A Custom Computing Machine Solution for Simulation of Discretized Domain Physical Systems

by

Kevin J. Paar

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

APPROVED:

Peter M. Athanas, Chairman

Michael A. Keenan

Charles E. Nunnally

June 5, 1996

Blacksburg, Virginia

Keywords: CCM, FPGA, Simulation, Floating Point, Heat Transfer

A Custom Computing Machine Solution for Simulation of Discretized Domain Physical Systems

by

Kevin J. Paar

Committee Chairman: Peter M. Athanas

Electrical Engineering

(ABSTRACT)

This thesis describes the implementation of a two-dimensional heat transfer simulation system using a Splash-2 Custom Computing Machine (CCM). This application was implemented as a proof of concept for utilizing CCMs in the simulation of physical systems. This paper discusses physical systems simulation and the need for discretizing the domain of such systems, along with the techniques used for mathematical simulation. Also discussed is the nature of CCMs, and why they are well suited to this application. A detailed description of the approach and implementation is included to fully document the design, along with an analysis of the performance of the resulting system.

ACKNOWLEDGEMENTS

I would like to thank my advisor, Peter Athanas, for giving me the opportunity to work on this project, and for having the faith to let me tackle it on my own. I would also like to thank Charles Nunnally and Michael Keenen for serving on my committee and helping with the final changes to my thesis. Mr. Keenan deserves special thanks for his willingness to come on-board at the last minute.

I would like to thank my parents (all of them) and my sister for constant encouragement over the years, and for easing my decision to walk away from a perfectly good job and return to school.

I would like to thank all of my friends, both old and new, without whom school (and life in general) would be much less enjoyable. Specifically I need to give special thanks to my roommates, Henry "G" Green and Brendan "B" O'Connor, and also to Mark "Chewie" Cherbaka who probably should have been a roommate. Together we maintained a collective sanity that just would not have been achievable individually. A very special thank you goes out to Chris "Stan" Inacio who's diligent efforts as system administrator (from several hundred miles away) have made our lives *much* easier.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF FIGURES

LIST OF TABLES

Chapter 1

Introduction

Simulation of physical systems is increasingly relied upon by both designers and researchers in the physical sciences; but simulation of physical systems is computationally expensive. Although recent increases in the performance of general-purpose microprocessors have made physical simulation a reality for more and more users; large, complex simulations are still the realm of expensive supercomputers. Advances in reconfigurable computing technologies have spawned the development of a new genre of computer hardware referred to as *Custom Computing Machines* (CCMs.) CCMs promise to bridge the gap between the accessibility of general-purpose microprocessors and performance of special-purpose supercomputers [13]. This research investigates the use of CCMs for physical system simulation by implementing one type of simulation, called finite-difference heat transfer, on a Splash 2 CCM.

1.1 Physical Simulation

Designers of physical systems are typically involved in an iterative process of design and testing in order to meet their desired performance specifications. Traditionally the testing portion of this cycle has involved building prototype systems and performing physical testing on the prototypes to determine their suitability to the task at hand. The inherent problem with this paradigm of

design is speed. The process of building and testing prototypes can be lengthy and involved [9]. The advent of the computer has given the designer a new tool with which to solve this problem. Computer simulations of physical systems can yield the same information as prototype testing in a fraction of the time and at a fraction of the cost [9].

Researchers of physical systems frequently find themselves investigating parameters of a system which are prohibitively difficult to measure. These difficulties can arise from the dynamics of the system, or from the invasiveness of the measurement apparatus. Imagine trying to measure the stresses on a turbine blade in a running jet engine, or the temperature at the center of a homogeneous solid mass. Again, computer simulation can provide answers to these questions more readily and more efficiently than physical testing.

Unfortunately, simulations can only approximate the physical world, and will never provide exact results. As the user's requirements for accuracy of the physical model increase, so does the burden on the computer [5]; but in order for simulation to maintain its utility, computers must execute the simulations in a timely and accurate manner. Therefore designers and researchers must employ computers of ever increasing power to meet their needs. The responsibility of supplying these high-performance computing systems lies in the hands of the computer engineer.

1.2 Custom Computing Machines

It is an accepted rule-of-thumb that generality and efficiency in computers are inversely proportional to one another [13]. In other words, the wider the range of applications the computer can manage, the less productive it is at any one of them. This isn't surprising, considering that it is

simply a modern extension of the age-old proverb, "jack of all trades, master of none."

At one end of this spectrum lay general-purpose processors which are, as their name implies, designed to accommodate nearly any computational task. From simulation to word processing, the scope of applications which a general-purpose processor can handle is limited only by the software running on it. Unfortunately, this generality has its price. Consider the access rate of main memory, referred to as the *memory bandwidth*. In some processors as much as fifty percent (or more) of the available memory bandwidth can be consumed by instruction fetches, which simply tell the processor *what to do*. This overhead comes as a direct consequence of generality [6].

At the opposite end of the spectrum are processors (or complete systems) designed specifically for one task. Generally referred to as *Application Specific Integrated Circuits*, or ASICs, these components are designed from the ground up to serve a single, unique purpose. This computational tunnel-vision allows the ASICs to achieve the highest possible productivity for their given problems, but their speciality is their Achilles' heel. Once an ASIC is designed, it will be forever bypassed by advances in technology and algorithms, because in order to update an ASIC, it must be discarded and replaced by a newly designed ASIC [13].

Since the world is never as simple as black and white, there is obviously a lot of gray area between these two extremes. Within this gray area fall special-purpose processors, attached processors, vector processors, coprocessors, and even multiprocessors. Each has its own niche of applications in which it excels, but none fulfill *every* computing requirement.

Despite their omnipotent sounding name, supercomputers also fall within this area due to the very nature of their design. Most supercomputers are simply some permutation of the above men-

tioned concepts, taken to the extreme. In fact, the majority of modern supercomputers rely on the concept of *multiprocessing* (i.e. multiple processors running concurrently,) to achieve their remarkable performance gains. Many supercomputers have hundreds or even thousands of individual processors at their disposal. While their performance is impressive on computationally intense array-oriented problems, the great majority of applications will under-utilize the available resources, making these systems grossly inefficient [4].

It is obvious that when designing a new computer system a compromise between generality and performance must be made [13]. Ideally, a technology is needed which can blur this distinction and cover a wider portion of the spectrum between general-purpose processors and ASICs.

This technological need is fulfilled by *Custom Computing Machines* or CCMs. A CCM is a system that has the generality of software programmability and the performance of an ASIC. Although some emerging CCMs utilize systolic computational arrays [22, 25], the vast majority are based on *Field Programmable Gate Arrays* or FPGAs. FPGAs are devices typified by the use of reconfigurable logic resources which allow the FPGAs to be configured as virtual ASICs. It should also be noted that the FPGA configurations are RAM based, and can be changed as necessary [30].

Some FPGAs (and therefore some CCMs) allow the possibility of *run time reconfiguration* or RTR. RTR is the ability to change portions of the FPGA configuration *while the part is in use.* In much the same way that multitasking operating systems can swap tasks in and out of the processor, the RTR capable system can swap hardware configurations in and out of the FPGA [23, 31].

1.3 Research Contributions

This research demonstrates the use of CCMs for physical system simulation by developing a heat-transfer simulation system on a Splash-2 CCM. The application is designed to accommodate two-dimensional heat-transfer problems consisting of conductive, convective, and direct flux modes of heat transfer. The simulation uses a discretized-domain method with a regular mesh, and an explicit, time-stepping, finite-difference technique for solving the differential equations. The CCM hosts a SIMD multiprocessing approach in which individual processors utilize a custom floating-point format to compute solutions for the simulation.

1.4 Thesis Organization

The thesis is organized into five chapters including the introductory chapter. Chapter 2 contains background information necessary to understand the simulation application. This includes discussions of heat-transfer (Section 2.1), simulation (Section 2.2), Custom Computing Machines (Section 2.3), and binary floating-point numbers (Section 2.4).

Chapter 3 contains a discussion of the general approach used to develop the application which includes sections on partitioning the problem (Section 3.1), generalization of the finite-difference equations (Section 3.2), mapping the algorithms to hardware (Section 3.3), and the custom floating-point format (Section 3.4). Chapter 4 contains implementation specific information including the memory organization (Section 4.1), description of the control PE operation (Section 4.2), hardware designs (Sections 4.3, 4.4, 4.5), and VHDL source code organization (Section 4.6). Chapter 5 summarizes the results of the application development process, including descriptions of the system

performance (Section 5.1), discussion of the error analysis (Section 5.2), and final thoughts on the application with possible improvements on the design (Section 5.5).

Chapter 2

Background

The purpose of this chapter is to familiarize the reader with the theories and technologies encompassed in the research application. This includes the application itself (heat transfer analysis), the method (simulation), the platform (the Splash 2 CCM), and the numerical formats necessary (floating point). This information is intended to prepare the reader to more fully understand the approach and implementation of the application.

2.1 Heat Transfer

Heat transfer (or heat) is energy in transit due to a temperature difference[5]. This transfer of energy can come in one of three forms, also known as *modes* of heat transfer. As illustrated in Figure 2.1, the three modes are known as *conduction, convection,* and *radiation* [5].

Each mode allows energy to transfer from an area of higher temperature to an area of lower temperature. This transfer of energy is expressed quantitatively as the *heat rate*, q, measured in Watts (W). The *heat flux*, q'', is the heat transfer rate per unit area *perpendicular* to the direction of transfer. Therefore $q = q'' \times A$ where A is the area of interest.

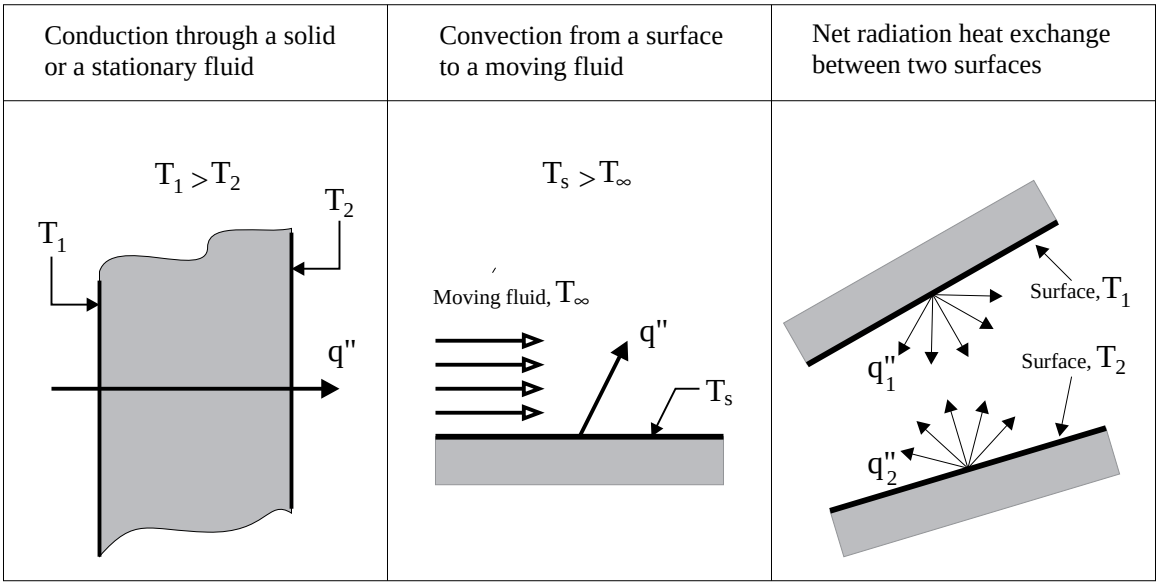

Figure 2.1: Conduction, convection, and radiation heat transfer modes.

2.1.1 Conduction

Conduction is the transfer of heat across a stationary medium. This medium can include either solids or fluids, but in the case of fluids one must be assured that the fluid is stationary. Conduction is the mode of heat transfer affecting a metal spoon placed in a hot cup of coffee. The heat traveling up the length of the spoon is due to conduction.

The heat rate equation associated with conduction is known as *Fourier's law*, and is expressed as

$$q'' = -k\frac{dT}{dx}$$

This equation simply states that the heat rate is proportional to the temperature gradient in the medium. The proportionality constant k is known as the *thermal conductivity* $(W/m \cdot K)$ and is a function of the physical properties of the medium through which the heat is being transfered.

That equation, however, is only for steady state conduction in a single dimension. The transient form of the equation for two dimensional problems is expressed as

$$\frac{1}{\alpha}\frac{\partial T}{\partial t} = \frac{\partial^2 T}{\partial x^2} + \frac{\partial^2 T}{\partial y^2}$$

where T is temperature, t is time, and α, also known as the *thermal diffusivity* (m^2/s), is the proportionality constant. The thermal diffusivity is related to the thermal conductivity, with the addition of two more factors. These are the density of the media ρ, and the specific heat c_p. The formal equation is

$$\alpha = \frac{k}{\rho c_p}.$$

2.1.2 Convection

Convection is the transfer of heat from a stationary surface to a moving fluid. Heat is transfered due to both the conduction between the two media and the bulk motion of the fluid carrying energy away.

This is the mode of heat transfer characterized by the cooling fins placed on some electronic equipment to remove excess heat. The heat is transferred to the surrounding bulk media, which in this case, is the air.

The heat rate equation associated with convection is known as *Newton's law of cooling*, and is expressed as

$$q'' = h(T_s - T_\infty)$$

This equation assumes that heat flows away from the surface media and into the fluid. The proportionality constant h $(W/m^2 \cdot K)$ is referred to as the *convection heat transfer coefficient.*

This one constant encompasses all the parameters which may influence the convective heat transfer including the density and velocity of the fluid and the physical properties of both media.

2.1.3 Radiation

Radiation is the transfer of heat due to the electromagnetic energy emitted by all matter. Given two objects at different temperatures, both will be radiating and absorbing electromagnetic energy. The transfer of heat from the hotter object to the cooler object will be due to the *net* transfer. This is because the cooler object will absorb energy radiated by the hotter object, but will be radiating energy back to the hotter object at the same time. The difference of these two transfers is the effective heat transfer.

This mode can be characterized by the "heat" given off by a campfire. The surrounding air can still be cool, but the fire is definitely giving off energy. This energy is in the form of radiation.

No further elaboration about radiation is necessary at this point because the research application does not simulate radiative heat transfer. If there is further interest, the reader is directed to [5].

2.2 Discretized Domain Simulation

In order to simulate a physical system within a computer, the system must be *modeled*. A model of a physical system is an abstract representation of the system in a form which the computer can store and process. This representation consists of both static data about the state of the system at some time, and dynamic data about how the system changes over time.

Take the problem of heat transfer as an example. Imagine a heat sink with cooling fins attached

to a heat source which is initially turned off. At the instant before the heat source is turned on, the temperature of the heat sink can be measured and recorded. This is the state of the system at that specific time, or the *static data* [9].

When the heat source is turned on, the heat sink will increase in temperature as the heat is conducted away and transfered to the surrounding air. The rate at which the temperature increases is dependent on the heat source, the physical properties of the material the sink is made of, and the temperature and flow of the air around the cooling fins [5]. This information which indicates how the system changes over time is the *dynamic data* [9].

2.2.1 Discretization

As the heat sink is warming up, the temperature at any of an infinite number of points on (or in) the heat sink could be measured. Also the temperature measurement(s) could be made at any of an infinite number of times. This demonstrates the continuous nature of the physical system [9]. Unfortunately computers do not have an infinite data capacity, so the physical system must be *discretized.*

First the physical domain of the problem must be discretized. This is done by subdividing the domain of interest into a finite number of small regions and assigning each region a reference point at its center. This reference point is referred to as the *nodal point* or *node.* Together all the points are called the *nodal network, grid,* or *mesh* [5]. Figure 2.2 shows a basic mesh and a single node within the mesh.

