_{m,j}^{i-1} < -\left(-C_b\right) \\ t_1 \times t_1, & T_{m,j}^{i-1} \geq -\left(-C_b\right) \end{cases}$
3	$t_1 = t_1 \times \left(-\Delta t \cdot C_a\right)$
4	$t_2 = \alpha_j^{i-1} \times \alpha_j^{i-1}$
5	$t_2 = t_2 + \left(-1\right)$
6	$t_1 = t_1 \times t_2$
7	$\alpha_j^i = t_1 + \alpha_j^{i-1}$
8	$t_1 = t_1 \times \Gamma$
9	$t_1 = t_1 + T_{m,j}^{i-1}$
10	$t_2 = T_{m,j-1}^{i-1} + T_{m,j+1}^{i-1}$
11	$t_2 = t_2 - T_{m,j}^{i-1}$
12	$t_2 = t_2 - T_{m,j}^{i-1}$
13	$t_2 = t_2 \times \frac{\Delta t}{\Delta x^2}$
14	$t_1 = t_1 + t_2$
15	$t_2 = T_{p,j}^{i-1} - T_{m,j}^{i-1}$
16	$t_2 = t_2 \times \Delta t$
17	$T_{m,j}^i = t_1 + t_2$
18	$T_{p,j}^i = T_{p,j}^{i-1} - t_2$

Figure 4-11. Heat flow equations partitioned for the BOOTH architecture

The cell and controller structures for the BOOTH architecture are shown in Figure 4-12. Each cell uses a single adder, but the C and A registers needed for the Booth algorithm increase the register count of the cell. The B register of the Booth algorithm is not explicitly included but is effectively the cell register that passes through the input-2 select logic. This requires the cell controller to hold the proper select lines until a multiplication completes. Each state in the cell controller's 18-state FSM determines both of the adder's inputs and the destination register that will latch an operation's result. The controller also distributes a constant parameter input to all cells and selects the operation to be either an add or a multiply. The cell controller has the ability to select a subtraction operation so that operations like $A - B$ can be performed. Although not shown in Figure 4-12, each cell includes a bitwise less-than unit and is instructed to do the test during line 2 in Figure 4-11, which is a multiply operation that takes the same number of clock cycles as the less-than test. This dual use of the W clock cycles saves area that would be consumed by using a full-width less-than unit. The counter in the cell controller counts W cycles for multiply operations and is implemented with a simple shift register rather than an incrementer in order to save area and increase speed. In this implementation, multiply operations take $W + 2$ cycles due to required initializations, but it may be possible to construct a design such that they take W cycles. Rather than generating $-B$ with a negation unit, $\overline{B}$ is provided and $+1$ is generated in the carry logic, which add together to produce a properly negated B.

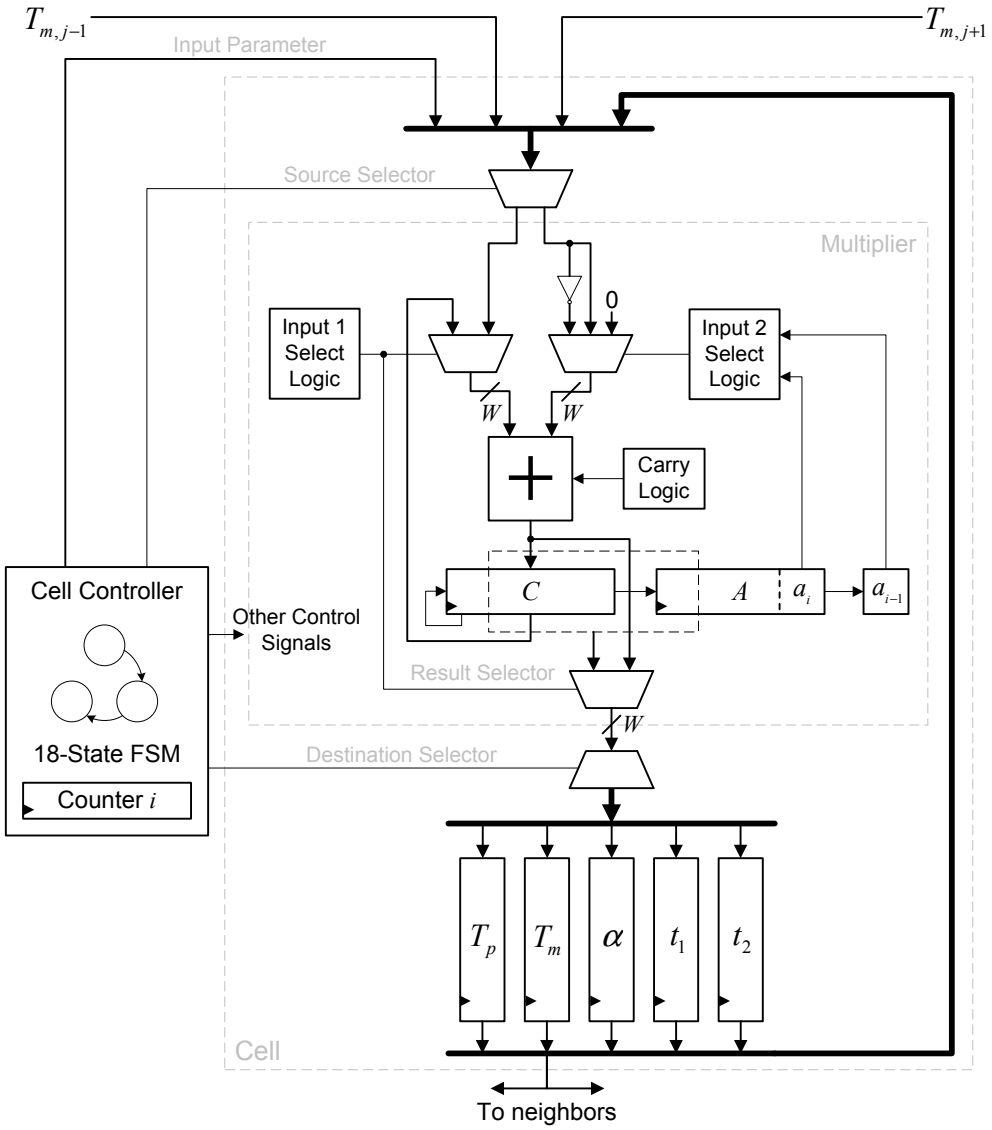

Figure 4-12. Cell and controller structures for the BOOTH architecture

BIT

This section discusses the cell controller and computing cell structures of the BIT architecture.

The MULTIPLE architecture can be thought of as the extreme in terms of requiring the least amount of time at a very high size expense. At the other end of the spectrum is an architecture that requires the least amount of space at a very high time expense. The way to achieve this extreme is by performing all operations one bit at a

time, which is what the BIT architecture does, hence the reason for its name. Like BOOTH, the BIT architecture uses the Booth algorithm to perform multiplies, but each add operation is bit-serial. Thus, multiplies take W^2 cycles and additions take W cycles, although the clock frequency may be able to be higher than for BOOTH because signals in the BIT architecture go through a 1-bit adder rather than a full-width adder. Although the BIT architecture consumes the least amount of area per cell, its VHDL description is the most complex of the four architectures due to the complicated timing control required.

Because of the BIT architecture's similarities to BOOTH, the heat flow equations are partitioned in nearly the same way as they are for the BOOTH architecture. The only differences between the BOOTH partitioning shown in Figure 4-11 and the BIT partitioning are the first four lines. The BIT architecture's first four operations are shown in Figure 4-13. The reason for the differences in these lines is due to an important constraint on the way in which multiply operations can be performed in the BIT architecture. The two operands to a multiply operation are shifted through the hardware one bit at a time but at different rates: the right operand rotates every clock cycle while the left operand rotates with the destination register every W cycles. Thus, the right operand cannot be the same as the destination register, which is the reason why both t_1 and t_2 are assigned in line 1 and why t_2 is set in line 3: so that the operations in lines 2 and 4 can be performed. As with SINGLE and MULTIPLE, t_1 and t_2 are essential temporary variables.

$$
\begin{array}{ll}
1 & t_1, t_2 = \begin{cases} 0, & T_{m,j}^{i-1} < -(-C_b) \\ T_{m,j}^{i-1} + (-C_b), & T_{m,j}^{i-1} \geq -(-C_b) \end{cases} \\
2 & t_1 = t_1 \times t_2 \\
3 & t_1 = t_1 \times (-\Delta t \cdot C_a),\ t_2 = \alpha_j^{i-1} \\
4 & t_2 = t_2 \times \alpha_j^{i-1}
\end{array}
$$

Figure 4-13. First four lines of the BIT architecture's partitioning of the heat flow equations

The cell and controller structures for the BIT architecture are shown in Figure 4-14. All registers are shift registers so that they can be read and written one bit at a time, but only the *C* and T_m registers are implemented with flip-flops. The other registers are implemented as LUT shift registers, otherwise they would quickly consume all FPGA flip-flops due to the large number of cells that can fit on the chip. Like BOOTH, the *B* register of the Booth algorithm is excluded, but the *A* register is also excluded in the BIT architecture because result bits can be shifted into their destination register as they are generated. Unlike BOOTH, selecting the correct result bits requires complex timing control in the cell controller. Because the *B* register is excluded, the cell controller must also hold the input-2 select lines throughout a multiplication and rotate the multiplicand a total of W^2 times. The required timing is achieved using two *W*-cycle counters, each implemented as a simple shift register to save area and increase speed. In this implementation, multiply and add operations take W^2+W and $W+1$ cycles, respectively, due to required initializations, but it may be possible to construct a design such that they take W^2 and *W* cycles. Each state in the cell controller's 19-state FSM determines both of the adder's 1-bit inputs and the destination register that will rotate in result bits as they are generated. The controller also distributes a constant parameter input one bit at a time to all cells and selects the operation to be either an add or a multiply. The cell controller has the ability to select a subtraction operation so that operations like $A-B$ can be performed. The BIT architecture requires 19 states instead of the 18 for BOOTH because the bitwise less-than test requires a separate operation, which occurs just before line 1 in Figure 4-13. There are also other states required for initialization and synchronization with SSRAM, but some of them occur only once at the beginning of a calculation cycle.

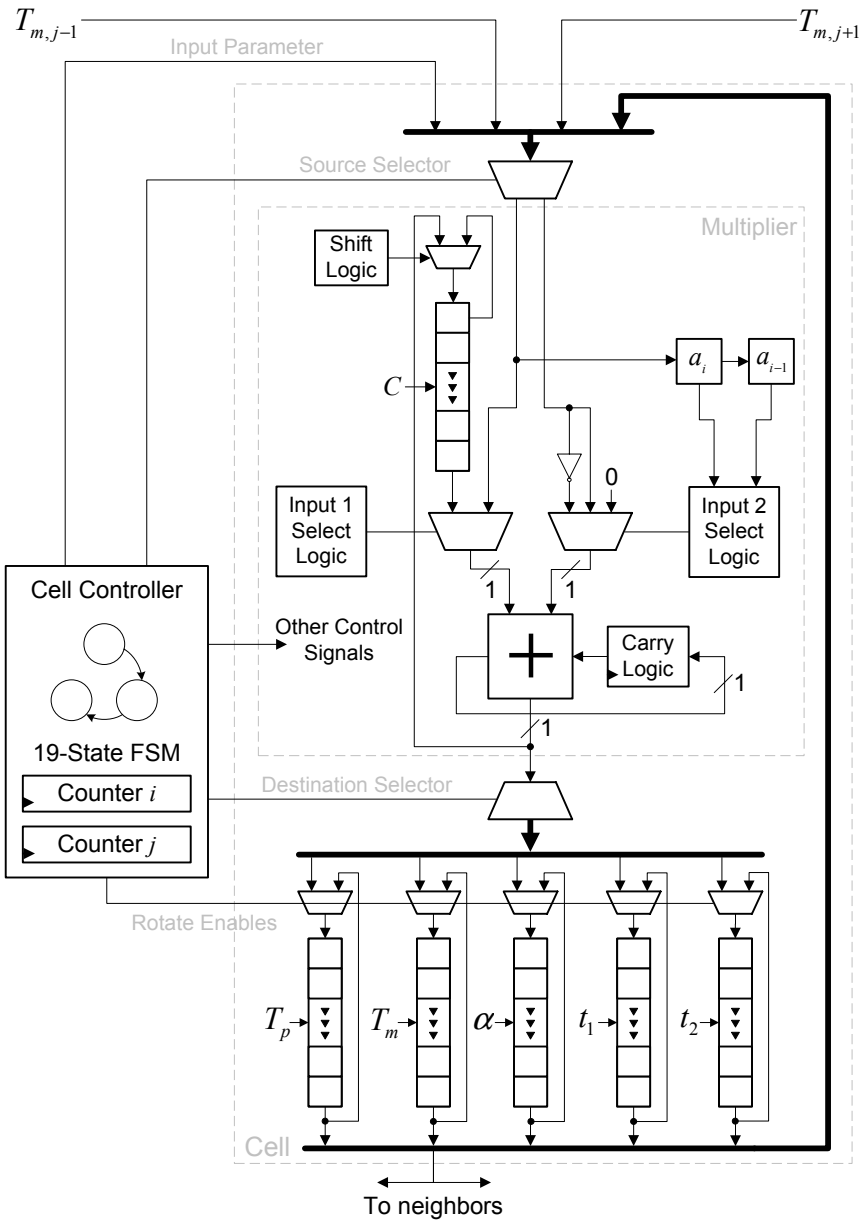

Figure 4-14. Cell and controller structures for the BIT architecture

Chapter 5
Results and Analysis

Each architecture described in Chapter 4 was implemented using the procedures outlined in Chapter 3. This chapter gives the performance results of each architecture followed by a brief analysis and discussion of the results. Also included in this chapter is a simple verification of each architecture's computations as well as some of the alternative designs considered in an effort to increase performance.

Design Verification

As a way of verifying that the designs operate as intended, Matlab scripts containing each architecture's sequence of operations were written. These scripts perform fixed-point scaled integer arithmetic and are expected to produce results identical to those computed by the FPGA-system. The results produced by executing the scripts are shown in Table 5-1 next to the values produced by simulating the VHDL descriptions using Cadence tools. Additionally, simulation waveforms are shown in Figure 5-1. For each architecture, the scaled integer matrix temperature of cell 1 of 50 is shown for the first 10 time steps with $W = 25$, $S = 5$, and $T_{m,left} = T_{m,right} = \Gamma = 2.6$. It can be seen that the VHDL descriptions behave as expected. The SINGLE architecture computes values that are slightly different than the other three architectures because of the differences in its partitioning of the heat flow equations. Each architecture simulation in Figure 5-1 uses the same clock frequency making it apparent that for a given clock rate, decreasing parallelism in each cell increases the computation time.

Table 5-1. Verification of 10 simulated time steps for each architecture

MULTIPLE		SINGLE		BOOTH		BIT	
Matlab	**VHDL**	**Matlab**	**VHDL**	**Matlab**	**VHDL**	**Matlab**	**VHDL**
0	0	0	0	0	0	0	0
1349246	1349246	1349246	1349246	1349246	1349246	1349246	1349246
1362739	1362739	1362737	1362737	1362739	1362739	1362739	1362739
1693341	1693341	1693338	1693338	1693341	1693341	1693341	1693341
1703256	1703256	1703252	1703252	1703256	1703256	1703256	1703256
1865335	1865335	1865330	1865330	1865335	1865335	1865335	1865335
1873433	1873433	1873426	1873426	1873433	1873433	1873433	1873433
1972797	1972797	1972791	1972791	1972797	1972797	1972797	1972797
1979742	1979742	1979735	1979735	1979742	1979742	1979742	1979742
2047997	2047997	2047989	2047989	2047997	2047997	2047997	2047997
Discrepancies: 0		Discrepancies: 0		Discrepancies: 0		Discrepancies: 0	

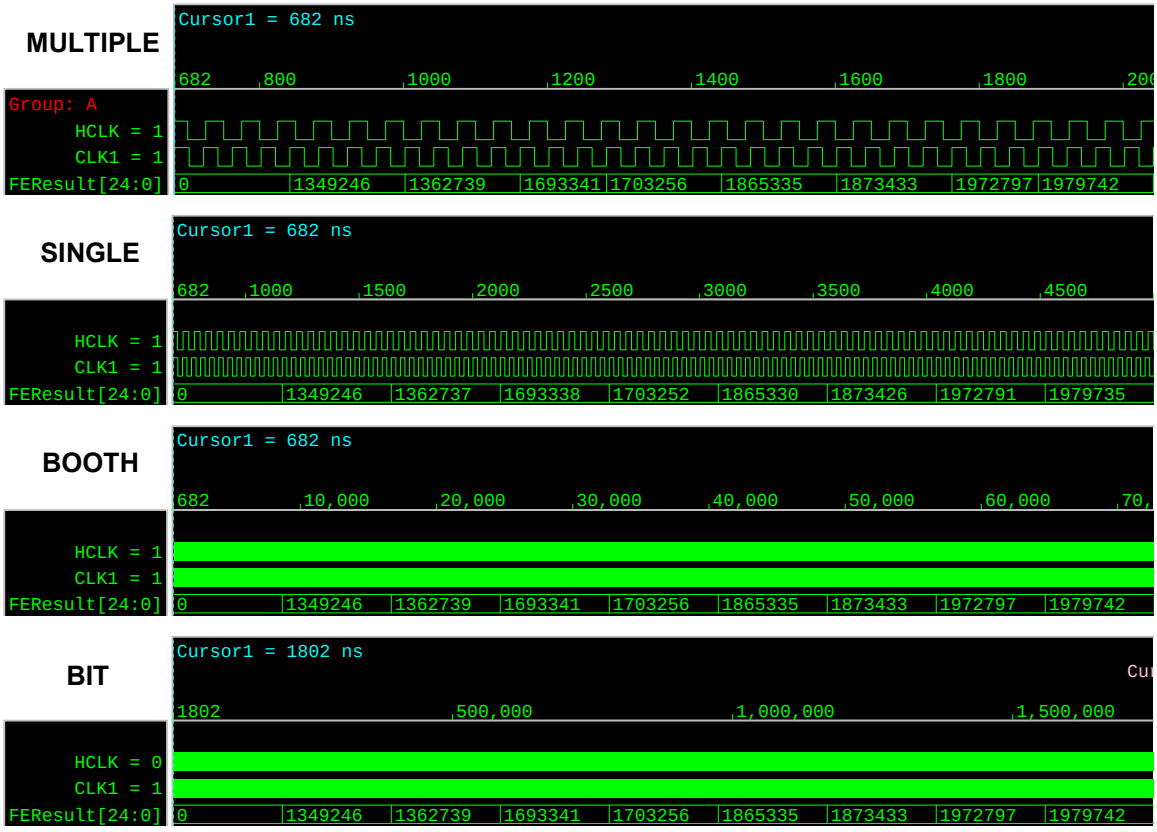

Figure 5-1. Simulation waveforms for each architecture

A hardware implementation may not always function as predicted by simulation, so values produced by the FPGA system were verified by comparing them to values from the Matlab scripts as shown in Table 5-2. The input parameters are the same as those

used for Table 5-1, but each architecture uses the maximum number of cells that can fit in the FPGA. It can be seen that the hardware implementations behave as expected. While only 10 time steps are shown in Table 5-2, the results of over 200,000 time steps were verified in the laboratory.

Table 5-2. Verification of 10 hardware-executed time steps for each architecture

MULTIPLE Cell 1 of 10		**SINGLE** Cell 1 of 37		**BOOTH** Cell 1 of 76		**BIT** Cell 1 of 424	
Matlab	**FPGA**	**Matlab**	**FPGA**	**Matlab**	**FPGA**	**Matlab**	**FPGA**
0	0	0	0	0	0	0	0
1342800	1342800	1349022	1349022	1349399	1349399	1349511	1349511
1356231	1356231	1362511	1362511	1362894	1362894	1363009	1363009
1682151	1682151	1692948	1692948	1693608	1693608	1693805	1693805
1691959	1691959	1702857	1702857	1703528	1703528	1703727	1703727
1850262	1850262	1864801	1864801	1865698	1865698	1865965	1865965
1858226	1858226	1872890	1872890	1873800	1873800	1874072	1874072
1954394	1954394	1972141	1972141	1973245	1973245	1973573	1973573
1961187	1961187	1979075	1979075	1980194	1980194	1980528	1980528
2026658	2026658	2047230	2047230	2048520	2048520	2048904	2048904
Discrepancies: 0		Discrepancies: 0		Discrepancies: 0		Discrepancies: 0	

Because the FPGA system uses limited-precision fixed-point numbers, it is important to verify the accuracy of the computed results by comparing them to higher-precision results. Figure 5-2 shows 7000 FPGA-computed time steps using the same setup used in Figure 2-3 but converted from scaled integers to real numbers. Overlaying these are the results of the same computation using machine-precision arithmetic on a PC. Relative percent error, as shown in Figure 5-3, is commonly used to quantify the accuracy of measured results, but in this case, relative error is not very useful because it generates high-magnitude transient error spikes at early time steps due to comparisons between very small numbers. Therefore, Figure 5-3 also shows an alternative method of quantifying the error that uses

$$100 \times \frac{g(n) - f(n)}{\max(f) - \min(f)},$$

which can be interpreted as the "visual" percent error. This is effectively the percentage of the range of f that each value in g is different than the corresponding value in f, where f is the set of accepted values. Figure 5-3 shows that "visual" error is similar to relative

error but that it does not have the transient problem. The fixed-point scaled integer results are accurate to within 4%, which is acceptable since input parameters to the heat flow test case are usually only known to within 5% to 20%. The error occurs to the greatest extent around the curing spike and is essentially zero everywhere else indicating that nearly all of the error is due to the computations associated with the curing effect.

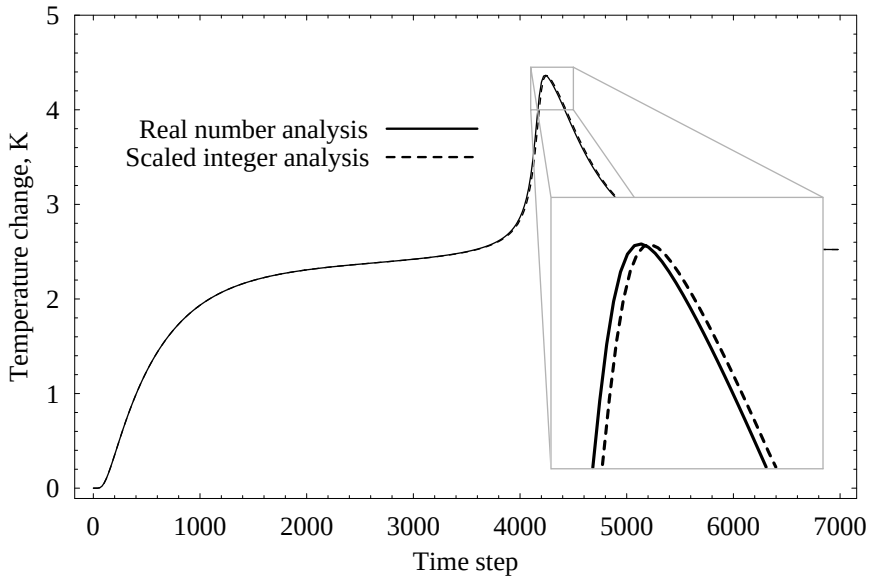

Figure 5-2. Comparison of results computed with real numbers and with scaled integers

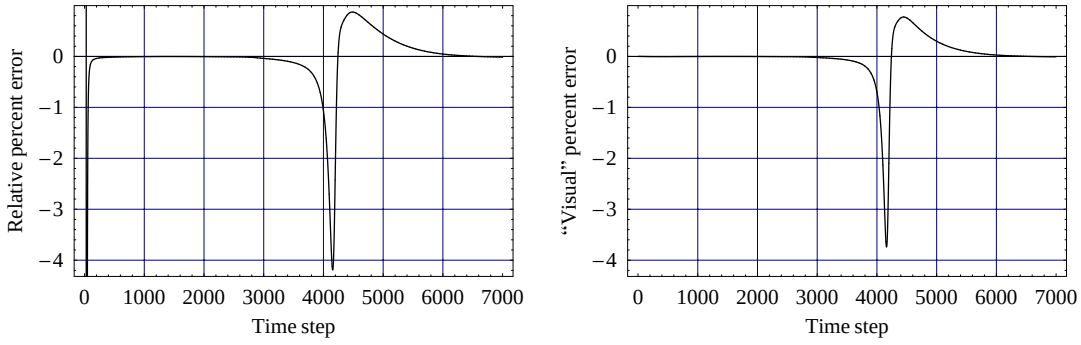

Figure 5-3. Two methods of quantifying the accuracy of the scaled integer computation

Results and Comparisons

The number of cells was increased for each architecture until the design could no longer fit on the FPGA or until it became difficult for the synthesis tools to operate efficiently. Using the procedure described in the Appendix, the synthesis tools were set up to achieve a near-optimal clock frequency. As a reference for the synthesis times below and the speedups reported in the next section, Table 5-3 is a description of the PC used for design and testing.

Table 5-3. PC setup used for design and testing

Model:	Dell Inspiron 5100
CPU:	Intel Pentium 4 2.66GHz
RAM:	512MB
Operating System:	Windows XP Professional SP2

The following graphs show various performance results of each architecture. Figure 5-4 shows the total synthesis time, which is the time required by Synplify to compile the VHDL and perform the first synthesis stage added to the time required by ISE to perform the second synthesis stage and generate a programming file added to the time required to load the LM flash with configuration data. Figure 5-5 shows the maximum number of cells, N_{max}, that fit in the chip for each architecture. Figure 5-6 shows the worst-case maximum clock frequency at which the computing cells can run. This value is reported by ISE after synthesis as an estimate of the maximum clock frequency that should remain valid through acceptable device tolerances and operating conditions. Figure 5-6 also shows the actual maximum clock frequency, f_{max}, achieved in the laboratory. This was determined by stepping up the clock frequency by 1MHz until the FPGA produced erroneous results. Figure 5-7 shows the number of clock cycles, N_{cycles}, required to complete one iteration as determined by the number of states in the cell controller's FSM and by the type of operation performed in each state. Figure 5-8 shows each architecture's calculation rate, which is given by

$$\frac{N_{max} f_{max}}{N_{cycles}}.$$

Figure 5-9 shows the time required to compute 200,000 iterations and send the 32-bit results to Matlab over two 460,800 baud serial links.