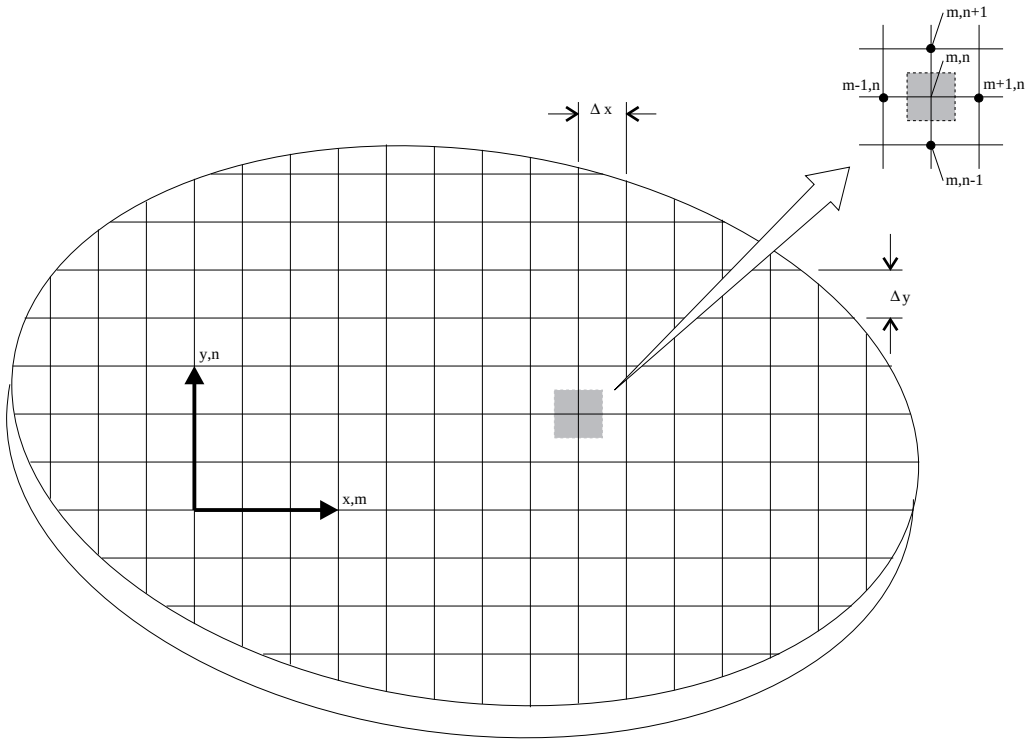

Figure 2.2: Two-dimensional nodal mesh.

2.2.2 Partial Differential Equations

The mesh forms the static data of the model. It is now necessary to form the dynamic model. The dynamics of a physical system are typically represented with partial differential equations (PDEs) which mathematically describe the physical interactions within the system. By solving the PDEs under different circumstances, one can predict how the real system will act over time. This is the principle used in simulation.

There are two general methods for finding solutions to the PDEs. The first method is *analytical*, which usually involves symbolic manipulation of the equations to derive an answer. Although effective, analytical solutions can quickly become cumbersome when the physical systems become complex [5].

The second method used to solve PDEs is *numerical.* Numerical solutions are mathematical approximations which use much simpler equations (than the PDE) to calculate the solution. Simpler equations can be used because the approximations are only calculated for discrete points within the domain of the PDE and not for the entire domain. This is where the nodal mesh comes in.

It should be noted here that the accuracy of the numerical solution is highly dependent on the number of nodal points in the mesh. A mesh with a relatively small number of nodes is called a *coarse mesh* and will have a very limited accuracy. If a large number of nodes is used (a *fine mesh*) a very high accuracy can be achieved. In fact, if the mesh is fine enough, the solution should be indistinguishable from an analytical solution (assuming an analytical solution is even feasible.)

The nodal equations can be derived from the PDEs, but they are still dependent upon the type of modeling used, either *finite difference* (FD) or *finite element* (FE). The FD and FE methods utilize

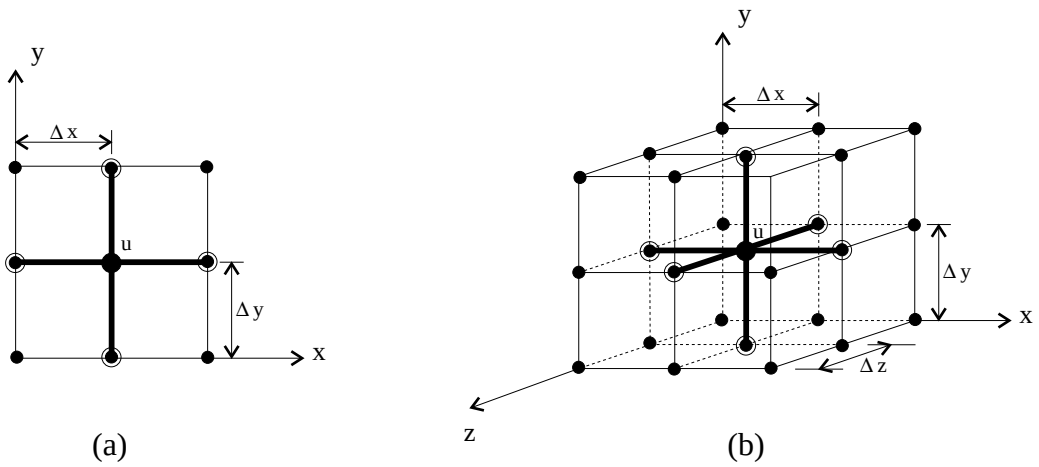

Figure 2.3: (a) Two-dimensional finite difference grid with standard five-point template. (b) Three-dimensional finite difference grid with seven-point template.

different philosophies in their representation of the system, and therefore use different equations. Specifically, the difference between the FD and FE methods lies in their interpretation of the data *between* nodal points. Finite difference methods use Taylor series expansions while finite element methods use piecewise continuous polynomials [3]. The application discussed in this paper utilizes the finite difference method because it is conceptually and mathematically simpler. In fact, the concept behind the finite difference method is simply that the difference in the *value* of the solution between two nodal points must be finite, and can be approximated by simple equations. Thus the name, *finite difference.*

2.2.3 Finite Difference Approximation

The nodal equations used in finite difference approximations are unique to each node. The equation (or equations) at each node are a function of that node and its nearest neighbors. In this sense, one might think of the equations as being a template that moves from node to node. For

two-dimensional problems this template involves five nodes, while in three dimensions a seven node template is used. This is illustrated in Figure 2.3.

Together the equations for each node comprise a set of simultaneous equations. There are many different methods for finding solutions to sets of simultaneous equations, but they all fall into one of two categories. These are *direct* and *iterative* solvers. For the purposes of this research, we are more interested in the iterative methods. Detailed discussions of both classes of solvers can be found in [3].

An iterative solution must start with all nodes at some initial value, and all new nodal values are calculated from there. The new nodal values are calculated "simultaneously" in the sense that the new set of values for all nodes is based solely on the set of values from the previous iteration. There are actually algorithms that take values from iterations other than the previous, but we will limit our discussion to the type known as *Gauss-Seidel iteration.*

The Gauss-Seidel iteration will drive the values at each of the nodes toward some solution. When the values stop changing from one iteration to the next, the solution is said to have *converged.* The solution that is produced is known as a *steady-state solution.*

This type of solution works well for static models, but we want to know how a system changes *over time.* Thus a *transient analysis* is needed. Transient analysis using this type of solver is nearly identical. The biggest difference is in the interpretation of the iterations.

With transient analysis each iteration is actually a time step. The initial values are the state of the system at time $t = 0$ and each iteration is a constant time step, Δt forward in time. Thus the time after the 100th iteration is $t = 100 \cdot \Delta t$. This is referred to as the *forward difference*

approximation. It should be noted here that the time step, Δt, is a discretization of time. This, along with the nodal mesh, completes the discretization of the physical model.

At this point it is time to explore how the finite difference method is used for the analysis of heat transfer.

2.2.4 Finite Difference Form of The Heat Equation

This section will introduce the finite difference equations necessary to implement a simulation engine. In this case the equations are for analysis of two-dimensional, transient heat transfer due to conduction, convection, or both.

Section 2.1.1 introduced the equation for transient heat transfer due to conduction. We must now discretize this equation for space and time. The space discretization is in the form of a mesh as outlined in Section 2.2.1 and shown in Figure 2.2. The time discretization is accomplished as outlined in Section 2.2.3 and is of the form, $t = p \cdot \Delta t$, where p is the iteration number. This yields the following finite difference approximation of the time derivative

$$\left.\frac{\partial T}{\partial t}\right|_{m,n} \approx \frac{T^{p+1}_{m,n} - T^{p}_{m,n}}{\Delta t}$$

Note the use of the iteration number p which denotes the new $(p+1)$, and previous (p) times. If we substitute the above equation into the conduction equation, assume that $\Delta x = \Delta y$ and solve for the new nodal value $T^{p+1}_{m,n}$, the conduction equation becomes,

$$T^{p+1}_{m,n} = Fo(T^{p}_{m+1,n} + T^{p}_{m-1,n} + T^{p}_{m,n+1} + T^{p}_{m,n-1}) + (1 - 4Fo)T^{p}_{m,n}.$$

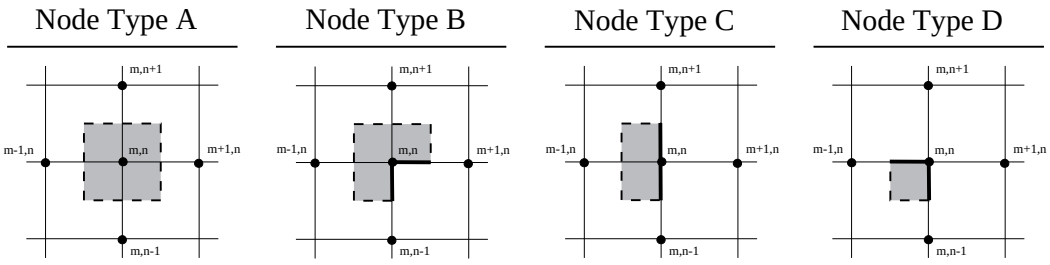

Figure 2.4: The four possible mesh node types.

Here Fo is the finite-difference form of the Fourier number, and is actually

$$Fo = \frac{\alpha \Delta t}{(\Delta x)^2}.$$

The thermal diffusivity, α, was discussed in Section 2.1.1.

For a given simulation, the value of Fo, and likewise $(1-4Fo)$, will remain constant. Therefore the calculation of a new nodal temperature requires only four add operations and two multiply operations. This calculation is simple and straight-forward.

This equation is sufficient if we are only simulating the interior of solid objects, but real-world problems are typically more interesting. For this purpose several different node types must be introduced to allow for boundary conditions. Figure 2.4 illustrates the four nodal geometries. Node Type A, an interior node, was discussed above. Node Types B, C, and D are used to form the boundaries of a region. In Figure 2.4, the boundary edges are designated with solid lines, while internal edges (edges between nodal regions) are designated with dashed lines. The boundary edges of a node may be subject to convective transfer, direct heat flux, or no transfer at all.

Together these four nodal configurations can approximate any two-dimensional geometry needed. As an illustration, the nodal configuration for a simple heat sink with cooling fins is shown in Fig-

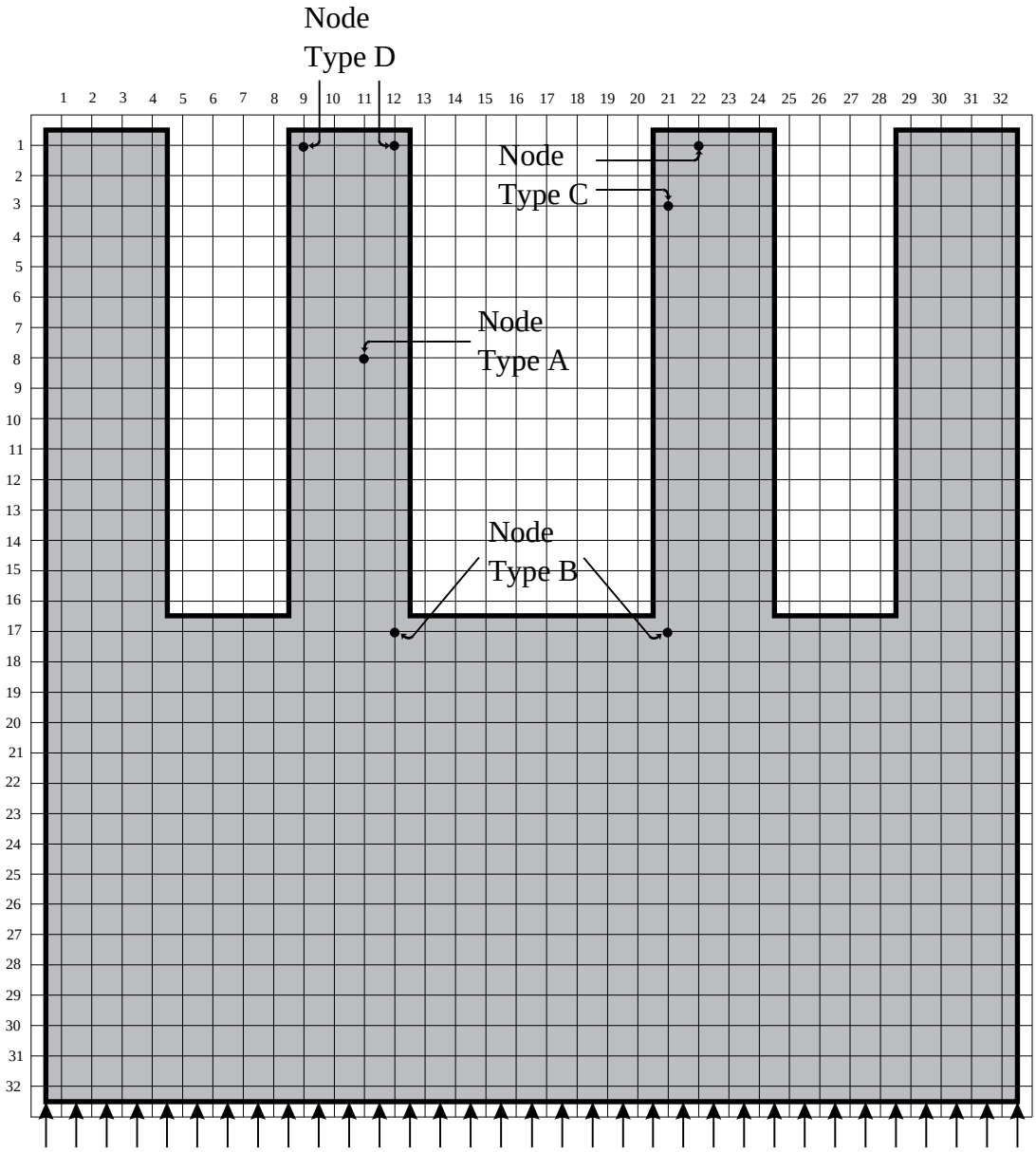

Figure 2.5: An example problem illustrating a heat sink with cooling fins. The different node types are indicated.

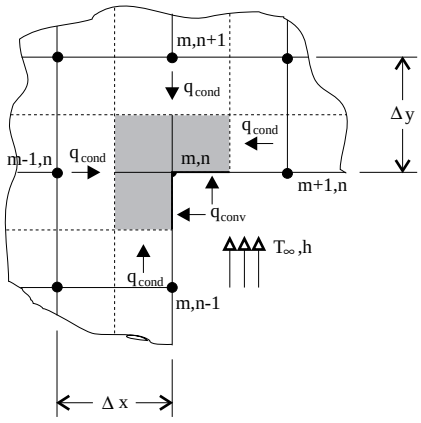

Figure 2.6: Internal corner of a solid with surface convection (Node Type B.)

ure 2.5. Note that the different nodal configurations are pointed out in the mesh. The nodes on the edges of the cooling fins are all subject to convective heat transfer, while the edge at the bottom of the mesh is subject to a direct heat flux, indicated by the small arrows.

Each of the different nodal configurations will require a different equation, and furthermore each different type of heat transfer will require a variant of that equation. To illustrate this we will examine an interior corner with surface convection, or node Type B, as shown in Figure 2.6. Note that all interior edges of the node are subject to conductive transfer from the adjacent nodes while the two exposed surfaces are subject to convective transfer with the fluid. The proper equation for this node is

$$T_{m,n}^{p+1} = \frac{2}{3}Fo(T_{m+1,n}^{p} + 2T_{m-1,n}^{p} + 2T_{m,n+1}^{p} + T_{m,n-1}^{p} + 2BiT_{\infty}) + (1 - 4Fo - \frac{4}{3}BiFo)T_{m,n}^{p}$$

where Fo is the Fourier number discussed previously, and Bi, known as the *Biot number*, is

$$Bi = \frac{h\Delta x}{k}$$

The convection heat transfer coefficient, h, was discussed in Section 2.1.2 and the thermal conductivity, k was discussed in Section 2.1.1.