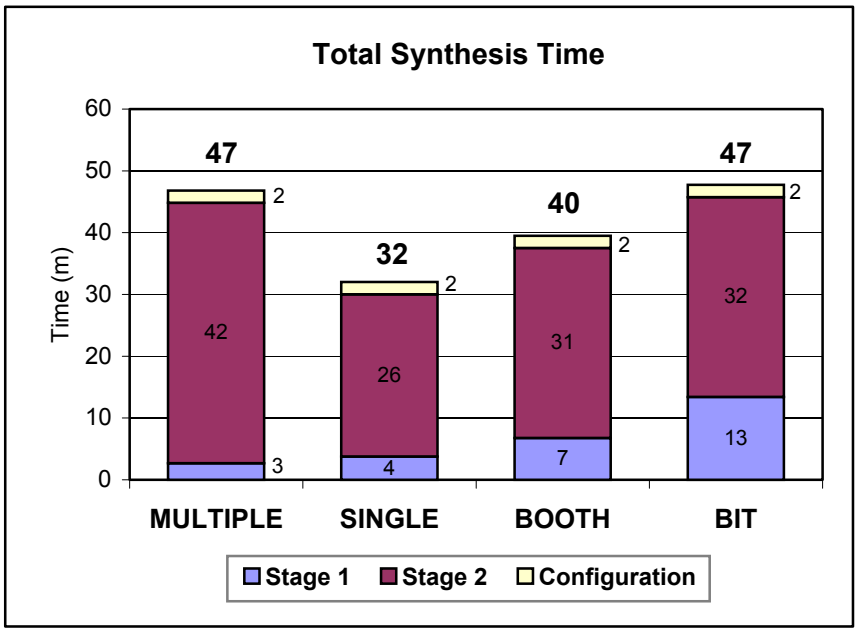

Figure 5-4. Total synthesis time for each architecture

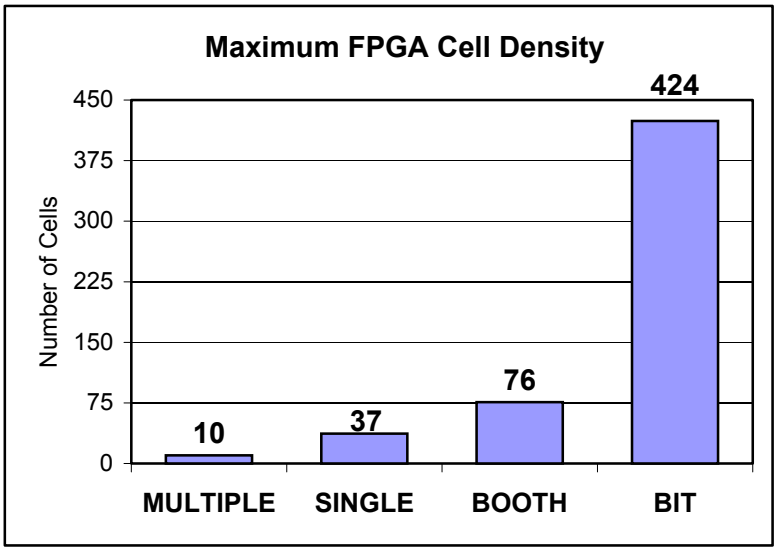

Figure 5-5. Maximum number of cells that fit in the FPGA for each architecture[1]

[1] 38 cells fit for SINGLE and 77 cells fit for BOOTH, but synthesis becomes difficult in both cases.

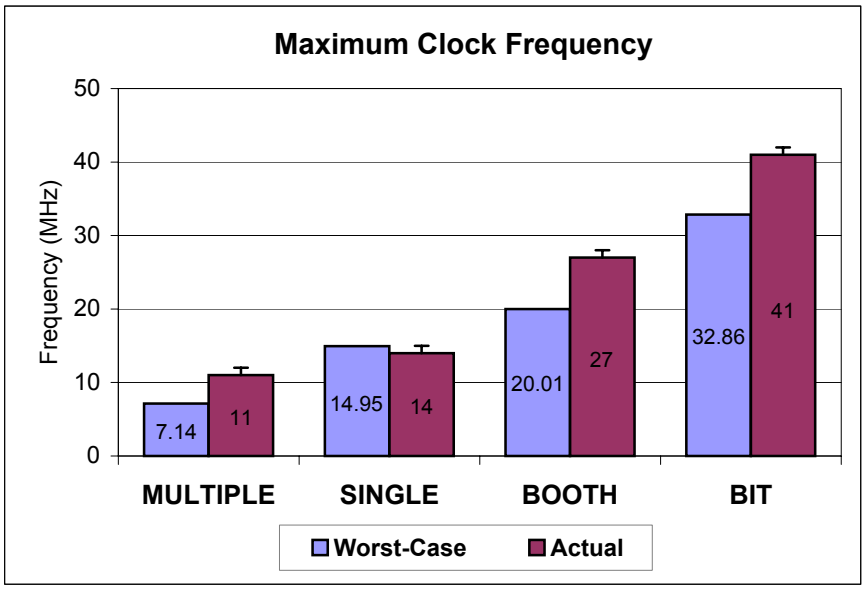

Figure 5-6. Worst-case and actual maximum clock frequency for each architecture[2]

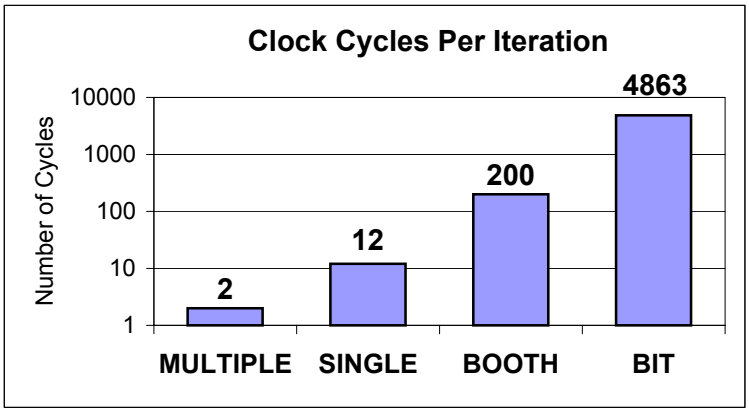

Figure 5-7. Number of clock cycles required for one iteration for each architecture

[2] Actual maximum frequencies measured to 1MHz precision

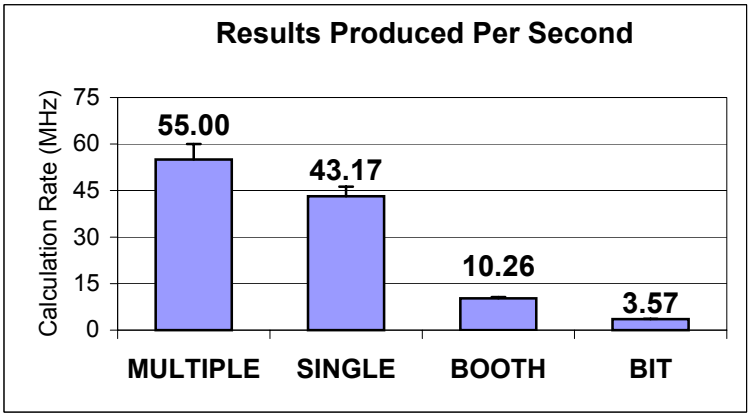

Figure 5-8. Calculation rate of the entire cell row for each architecture

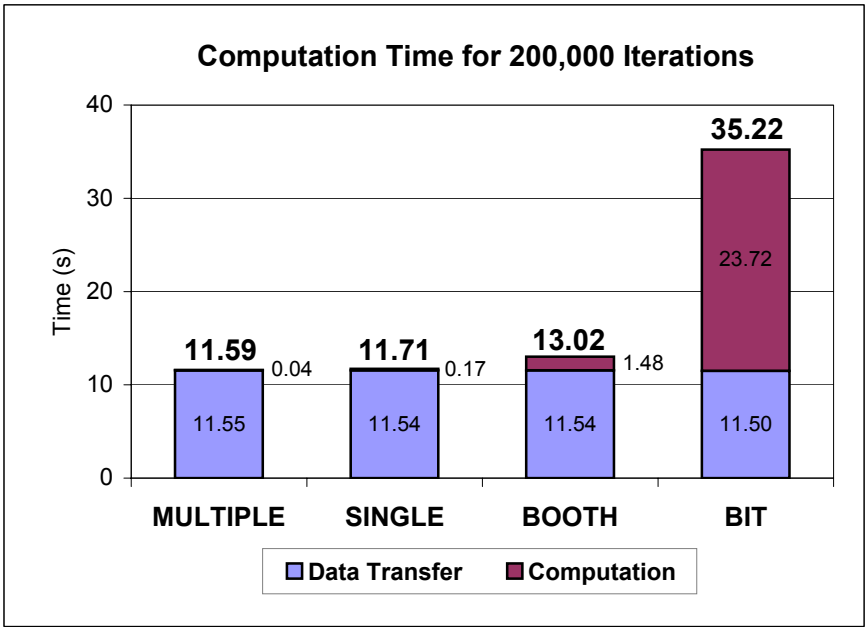

Figure 5-9. Time required to compute and transfer the results of 200,000 time steps

Analysis

Figure 5-8 shows that the MULTIPLE architecture has the highest calculation rate, although its usefulness is severely limited by the small number of cells that fit in the FPGA. Conversely, the BIT architecture has the lowest calculation rate but permits a very large cell density. In order to stress the importance of a large number of cells,

Figure 5-10 shows a performance measurement that gives more weight to the cell density by squaring N_{max} to give

$$\frac{(N_{max})^2 f_{max}}{N_{cycles}}.$$

Using this index, the SINGLE and BIT architectures show the best performance.

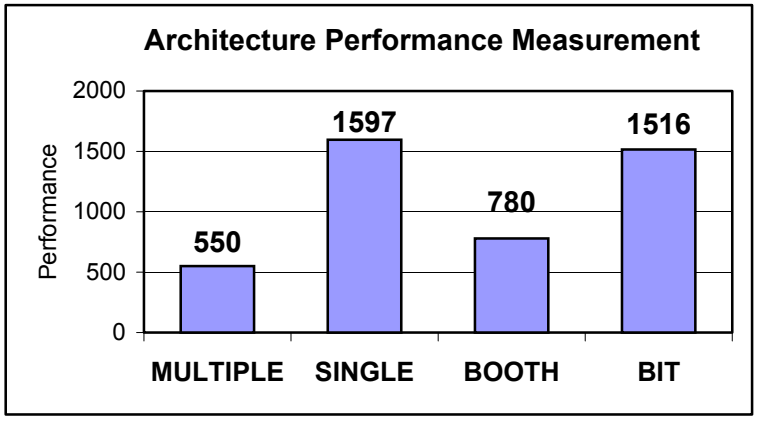

Figure 5-10. Performance index that gives more weight to cell density

While the performance measurements given above are interesting, the most important measurement is the architecture's ability to accelerate calculations done with a single-processor machine. Figure 5-11 shows the speedups obtained over four common methods that researchers might use to do computations on a PC. The four methods are:

- *Mathematica*: This method uses the `Nest` function in Mathematica 5.0.1.
- *Matlab 1*: This method uses nested loops in Matlab 6.5 but simulates how the computation would take place in a pre-6.5 version without Performance Acceleration.
- *Matlab 2*: This method uses vector operations as recommended in the Matlab 6.5 documentation [24].
- *Matlab 3*: This method also uses nested loops in Matlab 6.5 but takes advantage Matlab's Performance Acceleration feature. Introduced in version 6.5, this feature accelerates certain loops by translating them into highly optimized code.

Each method was used to compute 200,000 time steps of the heat flow equations shown in Figure 4-4 using standard real number machine arithmetic. Figure 5-11 shows that the BIT architecture is actually slower than two Matlab methods and that the MULTIPLE architecture provides the greatest speedup. It should be emphasized, however, that the computation times for MULTIPLE are relatively short on both the FPGA and the PC, so the real benefit of the FPGA-based MULTIPLE architecture would be seen when doing repetitive calculations such as the temperature dynamics of all cells over time.

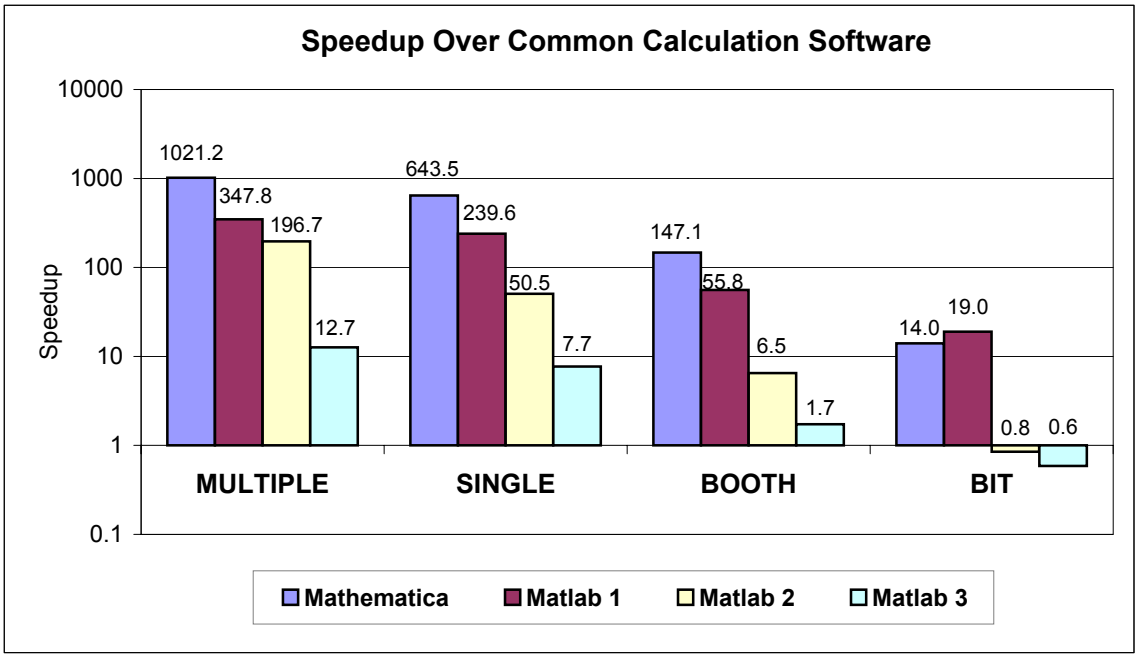

Figure 5-11. Each architecture's speedup over Mathematica and three calculation methods in Matlab[3,4]

The Synplify synthesis tool reports estimated maximum clock frequencies after the mapping process that are usually fairly accurate, but they are significantly overestimated for some of the architectures in this design. This may be due to the high routing delays, which, among other things, can be caused by the near-maximum device utilization. The Xilinx documentation [42] suggests that a typical design's total delay should be 40% logic and 60% routing delay, but for this design, the total delay is around 20% logic and 80% routing. Different synthesis techniques may be able to reduce the

[3] The FPGA's execution time does not include the time required to transfer computed results to a host PC.
[4] Performance results were obtained using the PC system in Table 5-3.

routing delay and achieve Synplify's estimated frequencies. Figure 5-12 shows Synplify's maximum clock frequency estimates and Figure 5-13 shows the speedups possible if these frequencies can be realized.

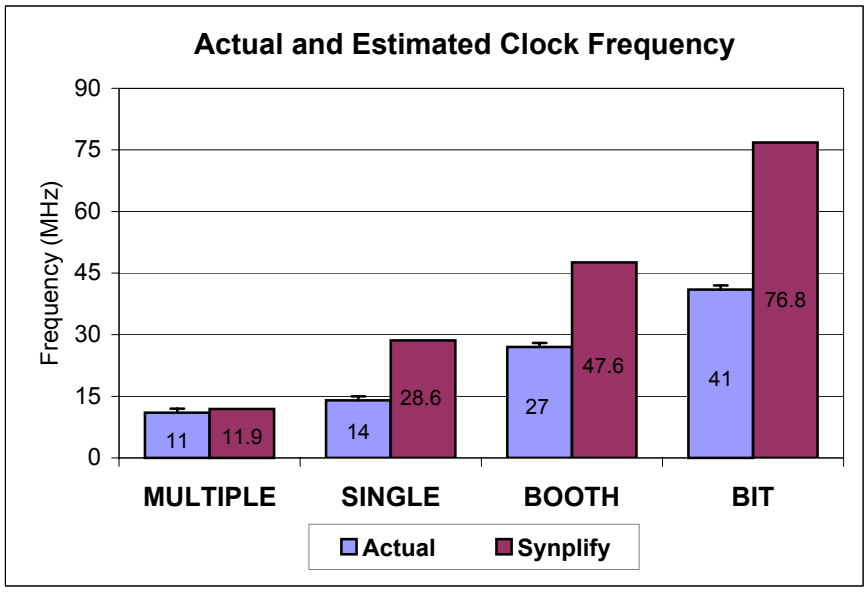

Figure 5-12. Maximum clock frequencies estimated by Synplify

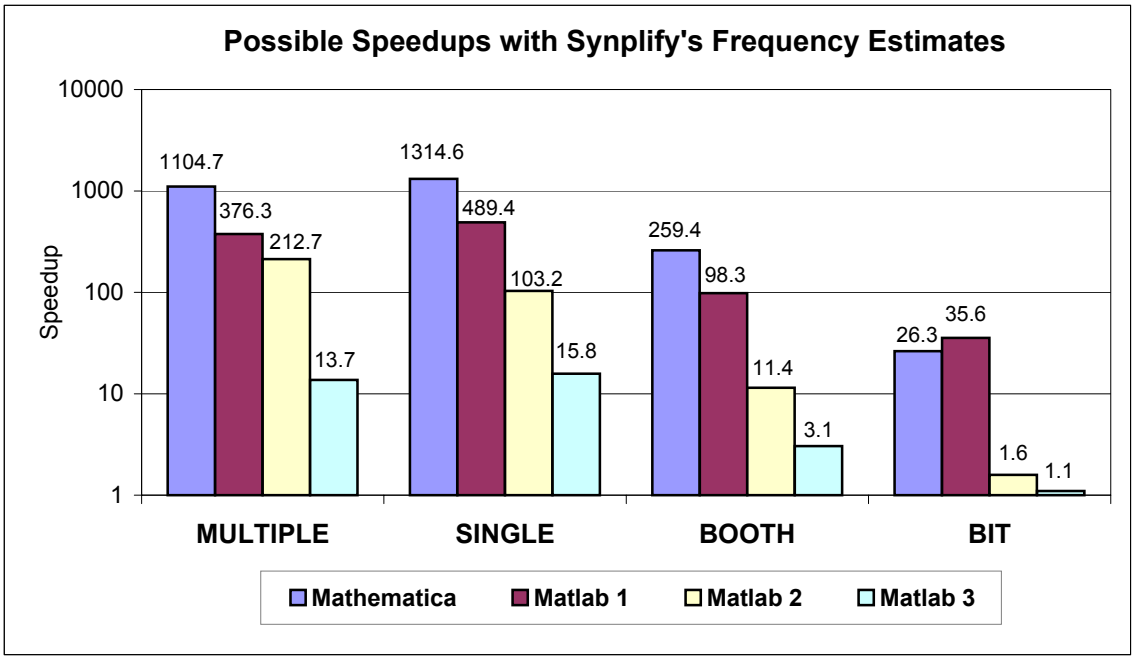

Figure 5-13. Speedups possible if Synplify's frequency estimates can be achieved

Next-generation FPGAs from Xilinx will have 4.6 times as many logic cells and will operate 3.8 times as fast as the XCV2000E device [43]. Figure 5-14 shows the speedups possible with these next-generation FPGAs assuming that the computing cells can operate 3.8 times as fast as the maximum frequencies reported by Synplify and that PC execution times remain the same.

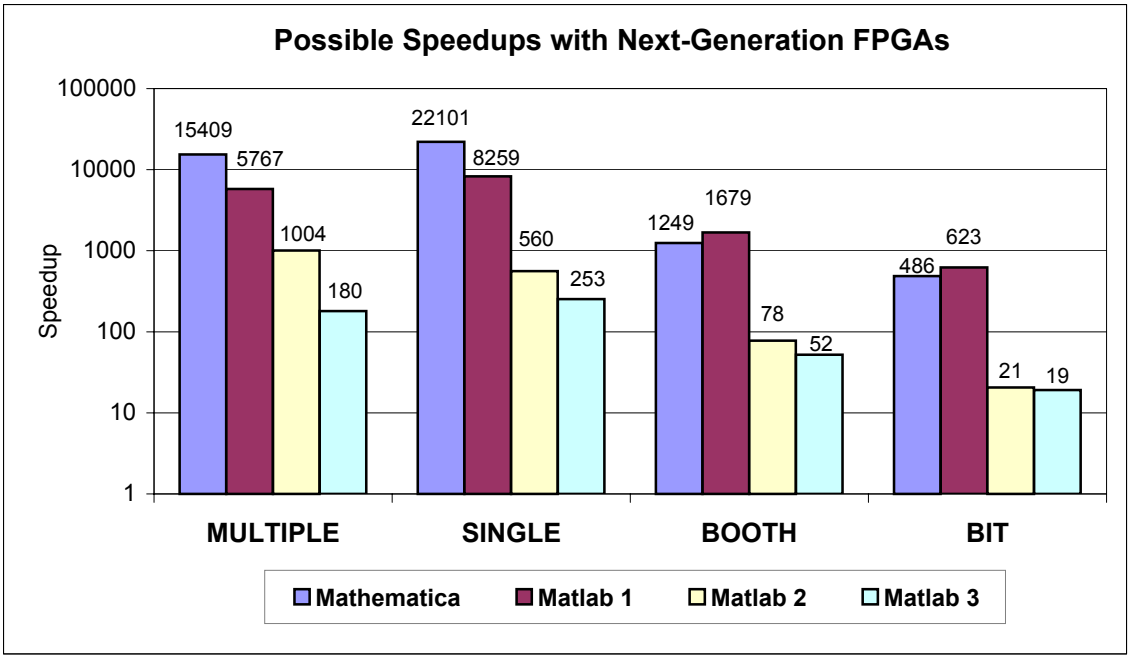

Figure 5-14. Speedups possible with next-generation FPGAs from Xilinx

Figure 5-15 shows the performance/cost ratios possible with next-generation FPGAs for each architecture assuming an FPGA cost of $500. The ratio is given by

$$\frac{N_{max} f_{max}}{N_{cycles}} \Big/ \$_{FPGA} .$$

For comparison, Figure 5-15 also shows the performance/cost ratio achieved by the Virginia Tech "X" cluster supercomputer, which is given by 10×10^{12} ops/$5.2 million. It can be seen that at least two of the architectures are expected to compare favorably to this large, expensive supercomputer.

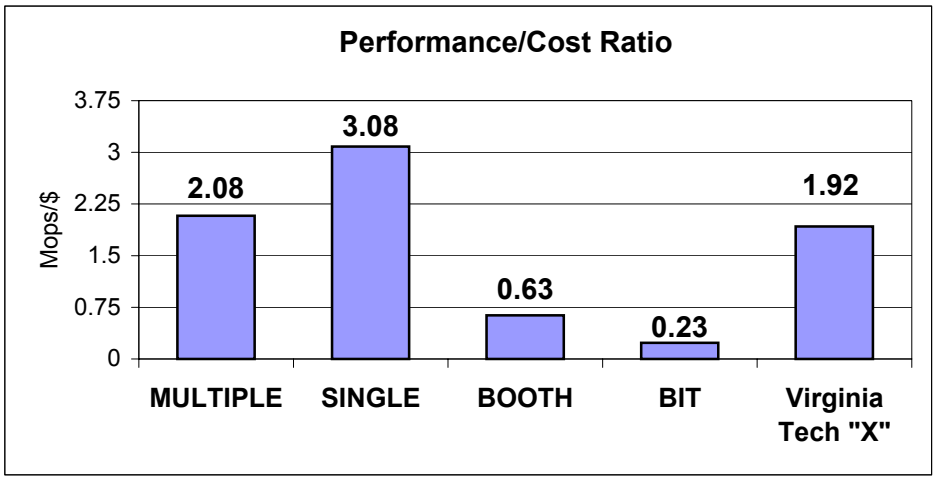

Figure 5-15. Performance/cost ratios for each architecture and for a cluster supercomputer

Although the FPGA-based system is aimed at accelerating calculation software like Matlab and Mathematica, it is informative to compare the FPGA to more efficient PC execution methods. Figure 5-16 shows each architecture's speedup over an optimized C program that calculates the heat flow equations using floating-point arithmetic on the PC system in Table 5-3. Only floating-point is shown because integer arithmetic in C is only slightly faster than the efficient *Matlab 3* method discussed above. Figure 5-16 shows that the single-chip FPGA-based computer discussed in this thesis is currently not able to outperform C code on a PC.