Although slightly more complex, note that the general form of the equation is similar to the equation for node Type A. The new value is still dependent on the four neighboring nodes, but terms have been introduced to account for the convective transfer. These are the terms involving Bi. In fact, if the exposed edges of the node were *adiabatic*, or insulated, the nodal equation could be derived by setting $Bi = 0$ to yield

$$T_{m,n}^{p+1} = \frac{2}{3}Fo(T_{m+1,n}^{p} + 2T_{m-1,n}^{p} + 2T_{m,n+1}^{p} + T_{m,n-1}^{p}) + (1 - 4Fo)T_{m,n}^{p}.$$

The other possibility for the exposed surfaces is that they experience a direct heat flux like the bottom row of nodes in Figure 2.5. In this case the above equation is used, with the addition of a term for the heat flux. This equation looks like,

$$T_{m,n}^{p+1} = \frac{2}{3}Fo(T_{m+1,n}^{p} + 2T_{m-1,n}^{p} + 2T_{m,n+1}^{p} + T_{m,n-1}^{p} + q''\frac{2\Delta x}{k}) + (1 - 4Fo)T_{m,n}^{p}$$

where q'' is the direct heat flux.

Similar manipulations can be performed for each of the other types of nodes. The nodal equations for convective transfer (except Type A) are summarized in Table 2.1.

The only caveat in this simulation method is *stability*. If all the parameters of the system remain constant during the simulation, the nodal temperatures should converge toward a steady-state solution with increasing time. It is possible however, to introduce numerically induced oscillations which are physically impossible. It is even possible for the oscillations to become *unstable* and diverge from the actual solution.

Table 2.1: Finite difference equations for transient heat analysis.

Note Type	Equation
A	$T^{p+1}_{m,n} = Fo(T^p_{m+1,n} + T^p_{m-1,n} + T^p_{m,n+1} + T^p_{m,n-1}) + (1 - 4Fo)T^p_{m,n}$
B	$T^{p+1}_{m,n} = \frac{2}{3}Fo(T^p_{m+1,n} + 2T^p_{m-1,n} + 2T^p_{m,n+1} + T^p_{m,n-1} + 2BiT_\infty) + (1 - 4Fo - \frac{4}{3}BiFo)T^p_{m,n}$
C	$T^{p+1}_{m,n} = Fo(2T^p_{m-1,n} + T^p_{m,n+1} + T^p_{m,n-1} + 2BiT_\infty) + (1 - 4Fo - 2BiFo)T^p_{m,n}$
D	$T^{p+1}_{m,n} = 2Fo(T^p_{m-1,n} + T^p_{m,n-1} + 2BiT_\infty) + (1 - 4Fo - 4BiFo)T^p_{m,n}$

In order to avoid this situation, a *stability criterion* must be met. In this case the stability is directly affected by the value of Δt, which must be set below a certain limit to avoid oscillations. This limit is dependent on many parameters of the simulation. For problems of interest in this research, *the criterion is determined by requiring that the coefficient associated with the node of interest at the previous time is greater than or equal to zero* [5].

If we examine the equation for node Type A, the coefficient of interest is $(1-4Fo)$. The stability criterion indicates that we must maintain $Fo \leq \frac{1}{4}$, which is accomplished by setting Δt sufficiently small.

2.3 Custom Computing Machines

Section 1.2 introduced the concept of the custom computing machine. It described how the CCM fills much of the gap between general-purpose processors and ASICs. One of the reasons that CCMs can cover such a wide gap is that CCMs encompass a whole class of machines. The common denominator of all CCMs is the use of reconfigurable resources, but how those resources

are utilized *within a system* can vary significantly. The following sections survey some of the major types of CCMs.

2.3.1 Special-Purpose Devices

Special-purpose devices are basically ASICs implemented in FPGAs. Instead of designing the architecture of an ASIC and fabricating it in custom silicon, the design is transferred into an FPGA. The advantages of such a system are:

- Lower fabrication costs. Custom designs implemented in silicon have expensive fabrication set-up costs. If the design is low-volume, these costs can make it economically infeasible. FPGAs are purchased "off the shelf" at a relatively low cost, and customizing them is basically free.

- Inexpensive bug fixes. ASICs are designed by humans, and humans make mistakes. When an ASIC needs to be updated because of an error, the new ASIC will incur the same set-up costs as the original. FPGA based designs can be fixed by simply changing the configuration information.

- Field upgrade-able. Once a system with an FPGA based device is placed in the field, upgrades, either for bug fixes or for capability enhancement, are as simple as upgrading the software.

Although effective for their purpose, this type of CCM does not truly utilize the power of the reconfigurable resources. This is because the design (the configuration of the FPGA) changes infrequently, if at all. Other architectures, which are discussed below, more fully exploit this

capability.

2.3.2 Coprocessors

The term coprocessor usually applies to devices like floating-point coprocessors or graphics coprocessors. These are devices which are tightly coupled to the main, general-purpose processor of a computer system and are designed to off-load some of the computational burden from it. Usually these devices will not only relieve the main processor of some tasks, but will perform these tasks in a fraction of the time, greatly improving overall system performance.

By closely coupling a CCM with a general-purpose processor, the system is given the capability of off-loading the most computationally intensive portion of an application to custom hardware which can *change* from application to application. Regardless of the task, improved performance from hardware acceleration is simply a reconfiguration away.

There are several different platforms in which the main processor and coprocessor are so tightly coupled that the instruction set of the machine can change in response to different computational demands [18, 20]. Instructions not natively understood by the main processor are implemented in the CCM.

One of the more interesting examples of this type of architecture is the CM-2X. This is a Thinking Machines CM-2 supercomputer in which the floating-point coprocessors have been replaced with Xilinx 4005 FPGAs [21]. Although powerful, this type of system places the CCM in a secondary role compared to the main processor. Larger CCM systems can actually stand on their own and reduce the main processor to a supporting role.

2.3.3 Attached Processors

The last architectural class of CCM is the attached processor. In this case the CCM is a complete computing solution which uses a host computer system for configuration, and in some cases I/O. Examples of this type of system include CHAMP [19], the Virtual Computer [15], Splash, and Splash-2 [1], among others. These are larger CCM systems, which employ arrays of interconnected, configurable processors. The basic building block in most of these systems is the *processing element* or PE.

PEs can be single FPGAs, multiple FPGAs, or a combination of FPGAs and other processing hardware. PEs can contain dedicated memory or have access to global memory pools. PEs will also have access to communications resources which in many cases will be reconfigurable themselves.

Along with the PEs and interconnection resources, the attached processor CCM will have an interface with the host system, and usually some additional I/O capabilities for direct connection to other equipment.

The CCM system utilized in this research is an attached processor called Splash-2. The following section will elaborate on the architecture and capabilities of the system.

2.3.4 Splash 2

Splash-2 is an attached processor CCM designed by the Supercomputing Research Center (now called the IDA Center for Computing Sciences) in Bowie, Maryland. Figure 2.7 is an overall diagram of the Splash-2 system [1]. The system is comprised of:

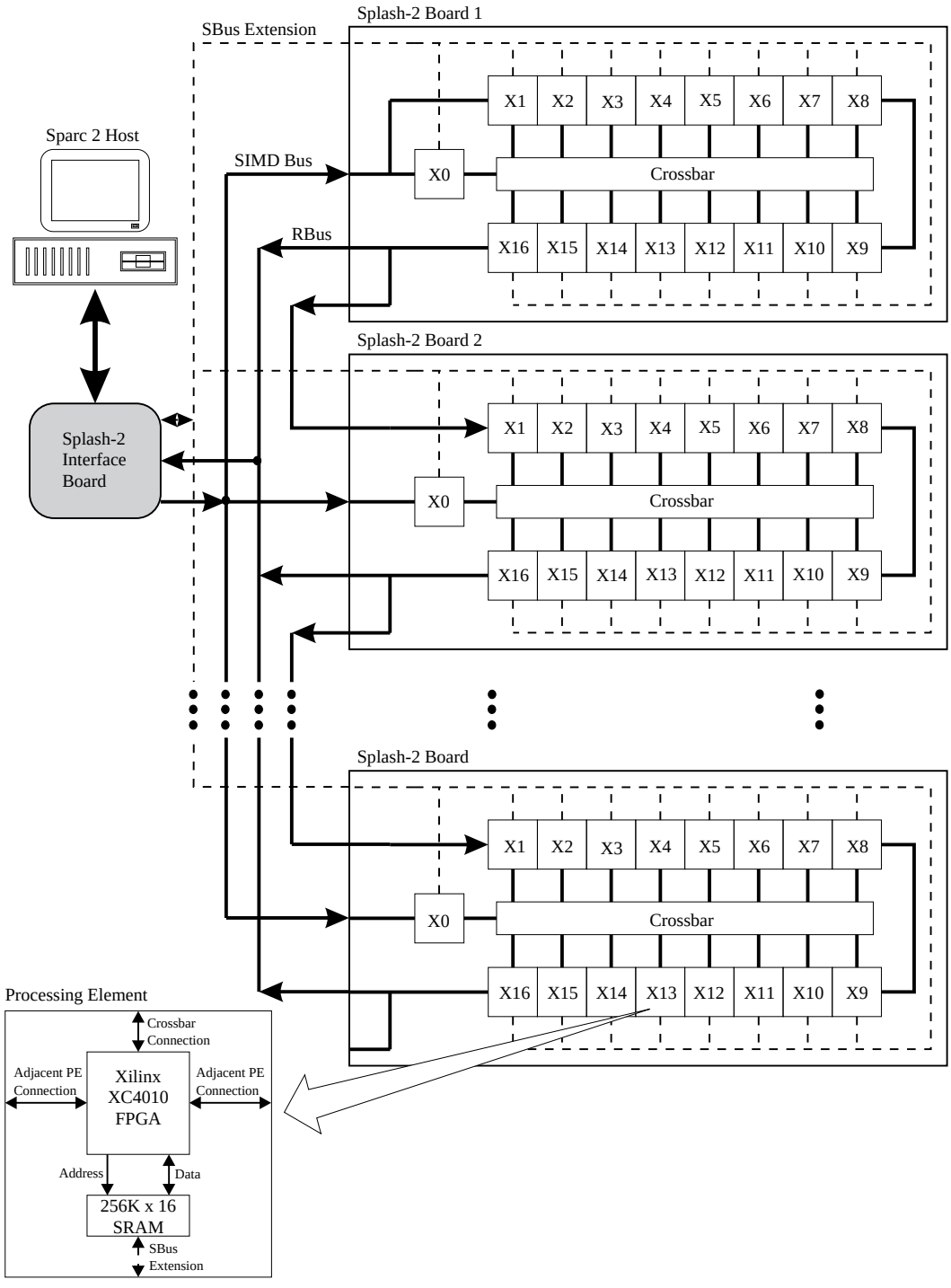

Figure 2.7: The Splash-2 System.

- A Sun Microsystems Sparc 2 workstation. This is the host computer which is responsible for down-loading configuration data into Splash, and performing data I/O as necessary. Development of Splash configurations is also accomplished using this machine.

- A Splash interface board. This board is responsible for the interface between Splash and the Sparc 2 host, and for all data I/O.

- Splash boards. At least one, and as many as sixteen Splash boards may be installed. These boards contain the individual processing elements.

Each Splash board contains sixteen processing elements, each of which is comprised of a Xilinx XC4010 FPGA and a bank of memory. Each PE is connected to its neighboring PEs and to a central crossbar. The crossbar is configurable and has the capability of allowing any PE to communicate with any other PE. A seventeenth PE (X0 in the diagram) is intended as a controlling PE and also has access to the crossbar.

All PE memory banks are connected to an extension of the Sparc 2's SBus which allows the memory to be mapped into the address space of the host. This allows for direct access of the memory in Splash by the host processor.

Programming Splash 2

The Splash 2 CCM is programmed using a combination of VHDL (the *VHSIC Hardware Description Language*) and configurable hardware macros called X-Blox. VHDL is a programming language for describing digital hardware systems. VHDL allows for programming at differing levels

of abstraction from *behavioral* (such as C, a high level programming language) to *structural* (a textual description of a hardware schematic.) The VHDL model of the hardware can be simulated to allow for testing and verification of a configuration without actually loading it into Splash. This is necessary because it is actually possible to damage the Splash system with improper programming.

Once the VHDL model has been verified, it is *synthesized* into a configuration with a synthesis tool. The synthesis tool is responsible for creating hardware constructs which will implement the behavior described in the VHDL code. The less abstract the code is (i.e. the more structural it is) the less work the synthesis tool must do. X-Blox modules are hardware macros that explicitly define hardware components such as adders, counters, registers, and multiplexors in order to relieve some of the burden on the synthesis tool. The X-Blox modules are incorporated into the VHDL code as structural components.

2.4 Binary Floating Point Numbers

The usual interpretation of a digital word is as a base 2 number with one bit set aside to indicate whether or not the number is negative (the sign bit.) But many times a programmer will require the computer to represent *real* numbers which can have a fractional part.

Fractional numbers can easily be represented by assuming that a certain number of bits in the binary word are to the right of the binary point. For example, the 6-bit binary word 110111 can be interpreted as 1101.11 which is 13.75 in decimal. In this case the two least-significant (rightmost) bits are interpreted as being to the right of the binary point. This representation is referred to as *fixed point*.

Figure 2.8: The IEEE 754 floating point format.

The problem with fixed point numbers is that the binary word size is limited and the way the word is interpreted affects the range of the representable numbers. Very small numbers will require more bits to be to the right of the binary point, while large numbers will require more bits to be to the left of the binary point. Ideally a single representation is needed which will handle both situations. This representation is called *floating point.*

In floating point representation the binary word is logically broken into three fields. The first field is the sign bit (S), and indicates whether or not the number is negative. The second field is the *exponent* (E) and the third field is the *mantissa* (M) or *significand.* The number represented is,

$$x = (-1)^s \times m \times 2^e.$$

A standard floating point format is the IEEE 754 standard for 32-bit single-precision numbers [29]. The format of the standard is shown in Figure 2.8. Note that the exponent is 8 bits and the mantissa is 23 bits. This format works basically as described above but with two minor changes. The first is that the exponent is *biased* so it can represent negative exponents. The bias in the IEEE standard is 127. The second change is that there is a binary 1 assumed to be to the left of the binary point, while all 23 bits of the mantissa are assumed to be to the right of the binary point. Therefore the number represented by the IEEE standard is

$$x = (-1)^s \times 1.m \times 2^{e-127}$$

Multiplication of floating point numbers is fairly straight forward in that the mantissas are

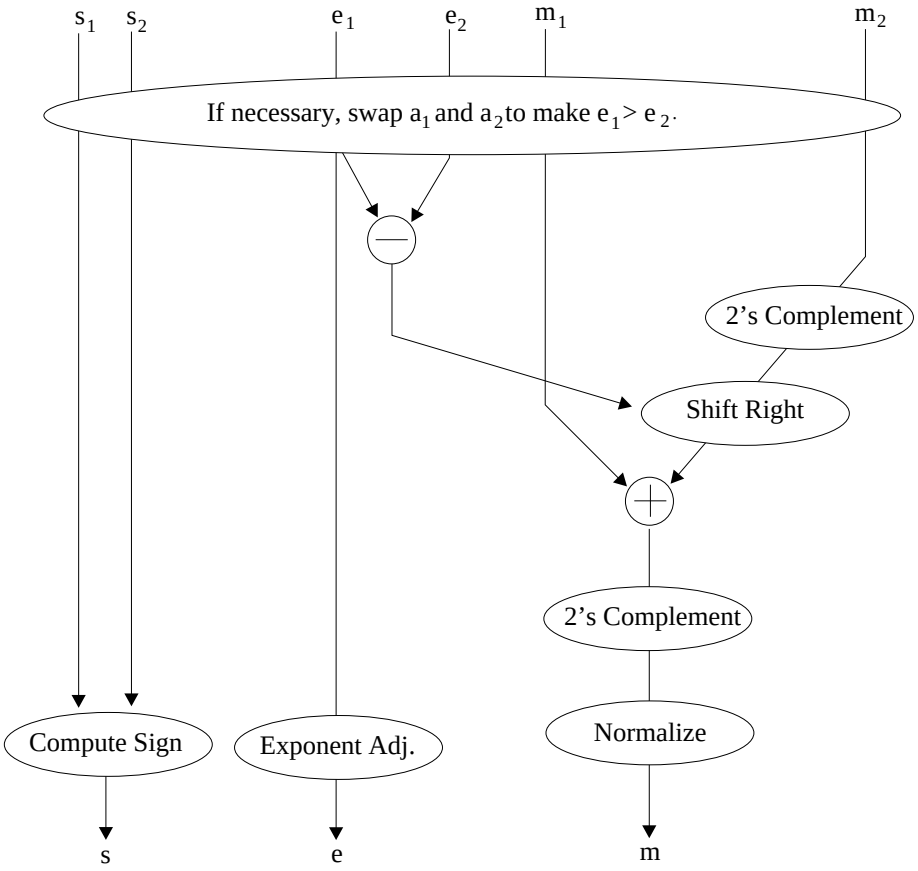

Figure 2.9: Flowchart of IEEE 754 floating point addition.

multiplied, and the exponents are added. Since both exponents have a bias, a bias needs to be subtracted from the sum of the two exponents. Thus, the operation is of the form,

$$(m_1 \times 2^{e_1}) \cdot (m_2 \times 2^{e_2}) = (m_1 \cdot m_2) \times 2^{e_1+e_2-127}.$$

The new mantissa may also need to be *renormalized*, which means that it needs to be shifted to maintain the format $1.xxxxxx\ldots$ where the x's represent the stored portion of the mantissa. Note that if the mantissa is shifted to the left the exponent must be decremented and if the mantissa is shifted to the right the exponent must be incremented.