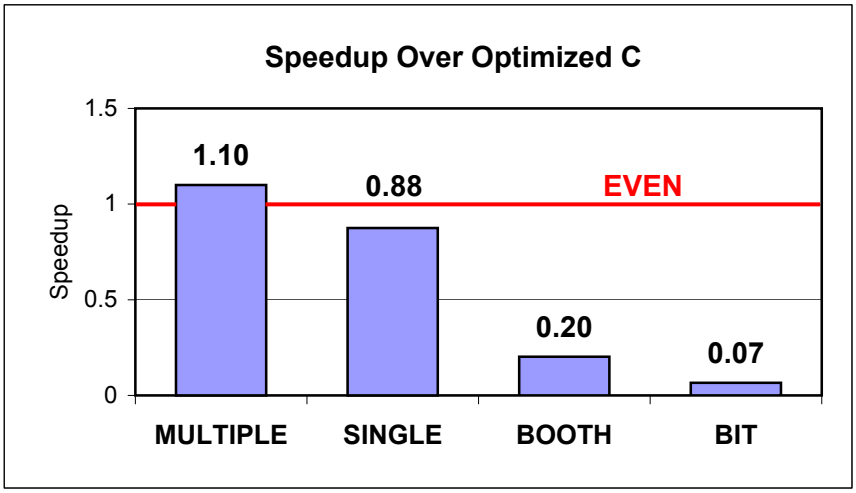

Figure 5-16. Each architecture's comparison to optimized C code

Alternative Designs Considered

A few alternative designs were implemented in an attempt to increase the cell clock rates, although none of them succeeded in providing any significant improvement. Most of the alternatives were attempted for the BIT architecture, but the results apply to the other architectures as well.

The control signals generated by the cell controller in the BIT architecture have an extremely high fanout because they must drive hundreds of cells, and this high fanout can limit the maximum clock frequency. In an attempt to reduce the fanout, the cell controllers were replicated using an algorithm in VHDL that evenly distributes C controllers over N cells so that the number of cells that each controller must drive is reduced to at most $\lfloor N/C \rfloor + 2$. In a separate attempt to reduce fanout, the changeable input parameter Γ, which also must be distributed to all cells, was eliminated by making it a constant parameter. Neither of these attempts made any improvement, which is probably due to the synthesis tool's own fanout reduction by logic replication and buffer insertion. Replicating cell controllers actually has a negative effect on performance because it reduces the number of cells that fit in the chip.

The ability to observe the results of any cell is achieved by using a large multiplexer that collects the output of every cell in the chip. This multiplexer is especially large for the BOOTH architecture because of the large number of full-width cell results that must be selected. To observe its effect on performance, the multiplexer was removed by hard-coding the number of the cell under observation. No performance increase resulted from the removal, which could be due to routing delays being much higher than the multiplexer delay or to an efficient multiplexer implementation by the synthesis tools.

An FPGA containing an array of computing cells brings to mind a regular matrix of well-defined cell boundaries. However, the synthesis tools actually spread a particular cell across the chip in a semi-random layout. Because cells communicate only with their nearest neighbors, it seems logical that placing cells next to their neighbors could increase performance by minimizing the distance that signals must travel. The synthesis tools allow a design to be floorplanned, which constrains selected logic to specific chip

areas, and Mathematica was used to generate area constraints that tile the row of cells in a snake-like pattern throughout the chip. Figure 5-17 shows the chip layout with and without these area constraints and with one cell darkened. Surprisingly, floorplanning does not increase the maximum clock frequency, and it has the negative side effect of decreasing the number of cells that fit in the chip. Also surprising is that even without floorplanning, decreasing the number of cells to make the design easier to synthesize does not seem to give better performance.

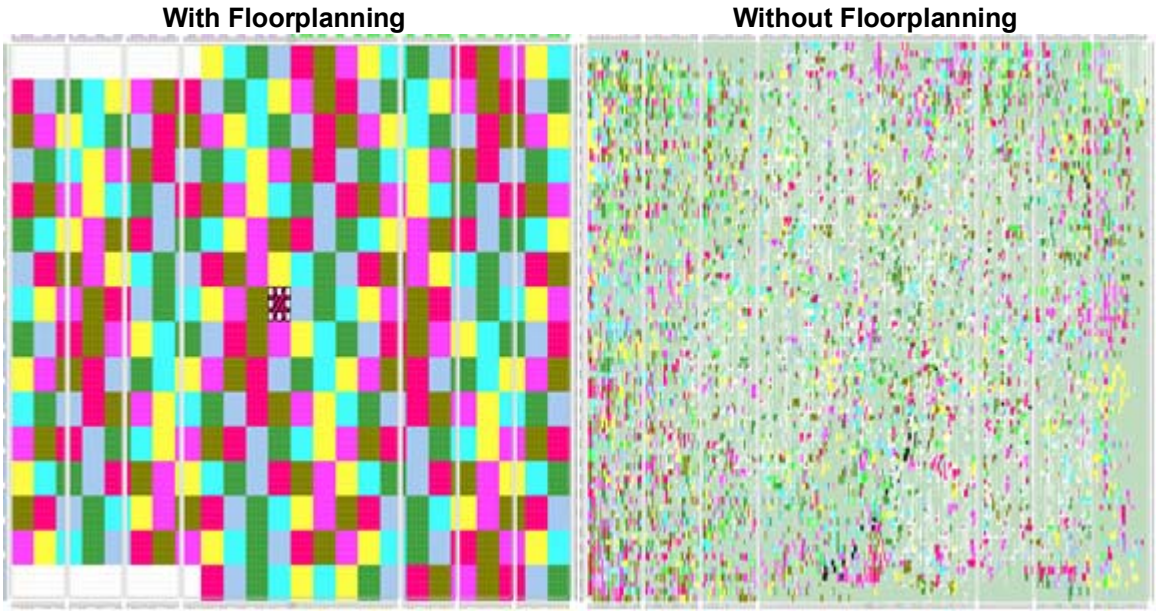

Figure 5-17. FPGA design layout with and without floorplanning

Chapter 6
Conclusions

This chapter outlines the procedure necessary for adapting the FPGA computing system to problems beyond the heat flow case study. It also recommends areas that need improvement and suggests possibilities for future research. The chapter concludes with a summary of the work presented in this thesis.

Extending the Designs to Other Problems

While the VHDL design descriptions are written to allow any number of computing cells, the code is deliberately problem-specific in order to maximize the computation rate of the particular problem. As a result, extending the descriptions to other problems requires considerable code modification. The code describing the interface between the FPGA and the development board is problem-independent and requires no modification, but extending the design requires the following changes to the problem-specific code.

- The problem's equations must be partitioned appropriately for the target architecture, and the resulting operations must be listed as states in the cell controller.
- Control signals generated by each state must be set to perform the associated operation.
- The controller's state variable must be modified to accommodate the required number of states.
- Enumerations for the source and destination selectors should be given proper names, and register rotate enables must be listed when targeting the BIT architecture.

- Input parameters must be distributed as needed to the row of cells.
- The cell structure must include all required registers, and these registers must be interfaced to the arithmetic resources according to the source and destination selectors from the cell controller.
- The common definitions file must be given the fixed-point parameters *W* and *S*, the number of cells *N*, and any constant parameters used by the calculation.

Future Work

The foremost problem with the FPGA-based system discussed in this thesis is the large amount of synthesis time required. The system cannot be a viable desktop computation accelerator if it takes an hour to set up the problem in the FPGA, and as FPGAs increase in size, the synthesis time will only increase. Thus, the next important step in this research should be finding ways to reduce or eliminate the synthesis time. One way to do this would be to make the hardware slightly more general so that it can apply to a larger range of problems. Another approach is Run-Time Reconfiguration (RTR) using a system like the JBits system [16] being developed by Xilinx [41]. JBits is a programming interface that provides access to all programmable resources in supported FPGAs. Using RTR, the FPGA could be quickly configured for a new problem by modifying its configuration stream directly without requiring a lengthy synthesis cycle.

A problem with the particular implementation for this research is the large routing delay in the FPGA, which severely limits the maximum attainable clock frequencies, especially for the BOOTH and BIT architectures. The focus of this research was on implementing the four architectures and not on tailoring them to a particular FPGA, so with a better understanding of the FPGA architecture and synthesis tools, it may be possible to reduce or eliminate the excessive routing delays. Alternatively, there are FPGAs available with routing architectures that have more predictable delays, which can increase the synthesis tool's ability to route a design efficiently.

The development platform used for this design is intended to serve as a proof-of-concept, but the usefulness of the FPGA-based system could be better demonstrated if

computed results could be transferred more quickly to the host PC. The dual serial links provide an adequate transfer rate, but they take several seconds to transfer the results of a large number of iterations. Another solution would be to use Ethernet or another protocol through the PCI slots on the Integrator/AP motherboard. This method may require an operating system such as eCos [14], which is freely available and has been ported to the Integrator/AP motherboard.

The concept of an FPGA-based computing system can be extended past the objective of a desktop PC accelerator to a much larger parallel computer. Any number of FPGAs, each with several hundred or more cells, could conceivably be joined to provide massive computational power for cellular parallel problems, although a few issues would need to be addressed. The FPGAs in the array would have to remain synchronized with each other in order to keep the row of cells in lock-step. This would reduce the maximum clock frequency compared to the frequency possible with a single FPGA. The maximum frequency might be further reduced due to the requirement for cells to send their data off-chip. The FPGA array would require a high-speed interconnection network to reduce this negative effect on the clock frequency.

The BOOTH and BIT architectures are actually special cases of a more general digit-serial architecture. A digit-serial architecture uses a D-bit adder, so BOOTH is digit-serial with $D = W$ and BIT is digit-serial with $D = 1$. Such an architecture requires the same complex control as BIT but uses fewer cycles per iteration, and more cells can fit on a chip than for BOOTH because the adders are smaller. It may be possible that a digit-serial architecture with $1 < D < W$ can provide a better calculation rate than both BOOTH and BIT.

Researchers using an FPGA-based cellular computing system cannot be expected to know how to transform their problems for computation on the FPGA. This transformation should be completely automated so that a researcher can formulate a problem in Matlab or Mathematica and transparently receive results computed by the FPGA. Therefore, another step in this research should be to construct an automated process that partitions equations from Matlab or Mathematica into operations suitable for the particular FPGA architecture, inserts those operations into a configuration that can be

programmed on an FPGA, and retrieves the results generated after running the FPGA computer.

Summary

This thesis has presented the design of four cell architectures for FPGA-based cellular computing as well as the application of the design to a real-life heat flow problem. Each architecture was implemented and tested on a development board, and performance results obtained were used to compare cell densities and speedups over PC calculation software.

The four architectures vary in the way they utilize chip area and execution time in order to compute a problem. Two architectures operate at the extremes of the area/time spectrum: one takes a very short time but consumes a large chip area, and the other uses a very small chip area but takes a long time to execute. The other two architectures lie between these two extremes.

The structure of the FPGA computer is a row of computing cells under the control of a central controller. Each cell in the row is connected only to its left and right neighbors and has access to its data from the previous iteration. This structure limits the FPGA computer to certain types of parallel problems like cellular automata and like discretized differential equations with an explicit solution as in the heat flow case study.

The analysis shows that the MULTIPLE architecture provides the highest speedup and that PC calculation software outperforms the current implementation of the BIT architecture. However, with next-generation FPGAs, all four architectures can be expected to accelerate cellular parallel problems with speedups as high as 22,000. Due to the widely varying cell densities, no single architecture is the best choice for every problem: problems that require a large number of cells can benefit from architectures like BIT, and problems that need only a few cells would benefit more from architectures like MULTIPLE.

With additional research and development, an FPGA-based cellular parallel computer could substantially accelerate scientific problems as a PC peripheral or even as a component in a much larger supercomputer.

Appendix
Design Flow Details

The purpose of this appendix is to expand on Chapter 3 by giving the details necessary for implementing the FPGA-based system discussed in this thesis. Also included is some specific information about the VHDL descriptions.

VHDL

Figure A-1 shows the hierarchy of the VHDL source files for this design as well as the files used only for simulation. The `top_test.vhd` testbench simulates the actions taken by the Core Module (CM) including the Logic Module (LM) SSRAM transactions.

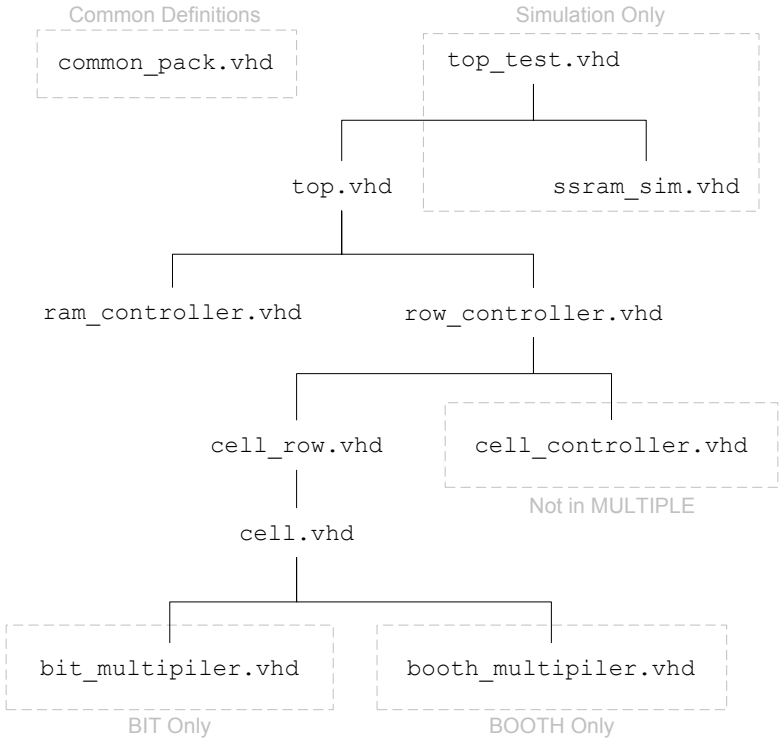

Figure A-1. VHDL source file hierarchy

The VHDL descriptions written for this design adhere to the language standard [19] as well as the synthesis subset of the standard [20]. It is common practice for VHDL designers to include `std_logic_arith` and related arithmetic libraries in their designs. However, these libraries are not the standard and are proprietary to Synopsys. The `numeric_std` library was used instead because it is the standard and its documentation [21] is readily available from IEEE [22].

The source code is completely general in terms of the fixed-point parameters *W* and *S* as well as the number of cells *N*. For this research, *W* is limited to 32-bits by the width of the SSRAM where results are stored, and *S* must be less than *W* and greater than *1*. The number of cells is generalized by using VHDL's `generate` statement, a loop-like construct that allows logic to be replicated any number of times. These design parameters as well as input parameters to the heat flow problem are contained in the definitions file `common_pack.vhd`. While supported by VHDL, the synthesis subset of the language does not allow real numbers in design descriptions. As a result, the scaled integer input parameters to the heat flow problem must be calculated outside the VHDL and manually inserted into `common_pack.vhd`. The definitions file also allows clock frequencies to be assigned to switch settings on the logic module. These switches can then change the frequency at which the computing cells operate.

For the MULTIPLE, SINGLE, and BOOTH architectures, implementing scaled integer arithmetic is trivial because the bit slice shown in Figure 4-5 can be written directly in VHDL. It is more complicated for the BIT architecture, however, because the result of a multiply operation is spread out over time. Thus, result bits must be carefully picked out of the bit stream generated by the Booth multiplier.

Due to the large number of cells that fit in the FPGA for the BIT architecture, some of the cell's shift registers must be implemented in LUTs to keep from exhausting the FPGA's flip-flop supply. The synthesis tools will infer LUT shift registers rather than flip-flop registers if the VHDL is coded in a certain way. According to the Synplify documentation [33], LUT shift registers will be inferred if a register does not have a synchronous reset and if only the register output bit is directly utilized.

Xilinx offers design tips in [42] that suggest registering module outputs so that routing delays need not be affected by paths through combinational logic. This technique

is used for the cell controller outputs and does cause a moderate increase in the maximum clock frequency.

Compilation and Mapping with Synplify

Synplify does not automatically recognize hierarchy in VHDL source files, so the synthesis files must be added to a Synplify project in the order shown in Table A-1.

Table A-1. Source file order for Synplify

	File	Applicable Architecture
1.	common_pack.vhd	All
2.	bit_multiplier.vhd	BIT
3.	booth_multiplier.vhd	BOOTH
4.	cell_controller.vhd	All except MULTIPLE
5.	cell.vhd	All
6.	cell_row.vhd	All
7.	row_controller.vhd	All
8.	ram_controller.vhd	All
9.	top.vhd	All

Figure A-2 shows the project options that must be set. Everything remains as the default except for the part selection, the "Write Vendor Constraint File" checkbox, and the "Top Level Entity" name. The constraint file checkbox is unchecked so that the place and route (PAR) step is free of timing constraints, which will enable Automatic Timespecing as discussed in the next section.

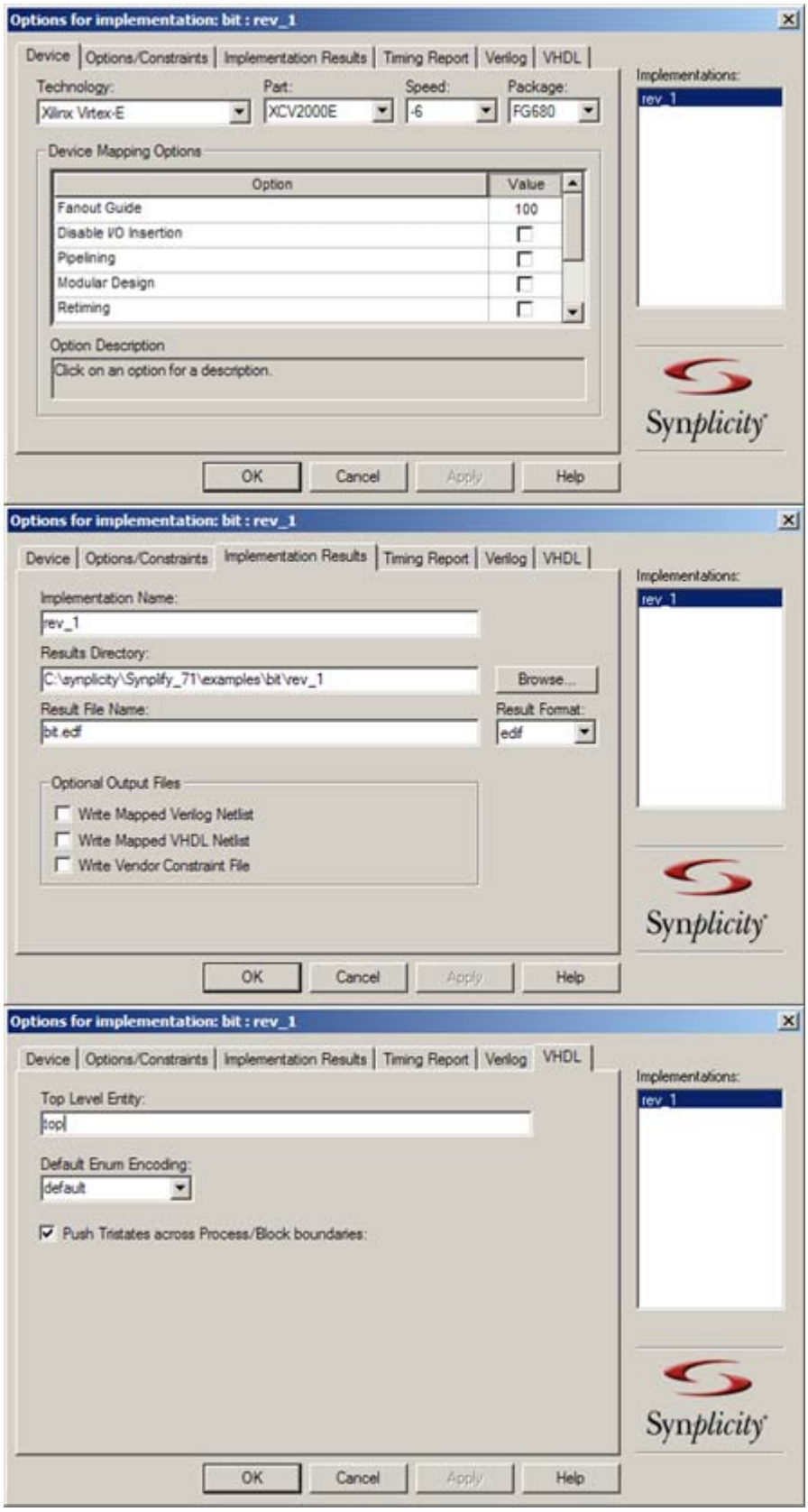

Figure A-2. Project options in Synplify

With the source files in place and the options set, the design should be compiled (but not mapped). A SCOPE constraint file should be entered as shown in Figure A-3. It is important to assign the two clocks to different groups so that the software does not treat them as synchronous clocks. Frequency goals are given for each clock so that the software can insert buffers and replicate logic in order to meet the constraints. The value for HCLK is the Integrator/AP AHB bus frequency, and the value for CLK1 should be slightly above the maximum frequency reported by the ISE PAR software.

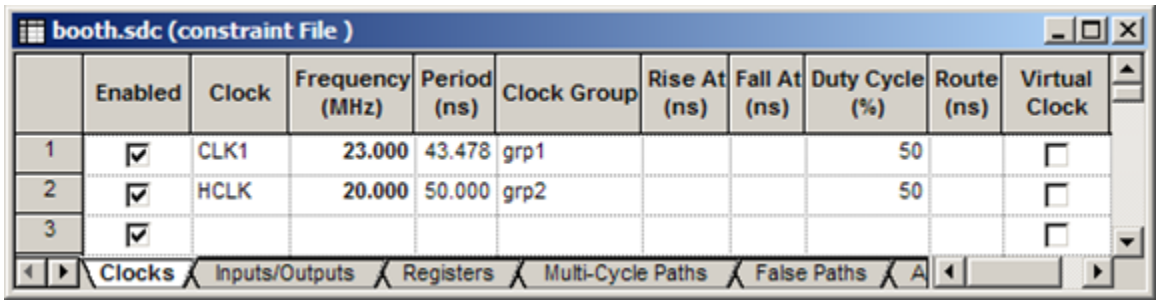

Figure A-3. SCOPE timing constraints

The options dialog box should list the SCOPE constraints file and the checkbox should be checked. At this point, the design can be mapped using the GUI interface or by executing the Tcl command `project -run synthesis`.

After its synthesis stage, Synplify generates a report that includes the FPGA's resource utilization. This information is often useful, so it is shown in Figure A-4 for each cell architecture.

MULTIPLE

```
Cell usage:
FDP              32 uses
FDC              6 uses
GND              12 uses
VCC              11 uses
MUXCY_L          31007 uses
XORCY            30030 uses
MULT_AND         11600 uses
MUXCY            721 uses
FDCE             821 uses
MUXF5            50 uses
MUXF6            25 uses
FDE              99 uses
FDPE             36 uses

I/O primitives:
IBUF             41 uses
IOBUF            65 uses
OBUFT            2 uses
OBUF             62 uses

BUFGP            2 uses

I/O Register bits:                   68
Register bits not including I/Os:    926 (2%)

Global Clock Buffers: 2 of 4 (50%)

Mapping Summary:
Total  LUTs: 33808 (88%)
```

SINGLE

```
Cell usage:
FDP              34 uses
FDC              55 uses
GND              40 uses
MUXCY_L          23226 uses
XORCY            22701 uses
MUXCY            667 uses
MULT_AND         10508 uses
FDRE             925 uses
MUXF5            1153 uses
FDE              1000 uses
FDCE             2852 uses
FDS              55 uses
FDRS             3 uses
FD               44 uses
MUXF6            53 uses
FDPE             38 uses
VCC              1 use

I/O primitives:
IBUF             41 uses
IOBUF            65 uses
OBUFT            2 uses
OBUF             62 uses

BUFGP            2 uses

I/O Register bits:                    68
Register bits not including I/Os:   4938 (12%)

Global Clock Buffers: 2 of 4 (50%)

Mapping Summary:
Total  LUTs: 34849 (90%)
```

BOOTH

```
Cell usage:
FDP              33 uses
FDC              118 uses
GND              2 uses
MULT_AND         1824 uses
MUXCY_L          1851 uses
XORCY            1920 uses
FDR              1940 uses
FD               1907 uses
FDRE             988 uses
FDSE             912 uses
FDCE             5963 uses
FDE              1975 uses
FDS              26 uses
FDRS             41 uses
FDPE             40 uses
MUXF5            506 uses
MUXF6            125 uses
MUXCY            1 use
VCC              1 use

I/O primitives:
IBUF             41 uses
IOBUF            65 uses
OBUFT            2 uses
OBUF             62 uses

BUFGP            2 uses

I/O Register bits:                   68
Register bits not including I/Os:    13875 (36%)

Global Clock Buffers: 2 of 4 (50%)

Mapping Summary:
Total  LUTs: 34692 (90%)
```

BIT

```
Cell usage:
FDP              34 uses
FDC              169 uses
GND              427 uses
VCC              425 uses
FDRS             10186 uses
FDRE             424 uses
FDRSE            424 uses
MUXF5            2590 uses
FDE              2220 uses
FD               502 uses
MUXF6            460 uses
FDCE             11583 uses
FDR              69 uses
FDS              228 uses
FDPE             39 uses
MUXCY_L          27 uses
XORCY            20 uses
MUXCY            1 use

I/O primitives:
IBUF             41 uses
IOBUF            65 uses
OBUFT            2 uses
OBUF             62 uses

BUFGP            2 uses

SRL primitives:
SRL16E           3392 uses

I/O Register bits:                    68
Register bits not including I/Os:  25810 (67%)

Global Clock Buffers: 2 of 4 (50%)

Mapping Summary:
Total  LUTs: 28262 (73%)
```

Figure A-4. FPGA resource usage for each architecture

Place and Route with ISE 5.1

A goal of this research is to run the computing cells at the highest possible clock frequency, but this rate is typically unknown until after the synthesis process. To address this problem, the ISE software provides Automatic Timespecing, which will attempt to place and route the design to achieve a near-optimal clock frequency without requiring multiple synthesis passes. According to the Xilinx documentation [40], Automatic Timespecing is enabled if no timing constraints are found and if the Overall Effort Level is Normal or higher.