Addition of floating point numbers is not straightforward. To add the two numbers a_1 and a_2, the following six steps are required: (also refer to Figure 2.9)

1. If $e_1 < e_2$, swap the operands. This ensures that the difference of the exponents satisfies $d = e_1 - e_2 \geq 0$. Tentatively set the exponent of the result to e_1.

2. If the signs of a_1 and a_2 differ, replace m_2 by its two's complement (i.e. $m_2 = -1 \cdot m_2$).

3. Shift m_2 by $d = e_1 - e_2$ places to the right (shifting in 1's if m_2 was complemented in the previous step).

4. Compute the preliminary mantissa $M = m_1 + m_2$ by adding m_1 to the shifted m_2. If the signs of a_1 and a_2 are different, the most significant bit of M is 1, and there is no carry-out, then M is negative. Replace M with its two's complement. This can only happen when d=0.

5. Shift M as follows. If the signs of a_1 and a_2 are the same and there was a carry-out in Step 4, shift M right by 1, filling in the high-order position with 1 (the carry-out). Otherwise shift it left until it is normalized. Adjust the exponent of the result accordingly.

6. Compute the sign of the result. If a_1 and a_2 have the same sign, this is the sign of the result. If a_1 and a_2 have different signs, then the sign of the result depends on which of a_1, a_2 were negative, whether there was a swap in step 1, and whether M was replaced by its two's complement in Step 4.

Floating point numbers are necessary for the research application described in this paper. This means that the hardware constructs for implementing the above procedures must be developed

for FPGAs. Several different implementations of floating point formats have been done before including the 32-bit IEEE standard format and some 16-bit and 18-bit formats. These formats and implementations can be found in [28, 14, 24].

Chapter 3

Approach

The process of mapping an algorithm to a CCM requires the algorithm to be partitioned into hardware mappable constructs. This hardware mapping is dependent on the same conflicting goals of generality and efficiency mentioned in Section 1.2. In this case, the implementation should not be specific to an individual problem, but rather to a limited class of problems. A compromise must be met by defining this class of problems and then developing the hardware necessary to implement the algorithms for them.

The target CCM platform must also be taken into consideration to ensure that the hardware algorithm maps efficiently within the constraints of the logic resources, memory resources, and communications resources. Furthermore, the representation of data within the hardware must be defined. In much the same way as a programmer must define data structures for his program, the CCM application designer must define the data formats on which the custom hardware will operate. These custom data formats must be efficient in their requirements for processing, storage, and communications resources within the CCM.

This chapter will explore the compromises made in mapping the heat transfer problem to the Splash-2 platform. Although these concepts are presented individually, they were developed together in order to achieve the best solution.

3.1 Partitioning the problem onto Splash

The first step in mapping the application onto Splash is deciding on how the algorithms will fit within the confines of the architecture. As discussed in Section 2.3.4 the Splash system consists of several boards, each of which contains 16 processing elements or PEs. Each of the PEs is an available resource which needs to be utilized to achieve the maximum efficiency possible.

Section 2.2.3 discussed how the finite difference technique computes a new set of nodal values from a previous set of values which are already known. This computation involves many individual computations (one for each node) which are completely independent of one another. This independence allows for the possibility of calculating numerous nodal values in parallel. This parallelism can be achieved by assigning each PE in the Splash system a different node to calculate, and then performing the calculations simultaneously.

The process of performing these calculations will require some degree of control to properly sequence each of the individual operations and to organize the overall operation of the system. Since each PE is performing the same operations there is no need to unnecessarily duplicate the control responsibilities. Control of the system is then appropriately given to the control PE, called X0.

PE X0 dictates to the other PEs what to do and when. It accomplishes this by broadcasting control data to the other PEs through the crossbar. This is illustrated in Figure 3.1 which shows a single Splash board, although the Splash system can support multiple boards. Since there is no provision available for broadcasting the control data across multiple boards, each board has its own control PE broadcasting control data across the local crossbar. The multiple control units (one

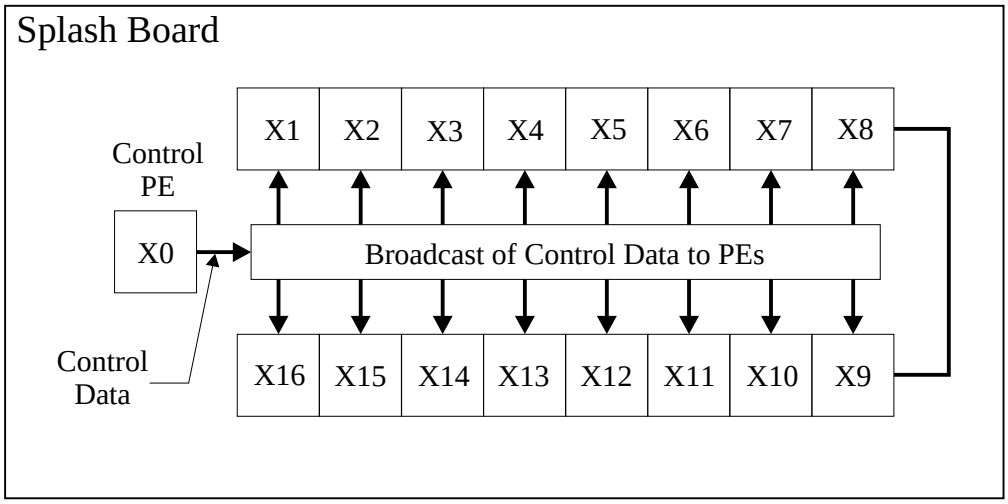

Figure 3.1: Detail of a Splash board showing the broadcast of control data.

for each board) run in lock-step and maintain synchronization between the boards. Because of the centralization of the processing control, this general organization is referred to as a SIMD *(single instruction, multiple data)* multiprocessing architecture.

3.1.1 Mapping the Nodal Mesh to the PEs

As discussed above, each PE is responsible for calculating one node at a time. Each of the nodes in the mesh are mapped to a specific PE for calculation, although these calculations will not necessarily happen simultaneously. This is because there will probably be many more nodes in the mesh than there are PEs in the Splash system. Therefore the PEs will only calculate a subset of the nodes simultaneously, and will iteratively calculate values for however many subsets are necessary to cover all the nodes in the mesh.

Figure 2.5, in Section 2.2.4, introduced an example of a 1024 node mesh organized as a 32×32 array of nodes. Figure 3.2 illustrates how the subsets of this array (or *banks* of nodes) are picked

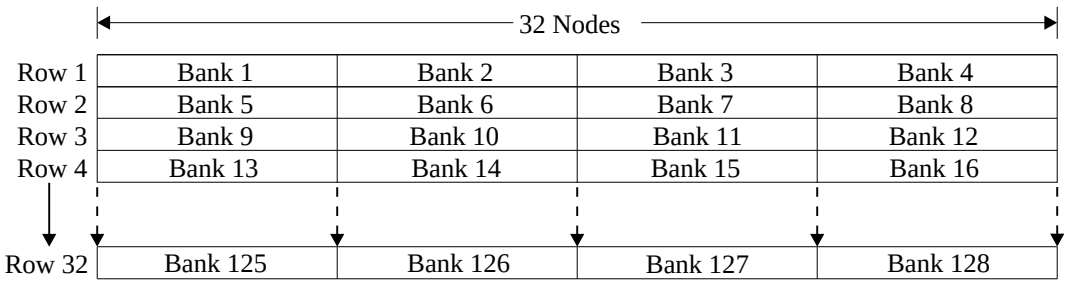

Figure 3.2: Execution order of the banks of nodes.

from the array. This example assumes that there are eight PEs available for processing nodal equations; therefore four banks are required to process one row of the array, and $4 \times 32 = 128$ banks to process the entire array. (The small number of PEs is only for illustrating the concepts, the application can support many more.)

In this example each of the eight PEs is responsible for 128 different nodes (one node from each of the 128 banks.) It is only logical that the information for these nodes should be stored in the memory local to the PE processing them. This accounts for the local node, but a given nodal computation also requires data from the four adjacent nodes to calculate a new value. These values come from the nodes to the left, right, above (previous row), and below (next row). Because of the way the banks of nodes are mapped to the PEs, the above and below nodes are actually mapped to the same PE as the node in question. Therefore these values can simply be read from the local memory and used in the computation.

The left and right nodes are mapped to the adjacent PEs which store these values in their local memories. Instead of maintaining a local copy of these values, the best solution is to get the values from the adjacent PEs. Likewise, these other PEs will need to get the value of the local node

	Bank 1								Bank 2								Bank 3								Bank 4							
Node	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31	32
PE	1	2	3	4	5	6	7	8	8	7	6	5	4	3	2	1	1	2	3	4	5	6	7	8	8	7	6	5	4	3	2	1

Figure 3.3: Mapping of nodes to PEs in successive banks.

for their own nodal computations. Therefore all adjacent PEs swap their data with one another. This data swapping is accomplished through the data busses which pass between adjacent nodes, as shown in the PE detail of Figure 2.7. Furthermore, there are busses which connect the last PE (X16) of one board with the first PE (X1) of the next board allowing the data exchange between PEs to extend across multiple boards. In this manner the PEs form a linear array of processors which can compute a bank of nodes as wide as the length (in PEs) of the linear array.

A problem arises however, when the first and last PEs in the array are considered. These PEs have no adjacent PE on one side with which to swap (or at least get) data. If the linear processing array were always as wide as the mesh this would not be a problem. This is because the first and last nodes on each row will not have any adjacent nodes in the mesh from which to get data (usually they will be a node of Type C or of Type D which do not require data from one of the four directions.)

The real problem arises when the mesh width is greater than the linear array width, which means there are several banks per row. In Figure 3.2, notice that the last node in Bank 1 is adjacent to the first node in Bank 2, and likewise the last node in Bank 2 is adjacent to the first node in Bank 3. Also recall that the banks are processed sequentially, so that Bank 3 is processed after Bank 2 is completed. The data values for these adjacent nodes must be exchanged for the algorithm to work properly, so special consideration must be given to these boundary nodes.

The solution is to reverse the mapping order of nodes to PEs with each bank. This is illustrated in Figure 3.3 which shows a single row of the previous example. Notice that the 32 nodes are broken up into 4 banks, and the banks will be processed sequentially, 1-4. Also notice that the 32 nodes have been mapped onto 8 PEs, and that the order of the mapping reverses with each bank. This is done so that the two nodes which are adjacent across a bank boundary are mapped to the same PE. This way, these boundary nodes can be fetched from the local memory of the first and last PEs, thereby eliminating any problem.

Now that the mapping of all the nodal data is defined, the exact operations necessary to perform the calculations must be defined.

3.2 Generalization of the Finite Difference Heat Equations

Section 2.2.4 introduced the finite difference equations needed to implement the heat transfer analysis application. In order to map the application onto Splash, these equations must be translated into a series of hardware operations which yield the desired result.

Recall from Section 2.2.4 that there were four different equations for the four different node types, and there were variations of these equations depending on the characteristics of the surrounding media (i.e. convective, adiabatic, or constant flux.) The hardware solution must incorporate support for all these variations if the application is to maintain the generality necessary for different problems.

To begin the process of generalization the equations must be examined to identify both the similarities and differences. The following equation is for node Type B, which is the most general

form of any of the equations

$$T_{m,n}^{p+1} = \frac{2}{3}Fo(T_{m+1,n}^{p} + 2T_{m-1,n}^{p} + 2T_{m,n+1}^{p} + T_{m,n-1}^{p} + 2BiT_{\infty}) + (1 - 4Fo - \frac{4}{3}BiFo)T_{m,n}^{p}.$$

There are several important observations that need to be made from this equation. First, the coefficient terms $\frac{2}{3}Fo$ and $(1 - 4Fo - \frac{4}{3}BiFo)$ are based completely on constant values. Therefore these values are constants, and do not need to be recalculated with every nodal iteration. Second, the equation involves all five of the nodal values which may or may not be necessary (depending on the nodal configuration.) Third, the equation contains another term for the convective transfer (or direct flux in another configuration,) which is also based on constant values.

These eight terms can be renamed in the following way

$$\begin{aligned}
T_{m,n}^{p+1} &\longrightarrow NEWLOCAL \\
\frac{2}{3}Fo &\longrightarrow FO \\
T_{m+1,n}^{p} &\longrightarrow RIGHT \\
T_{m-1,n}^{p} &\longrightarrow LEFT \\
T_{m,n+1}^{p} &\longrightarrow TOP \\
T_{m,n-1}^{p} &\longrightarrow BOT \\
2BiT_{\infty} &\longrightarrow CONV \\
(1 - 4Fo - \frac{4}{3}BiFo) &\longrightarrow END \\
T_{m,n}^{p} &\longrightarrow LOCAL
\end{aligned}$$

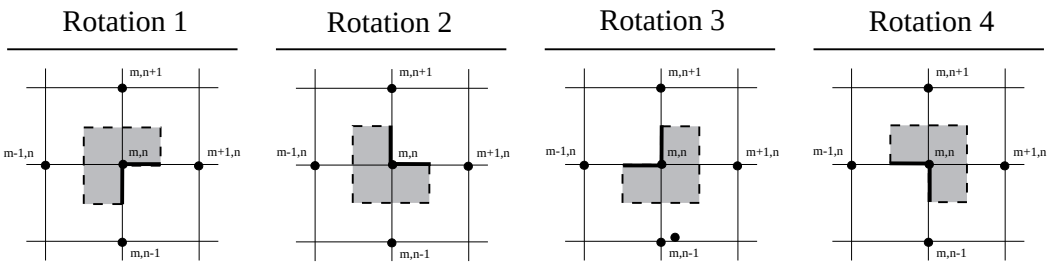

Figure 3.4: The four possible node rotations for node type B.

which translates the equation into a generalized form of

$$NEWLOCAL = FO \cdot (RIGHT + LEFT + TOP + BOT + CONV) + END \cdot LOCAL$$

Although this form of the equation is used to calculate the results for *all* of the equation forms outlined, the assignment of the renamed operands listed above is specific to this form of the equation for node Type B.

3.2.1 Node Specialization

The above equation includes all the necessary elements for supporting any of the equation forms outlined in Section 2.2.4, but differences still exist between the variations. These variations are accounted for by associating configuration data with each node in the nodal mesh, and then using this configuration data to properly map values to the operands of the generalized equation given above. The configuration data indicates what node type is used, what *rotation* the node type is in, and what boundary conditions are in effect (if any.)

The rotation of the node will affect how each of the of the four adjacent nodal values will be incorporated into the equation. The equations given in Table 2.1 are for the rotations shown in Figure 2.4. If the nodes are oriented differently in the mesh, the influences of each adjacent value

Table 3.1: Finite difference equations for transient heat analysis.

Rotation	Equation
1	$T^{p+1}_{m,n} = \frac{2}{3}Fo(T^{p}_{m+1,n} + 2T^{p}_{m-1,n} + 2T^{p}_{m,n+1} + T^{p}_{m,n-1} + 2BiT_{\infty}) + (1 - 4Fo - \frac{4}{3}BiFo)T^{p}_{m,n}$
2	$T^{p+1}_{m,n} = \frac{2}{3}Fo(T^{p}_{m+1,n} + 2T^{p}_{m-1,n} + T^{p}_{m,n+1} + 2T^{p}_{m,n-1} + 2BiT_{\infty}) + (1 - 4Fo - \frac{4}{3}BiFo)T^{p}_{m,n}$
3	$T^{p+1}_{m,n} = \frac{2}{3}Fo(2T^{p}_{m+1,n} + 2T^{p}_{m-1,n} + T^{p}_{m,n+1} + 2T^{p}_{m,n-1} + 2BiT_{\infty}) + (1 - 4Fo - \frac{4}{3}BiFo)T^{p}_{m,n}$
4	$T^{p+1}_{m,n} = \frac{2}{3}Fo(2T^{p}_{m+1,n} + T^{p}_{m-1,n} + 2T^{p}_{m,n+1} + T^{p}_{m,n-1} + 2BiT_{\infty}) + (1 - 4Fo - \frac{4}{3}BiFo)T^{p}_{m,n}$

will change. This is illustrated in Figure 3.4 which shows the four rotations of node Type B, and Table 3.1 which shows the respective equations.

Notice that in each of the equations two of the four adjacent nodal values are multiplied by a factor of two. The difference in the four rotations is *which* two values are multiplied by two. This works in the same way with the other two node types (node Type A obviously has no rotation) but with one difference. Node Type C and Type D have adjacent node values which are not factored into the equation at all. Again, which node values are not included depends on the rotation of the node.

Therefore each of the four inputs to the generalized equation (RIGHT, LEFT, TOP, and BOT) are actually either $0\times$, $1\times$, or $2\times$ the adjacent nodal value. This information is stored in the nodal configuration data so these inputs can be *preprocessed*, thereby allowing the one equation to handle any variation of nodal type and rotation without modification.

The FO, CONV, and END values can also differ depending on which equation variation is used, but they are all constant values within the scope of the problem. If all the necessary constants are

stored together in a table, the correct constant for each situation can be accessed through an index. These indices (one for each of the three constant types) are also stored in the nodal configuration data. Also note that the CONV constant is not used in some variations of the equation. In this case, the index will point to a zero value stored in the table.

In total then, there are seven items of information in the specification for a given node in the mesh. These are the preprocessing values (0×, 1×, or 2×) for each of the nodal inputs (RIGHT, LEFT, TOP, and BOT) and one index for each of the constant values (FO, CONV, and END.)

3.3 Mapping the Algorithm to the PE

In the previous section the equation was developed which will be mapped into hardware operations. This equation is of the form

$$NEWLOCAL = FO \cdot (RIGHT + LEFT + TOP + BOT + CONV) + END \cdot LOCAL$$

with the one result, NEWLOCAL, being dependent upon eight input values. These input values require a total of five addition operations and two multiplication operations, to compute the result.

Ideally, to achieve the maximum performance from the hardware, the seven operations should be performed simultaneously by utilizing seven hardware units. Unfortunately, there are dependencies within the equation which prevent some of the operations from starting until the previous operations are complete. These dependencies are illustrated in the form of a data-flow graph, as shown in Figure 3.5. This graph shows the order in which each of the operations must be performed. Notice that, at best, three operations (two adds and one multiply) can be performed simultaneously, and the rest of the time only one operation can be performed. Because of the low parallelism within

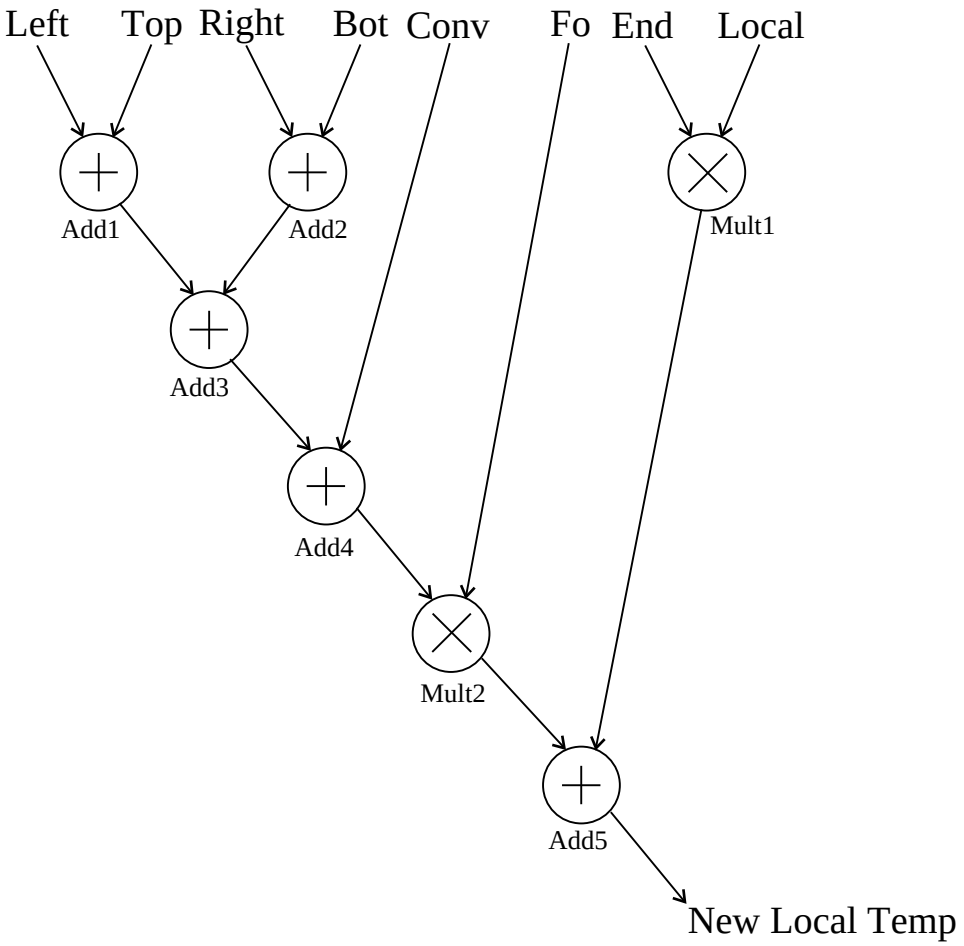

Figure 3.5: A data-flow diagram of the nodal computations.

the nodal computation, only two hardware units are actually placed into the PE. These are the adder and the multiplier.

The source of the data must also be considered to maximize performance. Section 3.1.1 discussed the mapping of nodes to PEs. Recall that the data from the left and right nodes will be exchanged through data busses, but all the remaining values will be stored in local memory. In fact, if the first or last PEs in the array are considered, either the left or right values will also be read in from local memory. Therefore it is possible that seven of the eight data values necessary for the computation must be read from local memory. The nodal configuration data must also be read from local memory, and when the calculation is complete, the result must be written back to local memory. In total there are nine memory operations which must take place for the computation of one new nodal value.

The Splash PE design allows for one memory operation per clock cycle. This means, as a minimum, nine clock cycles will be required per nodal computation (to read in the eight values and write out the result.) The Splash design also requires one idle memory cycle between a read operation and a write operation, so a total of ten clock cycles are required just for memory operations.

In this design the memory operations limit the performance of the computation. The math operations can easily be performed within the ten clock cycles needed for the memory operations, but only if the computations are overlapped. Several values must be read from memory before the mathematical operations can begin for any given node. If the previous node is completing its math operations while these reads are taking place, and then waits until the new node's operations are running to write its new value to memory, no cycles are wasted waiting for data. This is illustrated

Operation	Cycle Number															
	8	9	10	1	2	3	4	5	6	7	8	9	10	1	2	3
Add1								■	■							
Add2									■	■						
Add3	■									■	■					
Add4		■	■									■	■			
Add5						■	■									■
Mult1			■	■									■	■		
Mult2				■	■									■	■	
Memory Operation	Write NEWLOCAL	Read END	Read FO	Read LOCAL	Read Config Data	Read BOUND*	Read TOP	Read BOT	Read CONV	(idle)	Write NEWLOCAL	Read END	Read FO	Read LOCAL	Read Config Data	Read BOUND*

*The BOUND value is only needed for the first and last PEs in the array.

Figure 3.6: The schedule of the necessary operations.

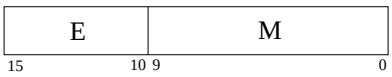

Figure 3.7: The 16 bit custom floating point format.

in the schedule of operations shown in Figure 3.6.

Note from Figure 3.6, that all the math operations require two clock cycles. This is because they are pipelined into two stages. If the input data values are presented to the units on one cycle, the results will be available two cycles later. Creating two-stage pipelines for both the adder and multiplier was determined to be the best solution for completing the math operations within the ten clock cycles required by the memory operations, and still allow for the proper overlap between nodal computations.

3.4 Custom Floating Point Format

Due to the wide range of values which will need to be represented in the application, a floating point format is needed. The standard IEEE 754 format discussed in Section 2.4 is a 32-bit floating point format, while the memory connected to each PE in the Splash system uses a 16-bit word size. Therefore, in order to load or store one IEEE standard floating point value, two memory operations would be required. Since memory operations already limit the scheduling of operations within each PE, using this data format would only make matters worse.

Instead each read or write of a data item must require only one memory operation. This means that the floating point format can only have 16 bits, instead of 32. This reduction in format size requires that a custom format be devised for the application.

Section 2.4 outlined the numerous steps necessary to complete a floating-point addition. The complexity of these operations can be significantly reduced if the numbers are always positive. This is not an unreasonable expectation; because in the heat transfer application, all the nodal values (temperatures) are in degrees Kelvin. It is physically impossible to have a negative temperature in degrees Kelvin because this would be a temperature *below* absolute zero. Also, the constant coefficients used in the nodal computations *must* be greater than zero to maintain stability of the model, as discussed in Section 2.2.4. This one concession in the floating point format saves one bit (the sign bit,) but the sizes of both the exponent and mantissa fields must still be reduced to fit into a 16 bit word.

The size of the exponent field determines the range of numbers that the format can represent. This width can be reduced slightly, to six bits from eight, without losing any of the range necessary for the application. The remaining reduction in bit usage comes completely from the mantissa. The reduction of the mantissa field to 10 bits from 23 allows the format to fit into 16 bits, but greatly reduces the precision of the number representation. This is an unfortunate, but necessary, consequence of the limited data size. The custom format is then 6 bits of exponent and 10 bits of mantissa to form a 16 bit word. This is illustrated in Figure 3.7.

There is another important distinction between the IEEE format and the custom format. Section 2.4 discussed how the IEEE format uses an implied 1 in front of the binary point in the representation of the mantissa. The custom floating point format actually stores this 1 in the mantissa field. This concession further reduces the precision of the format, but was done in order to significantly reduce the complexity of the necessary hardware units. IEEE format hardware units

must check a number's representation to determine if the number is zero during the "unpacking" of the value. This is because the implied 1 before the binary point only exists *if the number is nonzero,* otherwise it is an implied 0. The custom format stores this bit so no checking needs to be done, making the unpacking process simpler.

The IEEE standard also contains provisions for many features that are not implemented in the custom format due to the necessary hardware associated with them. These include representations for "special" cases such as infinite values, values which are not-a-number (NaN), and gradual underflow [6]. The IEEE standard also includes specific guidelines for rounding the results of computations. Again, these features are not implemented in the custom format.

The following is a full description of the steps necessary to perform an addition using the custom floating point format. This is in contrast to the description for the IEEE format given in Section 2.4.

1. If $e_1 < e_2$, swap the operands. This ensures that the difference of the exponents satisfies $d = e_1 - e_2 \geq 0$. Tentatively set the exponent of the result to e_1.

2. Shift m_2 by $d = e_1 - e_2$ places to the right, shifting in 0's on the left.

3. Compute the preliminary mantissa $M = m_1 + m_2$ by adding m_1 to the shifted m_2.

4. If there was a carry out from the addition operation, shift the result to the right by 1 bit, shifting the carry bit into the left. Increment the exponent by 1.

This operation is significantly less complicated than the one outlined for the IEEE format. This is primarily due to the elimination of the sign bit in the custom format.

Multiplication is not very complex in the IEEE format, but it is still slightly simpler in the custom format. IEEE standard multiplication may require shifting the resulting mantissa by several bits in either direction to renormalize it. The custom format can only require a single bit shift to right, which may not even be necessary. Again, this reduction in complexity is due to the unsigned operands.

Chapter 4

Implementation

The previous chapter outlined the high level specification for the system design, along with some of the rationale behind it. This chapter describes some of the specifics of the final implementation in an effort to fully document the design.

4.1 PE Memory Organization

Section 3.3 outlined each of the values which must be read in from memory but did not mention from *where* in memory. The memory in each PE is organized into 4 sections of equal size (64K words each for a total of 256K words). The first two sections are for holding the nodal data (temperatures). Two banks are needed because of the iterative nature of the finite difference algorithm. Recall from Section 2.2.3 that the computations for one iteration, or time step, are based completely on the previous iteration. Therefore the data from the previous iteration must be held until all of the new nodal values have been computed. This is accomplished by running iterations back and forth between these two regions of memory. On one iteration, the nodal data will be read from the first region while the new values are being written to the second region. The next iteration will then read all its values from the second region and write its new values to the first region. This way the data from the two iterations is kept isolated. It should be noted that the relative position of a

given node's data within each region is the same.

The third region contains the configuration data for the nodes stored in the first two regions (see Section 3.2.1.) Again the configuration data for a given node is stored at the same relative position within the region as with the first two regions. This way the same offset into memory can be used to access both temperature data values and the configuration data. In order to maintain this relative position, the configuration data must be the same size as the temperature data, which is one 16-bit word.

The last region in memory is used for the tables of constant data values used in the nodal computations. Because of the limited size of the configuration word, a very limited number of constants can be accessed in this table. Therefore this region of memory is mostly unused.

Section 3.1.1 gave a detailed description of how the nodes are distributed amongst the PEs in the linear array, but little mention was given to the actual organization of this data within each PE's memory. Figure 4.1 gives a detailed example based on the examples started in Section 3.1.1. The figure shows the memory space of each of the eight PEs as columns, with ascending addresses going down through the table. The values shown in each memory position are actually node numbers which correspond to the nodes shown in Figure 3.3. Two whole rows, and the first bank of a third row, are shown. Several important concepts can be explained here.

An example would be the memory for PE 4, at memory location 8 which is in Bank 5. If this is the node currently being computed by PE number 4, the TOP and BOT values must also be fetched from memory along with Node 5's current value. The TOP value is from Node 5 of the *previous* row, in this case row 1. This value is stored at location 4. Likewise the BOT value is

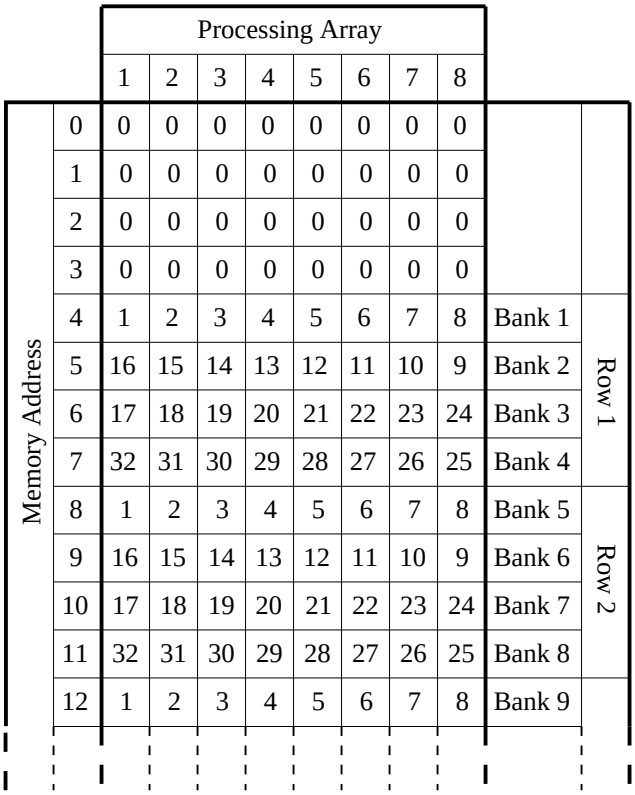

Figure 4.1: Organization of banks of nodes in PE memory.

from the *next* row, in this case Row 3. This value is stored at location 12. The difference in the addresses between a given node and its TOP and BOT values is dependent upon how many banks there are in one node. In this example there are 4 banks per row, and therefore the addresses are plus 4 and minus 4 from the current nodal address. If an entire row of the mesh fit into one bank, the TOP and BOT values would be in the memory locations directly adjacent to the current node.