The PAR synthesis step starts by adding an EDIF file produced by Synplify to an ISE project. It is very important to also add a constraint file (`pinout.ucf`) that assigns signals to pin locations, because all FPGA pins are predefined by the chip's placement on the LM. Settings in the "Place and Route Properties" box should be entered as shown in Figure A-5. All other settings can remain at their defaults. Double-clicking on "Generate Programming File" in the process window starts a sequence of processes that ends with a bit file (`top.bit`) used for programming the FPGA. The typical command lines of the intermediate steps should be similar to Figure A-6, although they do not have to be entered directly when the ISE GUI is used.

Timing constraints can be used to further increase the maximum clock frequency, but this method only provides a few megahertz improvement and can add several hours to the synthesis time.

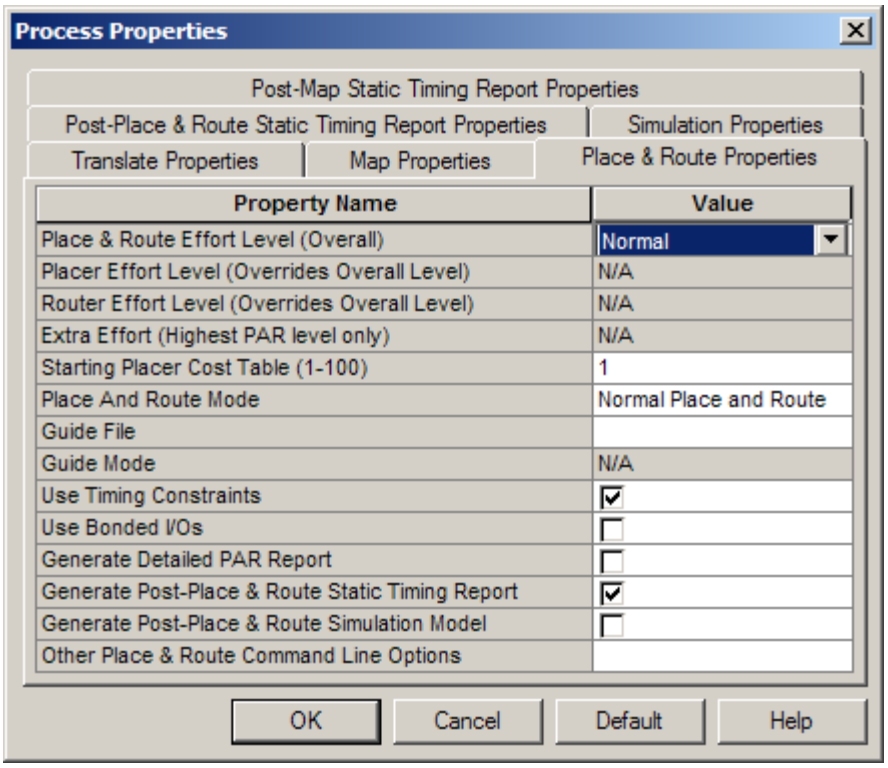

Figure A-5. Project options for Xilinx ISE

```
ngdbuild -quiet -dd _ngo -uc pinout.ucf -p xcv2000e-fg680-6 top.edf top.ngd

edif2ngd –quiet "top.edf" "./_ngo/top.ngo"

map -quiet -p xcv2000e-fg680-6 -cm area -pr b -k 4 -c 100 -tx off -o
    top_map.ncd top.ngd top.pcf

par -w -ol 3 -t 1 -detail top_map.ncd top.ncd top.pcf

trce -quiet -e 3 -l 3 -a -xml top top.ncd -o top.twr top.pcf

bitgen -f top.ut top.ncd
```

Figure A-6. Typical ISE command lines for the synthesis processes

Bit File Download Over Multi-ICE

The following is a list of the steps required for programming the FPGA with the bit file produced by the ISE software. Refer to Figure 3-5 for a diagram of the LM layout.

1. Turn off power to the Integrator/AP motherboard.
2. Connect the Multi-ICE cable between the Multi-ICE connector on the LM and the Multi-ICE unit and connect the Multi-ICE unit's parallel cable to a PC.
3. Fit a jumper to the CONFIG link on the LM.
4. Turn on power to the Integrator/AP motherboard.
5. On the host PC, start the Multi-ICE Server and click "Auto-Configure," which should detect the XCV2000E FPGA. The Multi-ICE Server opens `portmap` in the background and requires the program to stay open.
6. Execute the `progcards` utility with a `.brd` and a `.bit` file in the same directory as the executable. The `.bit` file is the programming file generated by ISE. The `.brd` file contains the actions requested of `progcards` as shown in Figure A-7. If there are multiple `.brd` files in the same directory, `progcards` allows one to be selected by providing a menu. The LM flash can hold up to two FPGA configuration files located at flash addresses `0x000000` and `0x200000`, and one of these addresses must be specified in the `.brd` file. The `.brd` file shown in Figure A-7 is set up to first configure the FPGA to allow the `.bit` file to pass through to the flash memory. It does this through another `.bit` file (available from the LM documentation CD) that must also be in the same directory. The `.brd` file then instructs `progcards` to send the `.bit` file containing the design to flash and then verifies the flash contents.
7. Turn off power to the Integrator/AP motherboard.
8. Remove the CONFIG link jumper and set the mode switches on the LM according to Table A-2. S1[3] must be open to allow the switches to select the flash image and S1[1] selects which flash data will configure the FPGA.
9. On Integrator/AP power-up, the LM configures the FPGA with the selected flash data.

```
[General]
Name = top AHB XCV2000E -> flash (addr 0x200000)
Priority = 1

[ScanChain]
TAPs = 2
TAP0 = XCV2000E
TAP1 = XC9572XL

[Program]
SequenceLength = 3
Step1Method    = Virtex
Step1TAP       = 0
Step1File      = lmxcv600e_72c_xcv2000e_via_reva_build0.bit
Step2Method    = IntelFlash
Step2Address   = 200000
Step2TAP       = 0
Step2File      = top@0x200000.bit
Step3Method    = IntelFlashVerify
Step3Address   = 200000
Step3TAP       = 0
Step3File      = top@0x200000.bit
```

Figure A-7. Example `.brd` file used for downloading configuration files to LM flash

Table A-2. LM switch settings for selecting a flash image

Flash Image	Image Base Address	S1[1]	S1[2]	S1[3]	S1[4]
0	0x000000	Closed[1]	x	Open	x
1	0x200000	Open	x	Open	x
Motherboard selects image		x	x	Closed	x

Core Module Code Generation with ADS and AFS

The program that runs in the CM is the first thing executed when the Integrator/AP motherboard is turned on, so it is responsible for all motherboard initializations. This is done through the ARM Firmware Suite (AFS) μHAL library, which also provides access to the serial ports. The CM program executes as shown in Figure A-8. After initialization, the program enters an infinite loop that receives parameters from the host PC, starts the FPGA calculation, sends FPGA-computed results to the host PC, and then returns to the loop's beginning to start a new calculation cycle.

[1] "Closed" is down and "Open" is up.

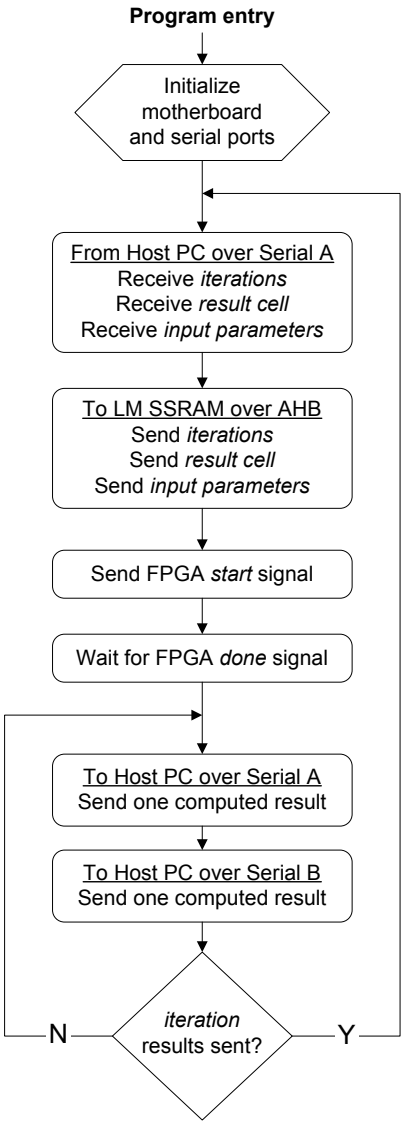

Figure A-8. Core module program flow

The simplest way to generate the CM binary is to create a CodeWarrior project based on an example project from AFS v1.4.1. For this research, the `hello` example was used as a starting point because it uses the µHAL library and performs serial I/O. The following is a list of the steps required to build the CM binary and download it to the Integrator/AP motherboard.

1. Copy `fpga_execute.c`, the source code file for the CM program, to `$AFS/Source/uHALDemos/Sources`[2].

[2] `$AFS` is the directory where the ARM Firmware Suite is installed.

2. Make a copy of `$AFS/Source/uHALDemos/Build/Integrator720T.b/hello.mcp` and rename it to `fpga_execute.mcp`. It should remain in the same directory as `hello.mcp`.

3. The line in `$AFS/Source/uHALDemos/Build/Integrator720T.b/scatter.txt` that reads `LR_1 0x24800000` should be changed to `LR_1 0x24000000`. This allows the CM image to run when the Integrator/AP is turned on.

4. Open `fpga_execute.mcp` in CodeWarrior and change the File, Link Order, and Targets as shown in Figure A-9.

5. The settings for the `standalone` target should be changed as shown in Figure A-10. All other settings can remain unchanged. The post-linker is set to "ARM fromELF" and the output format is set to "Plain binary" because the image is not meant to be executed from an operating system or a debugger.

6. Click "Make" to generate the binary image.

7. Turn off power to the Integrator/AP motherboard.

8. Connect the Multi-ICE cable between the Multi-ICE connector on the CM and the Multi-ICE unit and connect the Multi-ICE unit's parallel cable to a PC.

9. Turn on power to the Integrator/AP motherboard.

10. Start the Multi-ICE Server and click "Auto-Configure," which should detect the ARM720T processor. The Multi-ICE Server opens `portmap` in the background and requires the program to stay open.

11. Open the AXD Debugger.

12. Click the Options→Configure Target… menu item. The first time AXD is run, the Multi-ICE DLL must be made known to the debugger. Click Add and select the file `$MULTIICE/Multi-ICE.dll`[3]. Choose "Multi-ICE" in the "Choose Target" dialog box and click Configure. The software will try to locate the Multi-

[3] `$MULTIICE` is the directory where Multi-ICE is installed.

ICE Server, which can also be running on a remote computer. When the ARM720T processor shows up in the "Device selection" box, click Ok.

13. AXD will connect to the ARM720T processor through Multi-ICE. Click "Flash Download." In the "Image to load" box, enter or browse to `$AFS/Source/uHALDemos/Build/Integrator720T.b/fpga_execute_Data/standalone/fpga.bin` and click Ok. A console window will show the download's progress. Enter '`y`' at both prompts. At this point, the CM image resides in the motherboard flash.
14. Turn off power to the Integrator/AP motherboard.
15. Referring to Figure 3-4, set switch S1[1] to the off position to allow code execution to begin at `0x24000000`.
16. On Integrator/AP power-up, the CM image will begin executing.