Another example would be PE number 1, at memory location 5. PE 1 is at the end of the linear array and must account for nodal adjacencies across the bank boundaries. In this case Node 16 is the current node, and the adjacent node whose data must be fetched is Node 17. Because of the way the nodes were alternately mapped into the PEs, Node 17 is adjacent to Node 16 at memory location 6. Therefore the processor simply needs to fetch the data from the next memory location. On the next cycle Node 17 will be the current node, and will need to fetch the data for Node 16. In this case it will fetch the data from the previous memory location. The operation of the last PE in the array is very similar.

4.2 Control PE Operation

The control PE is responsible for directing the operation of all the other PEs in the array. As discussed in Section 3.1, the control PE broadcasts control data to all the other PEs. This control data is comprised of 18 control signals which direct the internal operation of the PE, and 18 bits of address data which is used to access the memory of each PE.

The control PE is simply a state machine which sequences through the ten steps necessary for each nodal cycle. For each step, control data and address data are sent to the other PEs to direct

their operation.

Each of the PEs operates on its own node during a processing cycle. The interesting part is that the addresses used in each PEs memory operations are the same. Referring to Figure 4.1, notice that a bank of nodes is at the same address in each PE. This allows the use of the same addresses for all PEs, even though each PE will be fetching different data. This way, the control PE can properly sequence up through each bank which needs to be processed, and generate the correct addresses for the current node, TOP node, and BOT node. The control PE also takes care of properly alternating the reads and writes between the two regions of temperature data in PE memory.

4.3 PE Hardware Organization

Figure 4.2 shows the basic layout of the PE data handling units. There are several things that should be noticed about this design. There are two sets of registers whose outputs are labeled A and B. These registers issue data to the two computational units which are not shown. Both units (the adder and the multiplier) are issued the same data, therefore only one can be used at a time.

Data is routed to the two register sets by multiplexors which select between several different sources. The data from the multiplexors is split up into the exponent part and the significand (mantissa) part before being registered. But first, the exponent part passes through a conditional incrementer. This incrementer is used to increment the exponent of the incoming data to effectively multiply the value by 2. This is required for preprocessing the nodal input values as discussed in Section 3.2.1. The increment is dependent upon information stored in the configuration data word.

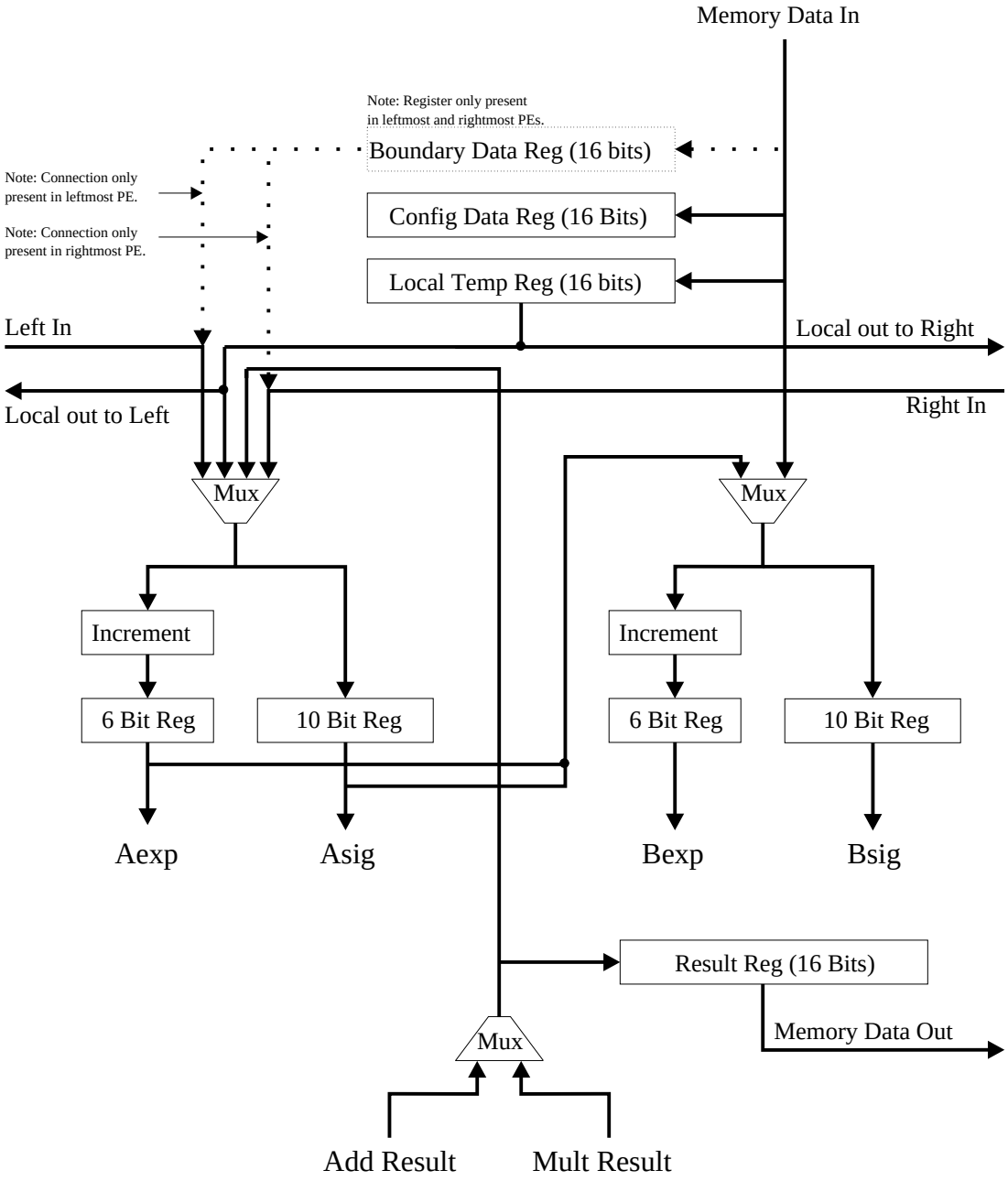

Figure 4.2: Block diagram of PE data routing and conditioning.

Recall from Section 3.2.1 that the preprocessing can require 0×, 1×, or 2× adjustment for any of the four nodal input values. The incrementer handles the 2× situation, and the 0× situation is handled by synchronously clearing the registers instead of loading the incoming data. This is also dependent on the information stored in the configuration data word. The 1× situation requires no special handling.

The results from the two computational units are routed through a multiplexor which feeds back into the input registers and into the result register. The output of the result register is sent to memory to be written out.

4.4 Adder Design

The adder design is shown in Figure 4.3. The add operation starts with the two exponents being fed into two subtracters. These subtracters calculate $e_1 - e_2$ and $e_2 - e_1$ simultaneously. One of these two values must be positive, and the signal which indicates this is used by the multiplexors to select the proper shift value, the tentative exponent value, and the swapping of significands. The outputs of the multiplexors are registered to complete the first cycle of the pipeline. On the next cycle one of the significands is shifted left by the number of bit positions calculated by the subtracters. This shifted significand and the other significand are added together in the adder and the result is then passed to another shifter which is conditional. If there was a carry out from the addition, this shift needs to be performed, and the exponent value needs to be incremented to compensate for the shift. This is accomplished by the incrementer. The final result comes out of the incrementer (for the exponent) and the shifter (for the significand.) Refer to Section 3.4 for a

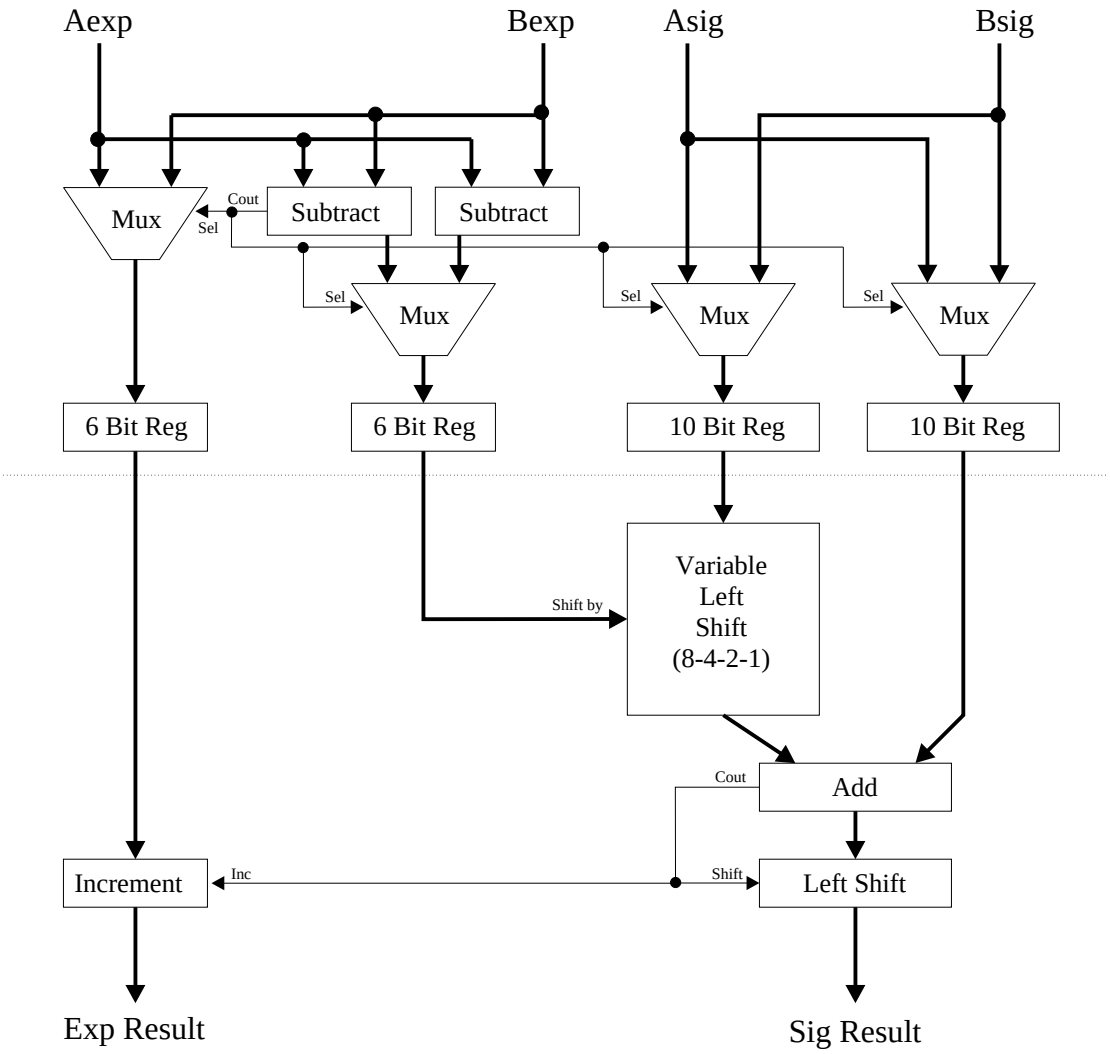

Figure 4.3: Block diagram of floating-point adder unit.

discussion of the algorithm implemented here.

4.5 Multiplier Design

The multiplier design is shown in Figure 4.4. The multiply operation operation starts with the two exponents being added together and then renormalized. Re-normalization is done by subtracting one bias value as discussed in Section 2.4. The result of the normalization is registered for the end of the first cycle.

The significands must be multiplied together, which is accomplished in ten stages. Each stage is a conditional adder which will add the multiplicand into the accumulated partial product (which is being passed from stage to stage) but only if the respective bit in the multiplier is set. There are five stages of the multiplier per pipeline stage. The two significands and the partial product are all registered at the end of the first pipeline stage. The output of the last stage is passed to a shifter which will shift left if there was a carry out from the last multiplier stage. The exponent is passed into a conditional incrementer which is also dependent upon the carry out from the last multiplier stage. The output from the incrementer and the output from the shifter make up the final result.

4.6 VHDL Source Organization

The VHDL source is in four files, one for each of the different PEs. These are the control unit, a standard processing unit, a left processing unit, and a right processing unit. The source code for the three processing units are basically identical, so the Appendix A only contains the source for the standard processing unit.

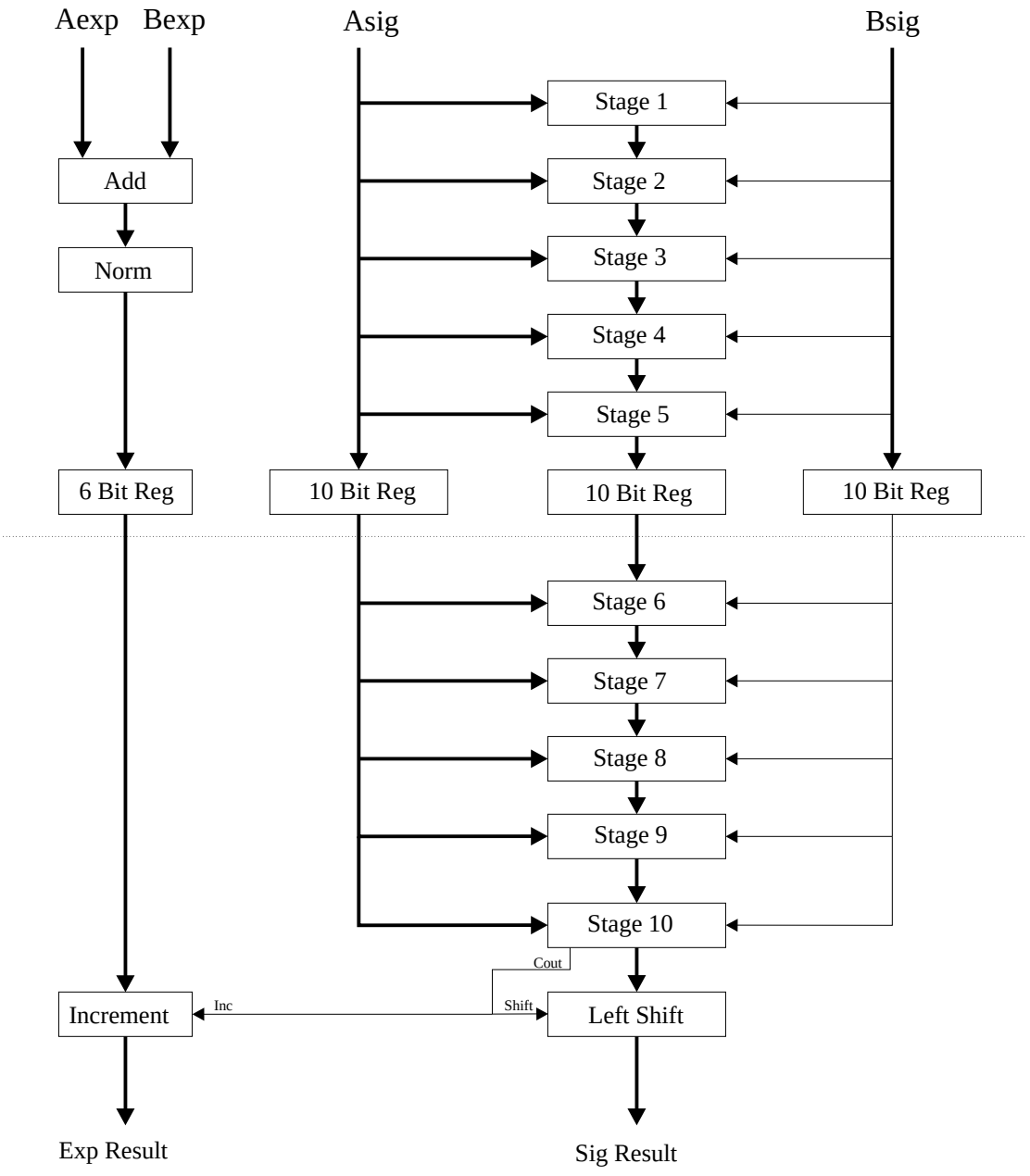

Figure 4.4: Block diagram of floating-point multiplier unit.

4.6.1 Control Unit

The control unit VHDL code is expressed mostly as a behavioral description. The control unit is made up of a two-mode state machine which is implemented as a case structure. When the system is first started, several values are read in from the memory of the control PE. These values indicate to the PE how many banks are in one mesh row, how many banks there are in the mesh, and how many time steps to compute. The state machine reads these values in when in the reset mode. Once the initialization is complete, the state machine runs in the standard mode which loops through all the states necessary for processing, and outputs the proper control values and memory addresses on the crossbar.

All of the addresses are calculated in the control PE based on the initialization values, so there are constructs for handling this. There are also constructs for counting the number of banks processed, and the total number of iterations which are processed. When the control PE determines that the requested processing is complete, a signal is sent to the host program indicating that the data is ready for retrieval.

4.6.2 Processing PEs

The code for the processing PEs is expressed mostly as a structural description, with the majority of the major components defined as X-Blox modules. Multiplexors are the only major components which are left up to the synthesis tool to define. One of the results of this type of code is that nearly every signal present in the final design must be defined explicitly. Although tedious, this method allows for the highest degree of control over the final product.

The structural definitions are organized as the major blocks which were discussed above (i.e. routing and preprocessing, adder, and multiplier.) Most of the control signals required for directing the operation of the PE come from the control PE, through the crossbar. Some other control signals are generated internally, but only those which are dependent on the nodal configuration word.

The addresses for the local memory also come from the control PE. Most of these addresses are passed through to the memory, but the addresses for the constants are modified internally. When a constant is being fetched, the base address is defined by the control PE, but several bits from the nodal configuration word are superimposed onto the address before it is sent to the memory. This is how the configuration word can pick the proper constants from memory.

Chapter 5

Results

The goal of this research project was to create an application which demonstrated the use of a CCM for physical system simulation. The idea was to harness the highly parallelized, configurable nature of the CCM to achieve substantial performance gains over general-purpose computing solutions.

The finite difference heat transfer simulation proved to be an excellent application for mapping to Splash. The algorithms proved to be straight-forward, relatively easy to implement, and capable of utilizing the majority of the available computing resources. The result is an efficient, functional application that demonstrates the capabilities of a CCM.

5.1 Capabilities and Performance

Because of the nature of physical systems simulation, the application proved to be very parellelizable. In fact, the system generally has a linear scalability with increasing processor array size. This means that the application can utilize as many processors as are available, and doubling the number of processors will double the performance of the application. This is a feature that is rare among applications.

Also the size of the problem which can be loaded into Splash will increase linearly with the

Table 5.1: Approximate maximum mesh sizes.

Boards	Total Nodes	Mesh Dimensions
1	1 Million	1,000 x 1,000
2	2 Million	1,400 x 1,400
4	4 Million	2,000 x 2,000
8	8 Million	2,800 x 2,800
16	16 Million	4,000 x 4,000

Table 5.2: Approximate performance compared to a Sun Microsystems Sparc-2 Workstation when computing 100 iterations of a 1024×1024 array.

Processor	Time	Speedup
Sparc-2	1.1 hours	1×

Processor	Clockrate					
	1 MHz		6 MHz[1]		12 MHz[1]	
	Time	Speedup	Time	Speedup	Time	Speedup
1 Splash Board	64 sec	62×	10.6 sec	372×	5.3 sec	744×
2 Splash Boards	32 sec	124×	5.3 sec	744×	2.65 sec	1488×
4 Splash Boards[1]	16 sec	248×	2.7 sec	1488×	1.35 sec	2976×
8 Splash Boards[1]	8 sec	496×	1.3 sec	2976×	0.65 sec	5952×
16 Splash Boards[1]	4 sec	992×	0.67 sec	5952×	0.33 sec	11904×

addition of processors. Table 5.1 shows the number of nodes which can be loaded into the Splash system depending on the number of boards which are installed. This is obviously a linear increase.

The computing performance of the system is dependent on the clock rate used to run the system. The development tools indicate the worst-case propagation delay through the processing PE to be around 170 ns, which means that the PEs can be clocked at approximately 6 MHz. The delay estimates are based upon a worst-case operating environment for the FPGAs, which is low voltage and high temperature with slow input signal rise times [30]. Experience collected over

[1]data extrapolated from experimental results.

many different applications has shown that the numbers returned by the development tools are very conservative for a controlled environment, such as the development lab. In this environment, where the temperature is controlled at 25°C and the supply voltages are a constant 5.0 Volts, applications have consistently run at clock rates 2-3 times the worst-case recommended rate. Therefore, the heat transfer application should easily run at clock rates as high as 12 MHz.

The application was developed and tested using a clock generated by the host workstation, running at approximately 1 MHz. Using this clock, timings were taken for running 100 iterations of a 1024×1024 mesh. Table 5.2 lists the computation times recorded. The times listed for 6 and 12 MHz clock rates are extrapolated from the initial test data. Likewise the data for four, eight, and sixteen Splash boards was also extrapolated because the development system only has two boards available. These extrapolations are valid approximations because of the linear performance increase with clock rate, and because of the linearly parellelizable nature of the application. The execution time required to complete the same task using the Sparc-2 workstation is included for comparison.

The speed advantage of the Splash-2 CCM is readily apparent, even when only configured with one board, running at a conservative 1 MHz clock rate.. The addition of more boards, and increasing the clock rate, scales the processing power linearly up to nearly 12,000$\times$ the performance of a Sparc 2 workstation (using 16 boards at 12 MHz).

Each PE computes a new nodal value every 10 clock cycles, during which seven floating point operations are performed. This information allows for computing a performance rating in millions of floating point operations per second, or megaflops. Table 5.3 shows the calculated performance

Table 5.3: Approximate performance ratings in megaflops.

Boards	1 MHz	6 MHz	12 MHz
1	11	67	134
2	22	134	269
4	45	269	538
8	89	538	1,076
16	179	1,076	2,152

ratings for one, six, and twelve MHz for several different board configurations.

5.1.1 Comparison to Digital Signal Processors

Digital signal processors are a class of processors specifically designed to perform mathematical operations quickly. Some DSPs claim high performance ratings in megaflops, but these ratings are not achievable under all situations.

The Analog Devices Sharc processor claims a performance of 120 megaflops [33], which is roughly comparable to the heat application running on a single Splash board. The Sharc performance rating however, is a peak rating which relies on the fact that its hardware is being utilized to the fullest. The utilization is achieved through instructions such as the multiply/accumulate instruction which performs both a multiply and an add. The equation utilized in the heat transfer application can only use a multiply/accumulate instruction in one instance. All other computations are single operations. Therefore a DSP could not actually achieve those performance numbers on the heat transfer application.

Furthermore, a DSP would require a lot more overhead (in terms of instructions) in order to fetch the proper values for each computation. This is overhead which is unnecessary under the

Splash architecture. This difference would further reduce the overall performance of the DSPs compared to Splash.

5.2 Error Analysis

Simulation of physical systems involves describing parameters of the system using real numbers. Floating-point representations of these numbers are effective, but not entirely correct. There is always some error involved in both the representation of numbers, and the result of calculations involving these numbers. These errors can propagate through a series of calculations, continuously compounding one another, until the overall accuracy of the simulation is compromised. This is especially true in an iterative simulation, where one iteration is completely dependent upon the results of the previous iteration. An analysis of these errors must be done in order to ensure the validity of the simulation results.

The representation of a number using a floating point format inherently includes some error. A measurement of the representation error is called the *average relative representation error*, or *ARRE*. The *ARRE* is calculated as

$$ARRE = \frac{\beta - 1}{ln\beta}\frac{2^{-m}}{4}$$

where β is the base of the exponent used in the format (2 for both the custom format and the IEEE format), and m is the number of bits in the mantissa. For the custom format, $m = 10$, so

$$ARRE = \frac{2 - 1}{ln2}\frac{2^{-10}}{4} = 0.0004.$$

Remember that this is a *relative* error, and the absolute error depends on the value being repre-

sented.

The representation error is not as important to the final result as the errors introduced by the calculations themselves. These errors are due to the necessary rounding of results which must occur during each mathematical operation. The rounding is caused by the limited number of bits available in the mantissa. According to [11], the error in a multiplication operation will be the sum of the errors of the two operands. For addition, the error is simply the maximum of the errors of the two operands. (This error rule only applies when both operands are of the same sign.)

Recall from Section 3.2 that the equation used for calculating nodal values is

$$NEWLOCAL = FO \cdot (RIGHT + LEFT + TOP + BOT + CONV) + END \cdot LOCAL.$$

The value $\delta(x)$ represents the error associated with the value x. Using this representation, and the rules given above, the error inherent in the above equation is

$$\begin{aligned} \delta(NEWLOCAL) = MAX((&\delta(FO) + \\ MAX(&\delta(RIGHT), \delta(LEFT), \delta(TOP), \delta(BOT), \delta(CONV))), \\ &(\delta(END) + \\ &\delta(LOCAL))). \end{aligned}$$

To condense this equation some, consider the error associated with any nodal value to be x. Since any add operation propagates the maximum error in any of its operands, the summation of the four surrounding nodal values will simply be the maximum of any of these, which is x (assume that x is larger than $\delta(CONV)$). In this case the above equation reduces to

$$\delta(NEWLOCAL) = MAX((\delta(FO) + x), (\delta(END) + x)).$$

Table 5.4: Resulting relative error after n iterations.

n	Relative Error
1	0.0004
10	0.004
100	0.04
1000	0.4
10000	4

Since $\delta(FO)$ and $\delta(END)$ are simply $ARRE$, the equation can be further reduced to

$$\delta(NEWLOCAL) = ARRE + x$$

which clearly shows how error will propagate through the simulation. This equation indicates that each iteration will introduce an additional error equal to the $ARRE$. Therefore, the error associated with any result is simply the number of iterations performed, multiplied by the $ARRE$. Table 5.4 shows the error after n iterations, for several different values of n.

Although these error estimates seem reasonable, problems can occur rather quickly. Section 2.2.4 introduced the concept of the stability criterion, and mentioned that in order to maintain stability within the simulation, a sufficiently small value of Δx must be selected. In many situations, Δx can be on the order of one one-thousandth of a second, or smaller. If a simulation investigates the change over a mere 10 seconds, the simulation will already introduce a 4% relative error into the results. This resolution may be acceptable for some simulations, but not all. Clearly, there is room for improvement in the application.

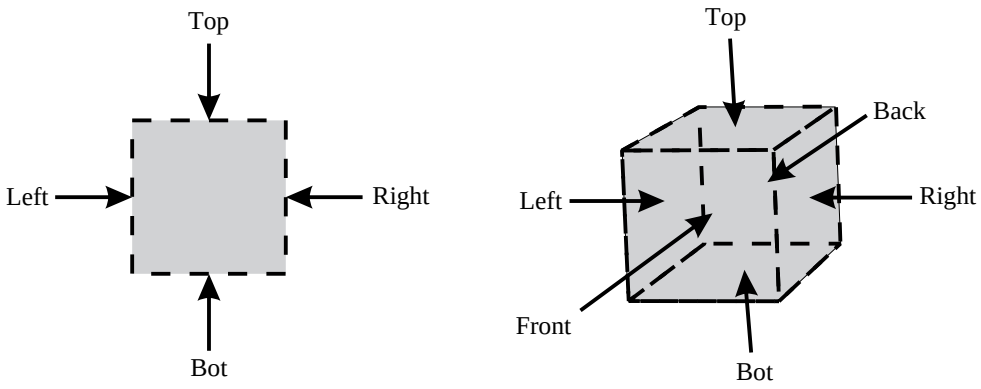

Figure 5.1: Adjacent nodal data for two and three dimensions.

5.3 Extension to Three Dimensions

The heat transfer application is readily extendible to three dimensional problems. The control unit would need to be redesigned to handle the extra dimension, but the changes would be minimal. The memory address generators would need to be modified to properly generate the addresses for the third dimension, and the compute cycle would need to be modified to handle the extra data. Specifically the processors would need to compute the result of the following equation

$$NEWLOCAL = FO \cdot (RIGHT + LEFT + TOP + BOT + FRONT + BACK + CONV) + END \cdot LOCAL.$$

Note the addition of the two new operands, FRONT and BACK. These operands are the two extra data points for the nodes adjacent in the third dimension. Figure 5.1 shows where the adjacent data values come from in both the two dimensional and three dimensional problems.

The constant data values would also change significantly, but these changes would not affect the hardware design of the application. Overall the extension to three dimensions is relatively straight-

forward, but because it requires modifications to hardware, a significant effort will be required.

5.4 Software Utilities

The heat transfer application is comprised of several utilities which allow for a complete analysis flow frow geometric definition to visualization of the results. The coordination of these utilities is achieved through the use of a graphical user interface (GUI) specifically designed for this application. The GUI is shown in Figure 5.2.

The geometric representation of a problem is defined through the use of a bitmap editing program called *xpaint*. This application allows the geometry of the problem to be defined by assigning a pixel to each node in the analysis. Different colors differentiate between the different types of boundary conditions (adiabatic, convective, direct flux, etc.) Physical constants for the problem are entered into the GUI, and then the utility *xpm2heat* is run from the GUI in order to convert the bitmap and the constants into a problem data file for the actual application. Figure 5.3 shows the bitmap definition for a small (16×16) problem. Note that the convective edges are show in a different color. Figure 5.4 shows the bitmap definition for a much larger (512×512) problem.

After the analysis is performed, the GUI will run another utility which will convert the output data into a visible result. Figure 5.5 is the resulting image from the problem defined by Figure 5.4. Note that the temperature gradients around the convective surfaces are clearly visible.

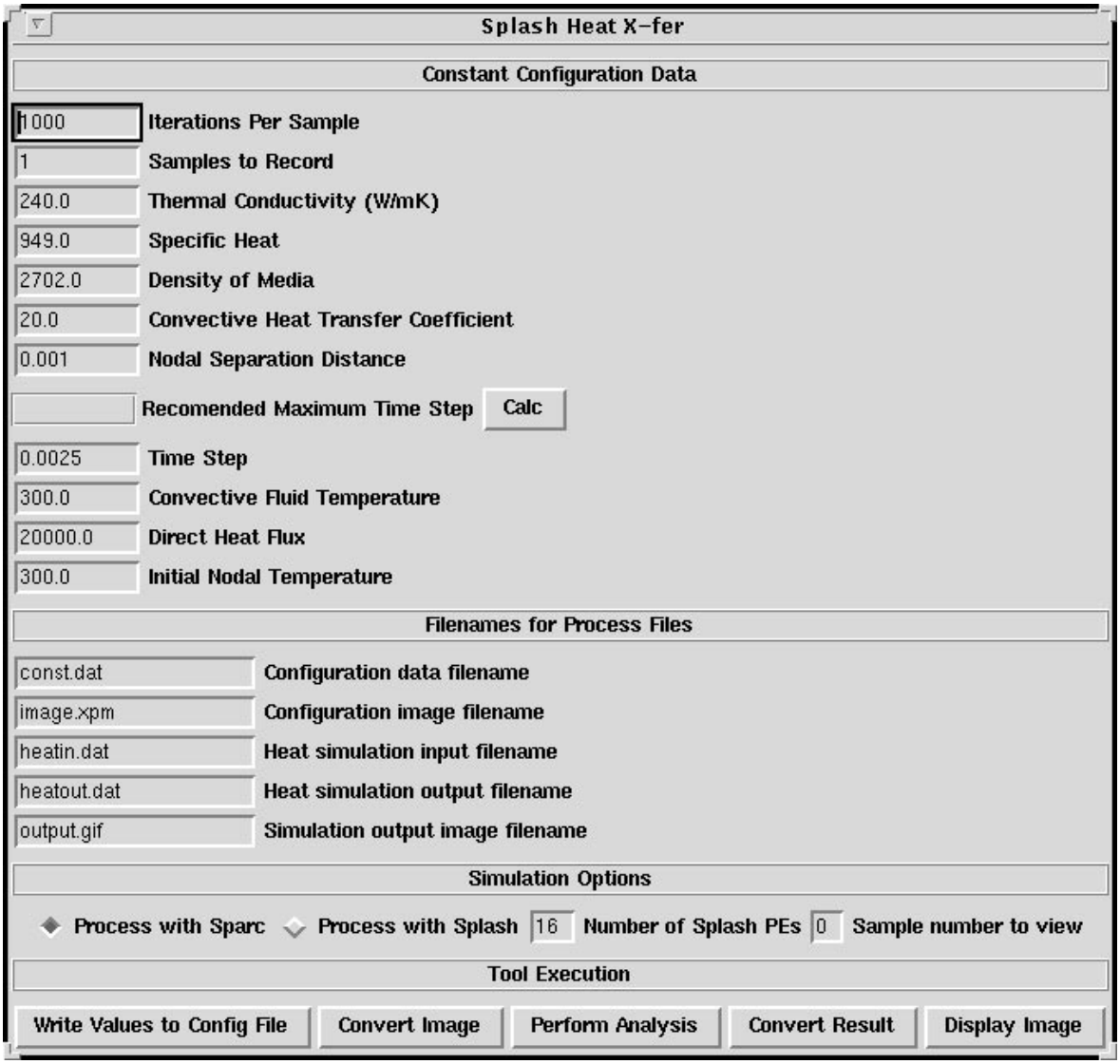

Figure 5.2: Graphical user interface for heat transfer application.