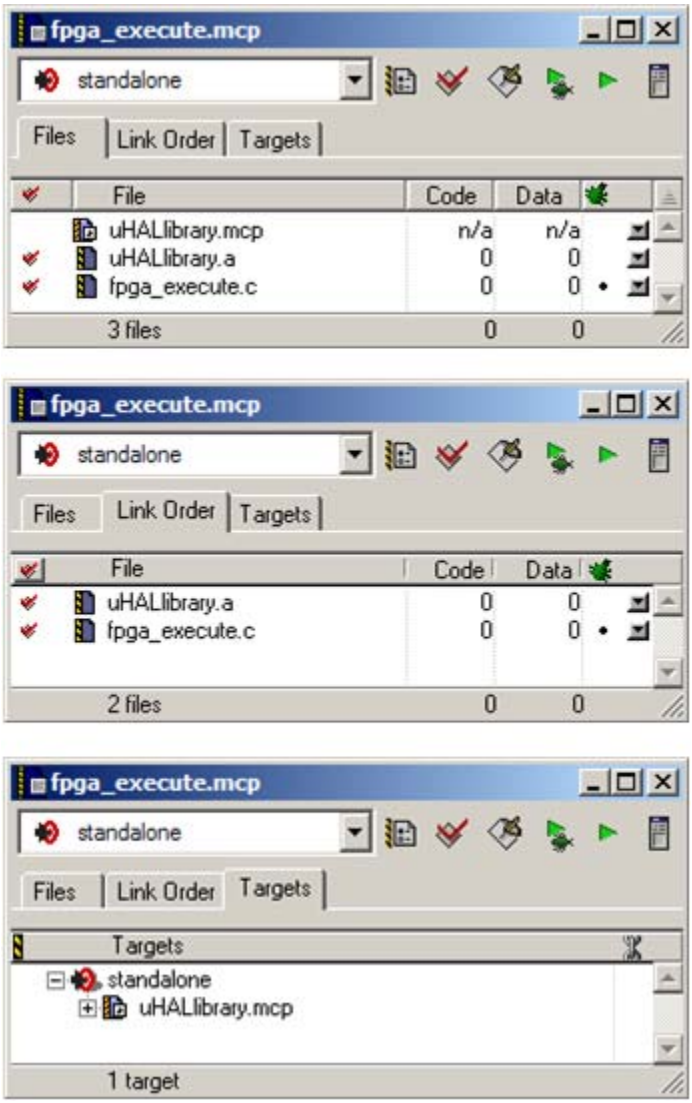

Figure A-9. CodeWarrior file and target settings

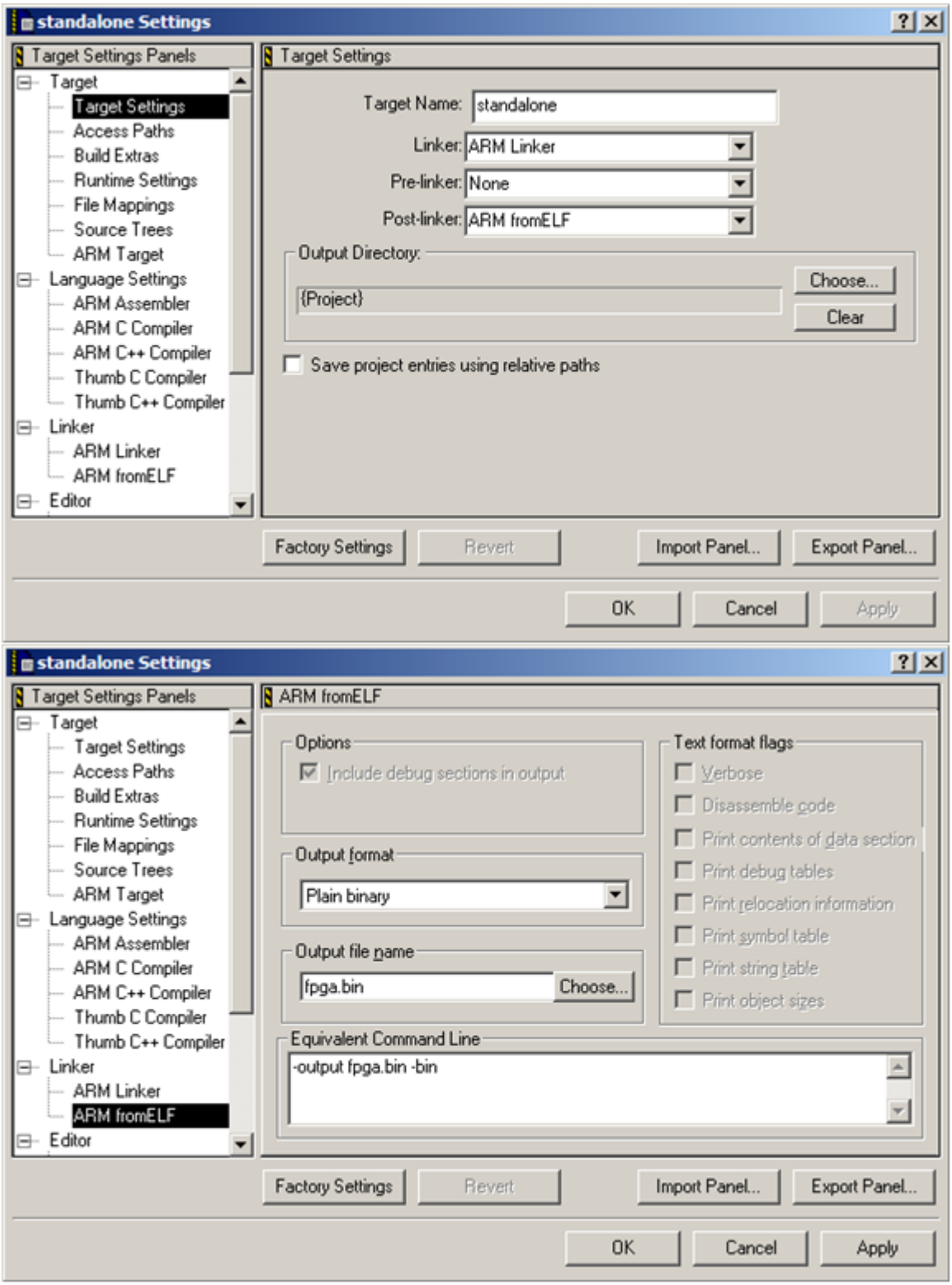

Figure A-10. CodeWarrior project settings

Matlab User Interface

The Matlab user interface to the FPGA system is a simple function that opens two serial ports, sends input parameters to the Integrator/AP over one port, and retrieves FPGA-computed results over both ports. Two serial ports are used to collect the results in order to maximize the data transfer rate. Figure A-11 is an example Matlab session showing the function usage. The example shows the retrieval of 10 time steps of the matrix temperature for cell 1 of 424 with $T_{m,left} = T_{m,right} = \Gamma = 2.6$ using the BIT architecture. The function arguments are the *number of time steps*, the *result cell*, a vector of the *input parameters*, and the *names of two serial ports* over which results will be transferred. The vector of input parameters must be the *left applied temperature*, the *right applied temperature*, and Γ. The first serial port name given must be connected to Serial A on the Integrator/AP motherboard.

```
>> fpga_execute(10,1,[2.6 2.6 2.6],'com6','com7')

ans =

           0
     1349511
     1363009
     1693805
     1703727
     1865965
     1874072
     1973573
     1980528
     2048904
```

Figure A-11. Example usage of the Matlab FPGA interface function

Due to a bug in the serial handling code for Matlab 6.5, a patch must be downloaded from the MathWorks website [23] before the function can be used.

References

[1] ARM Ltd., *AMBA Specification*, IHI 0011A, 1999.

[2] ARM Ltd., *ARM Developer Suite AXD and armsd Debuggers Guide*, DUI 0066D, 2001.

[3] ARM Ltd., *ARM Developer Suite Linker and Utilities Guide*, DUI 0151A, 2001.

[4] ARM Ltd., *ARM Firmware Suite Reference Guide*, DUI 0102G, 2002.

[5] ARM Ltd., *ARM Firmware Suite User Guide*, DUI 0136D, 2002.

[6] ARM Ltd., *Integrator/AP User Guide*, DUI 0098B, 2001.

[7] ARM Ltd., *Integrator/LM-XCV600E+ Integrator/LM-EP20K600E+ User Guide*, DUI0146C, 2001.

[8] J. Armstrong, B. Vick, E. Scott, "Platform Based Physical Response Modeling," *High-Performance Computing Symposium*, 2004, pages 91-100.

[9] J. M. Baker Jr., S. Bennett, M. Bucciero, B. Gold, R. Mahajan, "SCMP: A Single-Chip Message-Passing Parallel Computer," *International Conference on Parallel and Distributed Processing Techniques and Applications*, 2002, pages 1485–1491.

[10] A. D. Booth, "A Signed Binary Multiplication Technique," *Quarterly Journal of Mechanics and Applied Mathematics*, volume 4, 1951, pages 236-240.

[11] S. Brown, J. Rose, "FPGA and CPLD Architectures: A Tutorial," *IEEE Design & Test of Computers*, volume 13, number 2, summer 1996, pages 42-57.

[12] Several postings to the <comp.arch.fpga> newsgroup, <www.fpga-faq.com>.

[13] I. D'Antone, "FPGA Implementation of One-Dimensional and Two-Dimensional Cellular Automata," *Nuclear Instruments & Methods in Physics Research Section A-Accelerators Spectrometers Detectors & Associated Equipment*, volume 430, number 1, June 1999, pages 127-142.

[14] "eCos," 2004, <sources.redhat.com/ecos>.

[15] A. O. Frank, I. A. Twombly, T. J. Barth, J. D. Smith, "Finite Element Methods for Real-Time Haptic Feedback of Soft-Tissue Models in Virtual Reality Simulators," *Proceedings IEEE Virtual Reality 2001*, 2001, pages 257-263.

[16] S. A. Guccione, D. Levi, "XBI: A Java-Based Interface to FPGA Hardware," *Proceedings of the International Society for Optical Engineering*, volume 3526, 1998, pages 97-102.

[17] T. Hartka, *Cellular Automata for Structural Optimization on Recongfigurable Computers*, master's thesis, Virginia Tech, 2004.

[18] IEEE, *IEEE Standard Multivalue Logic System for VHDL Model Interoperability (Std_logic_1164)*, IEEE Std 1164-1993.

[19] IEEE, *IEEE Standard VHDL Language Reference Manual*, IEEE Std 1076-2002.

[20] IEEE, *IEEE Standard for VHDL Register Transfer Level (RTL) Synthesis*, IEEE Std 1076.6-1999.

[21] IEEE, *IEEE Standard VHDL Synthesis Packages*, IEEE Std 1076.3-1997.

[22] IEEE, "IEEE Xplore," 2004, <ieeexplore.ieee.org>.

[23] The MathWorks, Inc., "Technical Solutions," 2004, <www.mathworks.com/support/solutions/data/1-1AQN3.html>.

[24] The MathWorks, Inc., *Using Matlab*, part of the Matlab 6.5 documentation, 2002.

[25] Micron Technology, Inc., *8Mb: 512K x 18, 256K x 32/36 Flow-Through ZBT SRAM*, datasheet, 2002.

[26] M. Miwa, T. Furuhashi, M. Matsuzaki, S. Okuma, "CMAC Modeling Using Bacterial Evolutionary Algorithm (BEA) on Field Programmable Gate Array (FPGA)," *21st Century Technologies and Industrial Opportunities*, volume 1, 2000, pages 644-650.

[27] K. Paar, *A Custom Computing Machine Solution for Simulation of Discretized Domain Physical Systems*, master's thesis, Virginia Tech, 1996.

[28] S. V. Patankar, *Numerical Heat Transfer and Fluid Flow*, Hemisphere Publishing Corporation, 1980.

[29] K. Ramachandran, *Unstructured Finite Element Computations on Configurable Computers*, master's thesis, Virginia Tech, 1998.

[30] R. N. Schneider, L. E. Turner, M. M. Okoniewski, "Application of FPGA Technology to Accelerate the Finite-Difference Time-Domain (FDTD) Method," *Tenth ACM International Symposium on Field-Programmable Gate Arrays*, 2002, pages 97-105.

[31] M. Sipper, "The Emergence of Cellular Computing," *IEEE Computer*, volume 32, issue 7, July 1999, pages 18-26.

[32] Synplicity, Inc., *Synplicity-Xilinx High Density Methodology*, application note, 2000.

[33] Synplicity, Inc., *Synplify Pro Reference Manual*, part of the Synplify Pro 7.1 documentation, April 2002.

[34] Synplicity, Inc., *Synplify Pro User Guide*, part of the Synplify Pro 7.1 documentation, April 2002.

[35] B. Vick, E. P. Scott, "Heat Transfer in a Matrix with Embedded Particles," *Proceedings of the 1998 IMECE*, volume 4, 1998, pages 193-198.

[36] Virginia Tech, "Virginia Tech Terascale Computing Facility," 2004, <www.tcf.vt.edu>.

[37] J. Vuillemin, "Reconfigurable Systems: Past and Next 10 Years," *Vector and Parallel Processing 1998*, 1999, pages 334-354.

[38] R. Wang, "Fast Multiplication—Booth's Algorithm," 2004, <jingwei.eng.hmc.edu/~rwang/e85/lectures/arithmetic_html/node10.html>.

[39] S. Wolfram, *A New Kind of Science*, Wolfram Media, 2002, pages 376-382.

[40] Xilinx, Inc., *Development System Reference Guide—ISE 5*, part of the ISE 5 documentation, 2002.

[41] Xilinx, Inc., "The JBits SDK," 2004, <www.xilinx.com/products/jbits>.

[42] Xilinx, Inc., *Synthesis and Simulation Design Guide*, part of the ISE 5 documentation, 2002.

[43] Xilinx, Inc., "Virtex-4 FPGAs," 2004, <www.xilinx.com/virtex4>.

[44] Xilinx, Inc., *Virtex-E 1.8 V Field Programmable Gate Arrays*, DS022-{1,2,3,4}, datasheet, 2002.

[45] Xilinx Inc., "Xilinx," 2004, <www.xilinx.com>.

Vita

Ken Morgan was born in 1980 and grew up in Waterford, Michigan. After graduating from Waterford Mott High School in 1998, he received college scholarship offers from Rensselaer Polytechnic Institute and Calvin College. He chose to attend Calvin College in Grand Rapids, Michigan and graduated with an Engineering degree in 2002. During his time as an undergraduate, he had summer internships at General Motors and Delphi. After college graduation, he received graduate assistantship and full tuition scholarship offers from Iowa State University and Virginia Tech. This thesis is the final step towards achieving his master's degree in Electrical Engineering from Virginia Tech.