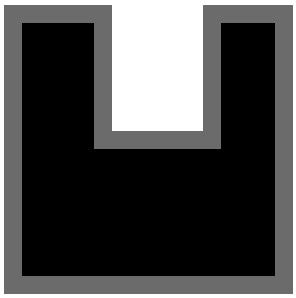

Figure 5.3: A sample bitmaped definition for a 16×16 mesh.

5.5 Conclusions

The heat transfer application proved to be an excellent candidate for implementation on a CCM. The speedup achieved through the use of custom hardware, and the linearly scalable nature of the problem, maximized the benefits of the platform.

The heat transfer application was selected primarily for its simplicity, leaving behind more challenging applications like structural modeling and fluid dynamics; these applications require significantly more complex functions to be mapped into the PEs of the target platform. However, the performance gains achieved with this application indicate that the effort necessary in investigating these simulations is warranted.

The only drawback to the current implementation is accuracy. The error analysis indicated that problems may arise quickly in large simulations. These problems can easily be overcome with a more robust floating point format. The larger floating point format will necessarily decrease the overall speed of the application, but a reduction in performance by a factor of two or four will still leave the CCM with an enviable advantage over the workstation solution. Furthermore, this reduction in performance could actually be eliminated if the data paths to and from the memory

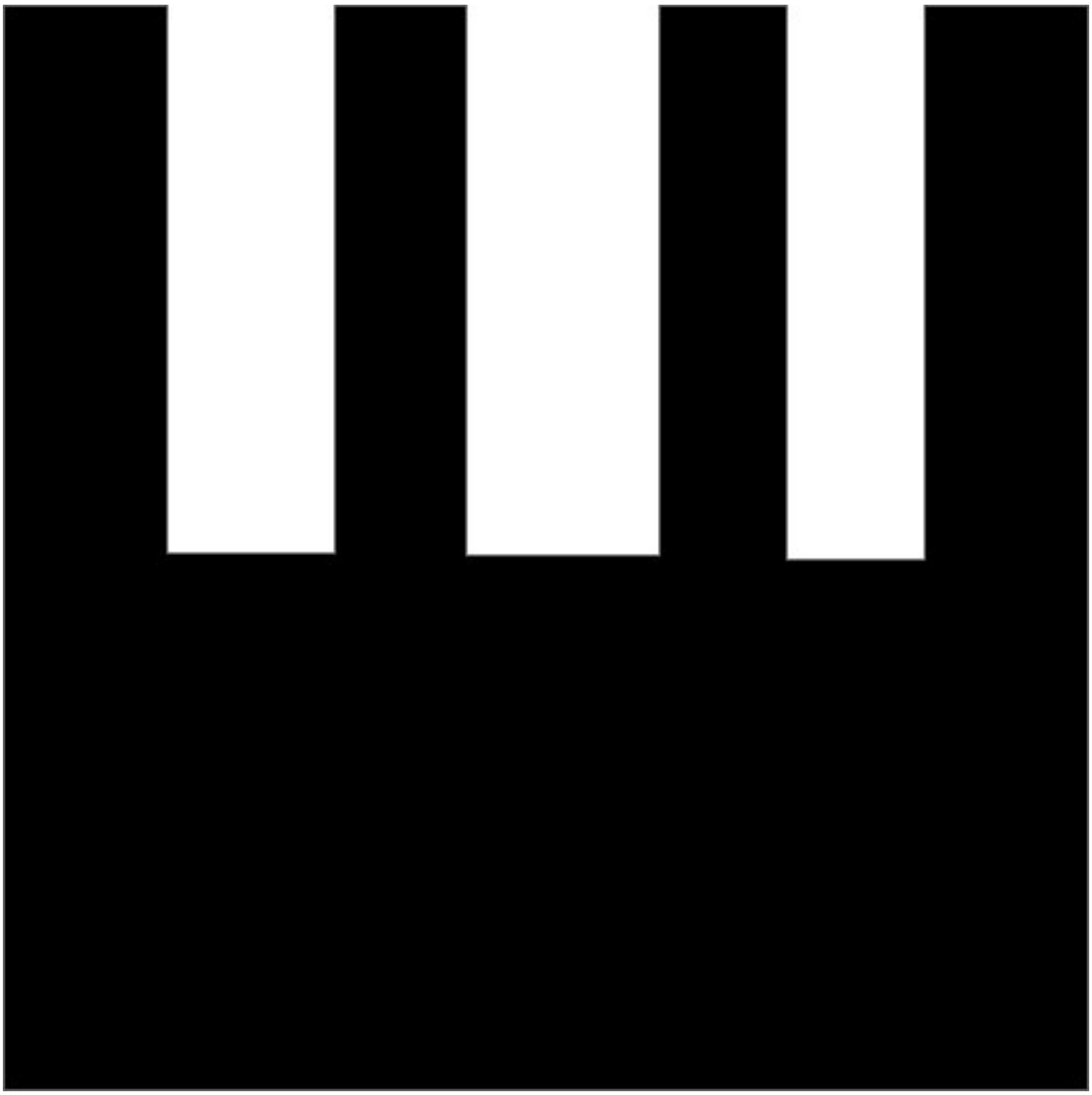

Figure 5.4: A sample bitmap definition for a 512×512 mesh.

Figure 5.5: Output image for a 512×512 sample problem.

banks were widened to allow for a greater memory bandwidth.

Although this application stands as a proof of concept for mapping physical simulation algorithms to CCMs, there is still much room for continued research.

REFERENCES

[1] D. A. Buell, J. M. Arnold, and W. J. Kleinfeldner, *Splash 2: FPGAs in a Custom Computing Machine*, Los Alamitos, CA: IEEE Computer Society Press, 1996.

[2] J. N. Reddy and D. K. Gartling, *The Finite Element Method in Heat Transfer and Fluid Dynamics*, Boca Raton, FL: CRC Press, 1994.

[3] L. Lapidus and G. F. Pinder, *Numerical Solution of Partial Differential Equations in Science and Engineering*, New York: John Wiley and Sons, 1982.

[4] K. Hwang, *Advanced computer architecture : parallelism, scalability, programmability*, New York : McGraw-Hill, 1993.

[5] F. P. Incropera and D. P. DeWitt, *Introduction to Heat Transfer, 2nd ed.*, New York: John Wiley and Sons, 1990.

[6] J. L. Hennessy and D. A. Patterson, *Computer Architecture A Quantitative Approach, 2nd ed.*, San Francisco, CA: Morgan Kaufmann Publishers, Inc., 1996.

[7] N. Weste and K. Eshraghian, *Principles of CMOS VLSI Design: A Systems Perspective, 2nd ed.*, Reading, MA: Addison-Wesley, 1993.

[8] J. R. Armstrong and F. G. Gray, *Structured Logic Design with VHDL*, Upper Saddle River, NJ: Prentice Hall, 1993.

[9] F. Neelamkavil, *Computer Simulation and Modelling*, Chichester: John Wiley and Sons, 1987.

[10] I. Koren, *Computer Arithmetic Algorithms*, Englewood Cliffs, NJ: Prentice Hall, 1993.

[11] J. L. Buchanan and P. R. Turner, *Numerical Methods and Analysis*, New York: McGraw-Hill, 1992.

[12] J. M. Arnold, D. A. Buell, and E. G. Davis, *Splash 2*, Proceedings of the 4th Annual ACM Symposium on Parallel Algorithms and Architectures, June 1992, pp. 316-322.

[13] D. A. Buell and K. L. Pocek, "Custom Computing Machines: An Introduction," *Journal of Supercomputing*, vol. 9, no. 3, pp. 219-29, 1995.

[14] N. Shirazi, A. Walters, and P. Athanas, "Quantitative Analysis of Floating Point Arithmetic on FPGA Based Custom Computing Machines," *Proceedings IEEE Symposium on FPGAs for Custom Computing Machines*, P. Athanas and K. L. Pocek (eds.), pp. 155-62, April 1995.

[15] S. Casselman, "Virtual Computing and The Virtual Computer," *Proceedings IEEE Symposium on FPGAs for Custom Computing Machines*, D. A. Buell and K. L. Pocek (eds.), pp. 43-48, April 1993.

[16] J. A. Arnold, "The Splash 2 Software Environment," *Proceedings IEEE Symposium on FPGAs for Custom Computing Machines*, D. A. Buell and K. L. Pocek (eds.), pp. 88-93, April 1993.

[17] R. Amerson, R. J. Carter, W. B. Culbertson, P. Kuekes, and G. Snider, "Teramac-Configurable Custom Computing", *Proceedings IEEE Symposium on FPGAs for Custom Computing Machines*, P. Athanas and K. L. Pocek (eds.), pp. 32-37, April 1995.

[18] P. M. Athanas and H. F. Silverman, "Processor reconfiguration through instruction set metamorphosis: Architecture and compiler", IEEE Computer, vol. 26, pp. 11-18, March 1993.

[19] B. Box, "Field programmable gate array based reconfigurable preprocessor", *Proceedings IEEE Symposium on FPGAs for Custom Computing Machines*, D. A. Buell and K. L. Pocek (eds.), pp. 40-49, April 1994.

[20] B. L. Hutchings and M. J. Wirthlin, "A Dynamic Instruction Set Computer", *Proceedings IEEE Symposium on FPGAs for Custom Computing Machines*, P. Athanas and K. L. Pocek (eds.), pp. 99-107, April 1995.

[21] S. A. Cuccaro, "The CM-2X: A Hybrid CM-2/Xilinx Prototype", *Proceedings IEEE Symposium on FPGAs for Custom Computing Machines*, D. A. Buell and K. L. Pocek (eds.), pp. 121-130, April 1993.

[22] A. DeHon and E. Mirsky, "MATRIX: A Reconfigurable Computing Architecture with Configurable Instruction Distribution and Deployable Resources," *Preliminary Proceedings IEEE Symposium on FPGAs for Custom Computing Machines*, J. Arnold and K. L. Pocek (eds.), April 1996.

[23] Y. K. Cheung, W. Luk, and N. Shirazi, "Modelling and Optimising Run-Time Reconfigurable Systems," *Preliminary Proceedings IEEE Symposium on FPGAs for Custom Computing Machines*, J. Arnold and K. L. Pocek (eds.), April 1996.

[24] T. A. Cook, W. H. Johnson, and L. Louca, "Implementation of IEEE Single Precision Floating Point Addition and Multiplicaiton on FPGAs", *Preliminary Proceedings IEEE Symposium on FPGAs for Custom Computing Machines*, J. Arnold and K. L. Pocek (eds.), April 1996.

[25] P. Athanas, I. Howitt, T. Rappaport, J. Reed, and B. Woerner, "A High Capacity Wireless Receiver Implemented with a Reconfigurable Computer Architecture", ARPA GloMo Principle Investigators Conference, San Diego, CA, Nov. 1995.

[26] J. A. Arnold, "Splash 2 Programmer's Manual." Technical Report, Supercomputing Research Center, December 1992.

[27] D. A. Buell, "A Splash 2 Tutorial, version 1.1." Technical Report, Supercomputing Research Center, December 1992.

[28] B. Fagin and C. Renard, "Field Programmable Gate Arrays and Floating Point Arithmetic," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 2, no. 3, pp. 365-367, Sept. 1994.

[29] IEEE, "IEEE Standard for Binary Floating-Point Arithmetic," ANSI/IEEE Std. 754-1985, August, 1985.

[30] Xilinx, Inc., *The Field Programmable Logic Data Book*, San Jose, CA, 1994.

[31] Xilinx, Inc., *XC6200 FPGA Family: Advanced Product Description,* San Jose, CA, 1995.

[32] Annapolis Micro Systems, Inc., *Wildfire*, product literature, 1995.

[33] Advanced Micro Devices, Inc., *ADSP-21060/21062 Super Harvard Architecture Computer*, product literature, 1995.

Appendix A

Processor Source Code

```
--------------------------------------------------------------------------------
--
-- Splash-2 Based Finite Difference Heat Transfer Analysis - Processing PE
--
-- Copyright (C) 1996 by Kevin J. Paar and
-- Virginia Polytechnic Institute and State University
-- All rights reserved.
--
-- Do not distribute further without the concurrence of the authors.
--
--------------------------------------------------------------------------------

library SPLASH2;
use SPLASH2.TYPES.all;
use SPLASH2.SPLASH2.all;
use SPLASH2.COMPONENTS.all;
use SPLASH2.ARITHMETIC.all;

-- Begin Entity Comment

--------------------------------------------------------------------------------
--  Splash 2 Simulator v1.5 Xilinx_Processing_Part Entity Declaration
--------------------------------------------------------------------------------
entity Xilinx_Processing_Part is
  Generic(
    BD_ID               : Integer := 0; --  Splash Board ID
    PE_ID            : Integer := 1  --  Processing Element ID
  );
 Port (
    XP_Left         : inout DataPath; --  Left Data Bus
    XP_Right         : inout DataPath; --  Right Data Bus
    XP_Xbar : inout DataPath; --  Crossbar Data Bus
    XP_Xbar_EN_L : out Bit_Vector(4 downto 0); --  Crossbar Enable (low-true)
    XP_Clk         : in  Bit; --  Splash System Clock
    XP_Int         : out   Bit; --  Interrupt Signal
    XP_Mem_A         : inout MemAddr; --  Splash Memory Address Bus
```

```
    XP_Mem_D         : inout MemData; --  Splash Memory Data Bus
    XP_Mem_RD_L : inout RBit3;       --  Splash Memory Read Signal
(low-true)
    XP_Mem_WR_L        : inout RBit3;      --  Splash Memory Write
Signal (low-true)
   XP_Mem_Disable : in   Bit; --  Splash Memory Disable
Signal
    XP_Broadcast         : in  Bit; --  Broadcast Signal
    XP_Reset         : in  Bit; --  Reset Signal
    XP_HS0                  : inout RBit3; --  Handshake Signal Zero
    XP_HS1                 : in     Bit; --  Handshake Signal One
    XP_GOR_Result : inout RBit3; --  Global OR Result Signal
    XP_GOR_Valid         : inout RBit3; --  Global OR Valid Signal
    XP_LED         : out   Bit --  LED Signal
  );
end Xilinx_Processing_Part;
--  End Entity Comment

architecture heat_proc of Xilinx_Processing_Part is

  component xb_expadd
    port (
      a:       in   Bit_Vector(5 downto 0);
      b:       in   Bit_Vector(5 downto 0);
      c_in:   in   Bit;
      func:  out  Bit_Vector(5 downto 0)
      );
  end component;

  component xb_expsub
    port(
      a : in  Bit_Vector(5 downto 0);
      b : in  Bit_Vector(5 downto 0);
      func         : out Bit_Vector(5 downto 0);
      c_out         : out Bit;
      ovfl         : out Bit
      );
  end component;

  component xb_expreg
    port (
      d_in:    in   Bit_Vector (5 downto 0);
      clock:   in   Bit;
      q_out:   out  Bit_Vector (5 downto 0)
      );
  end component;
```

```
component xb_expreg_clr
  port (
    sync_ctrl: in   Bit;
    d_in:      in   Bit_Vector (5 downto 0);
    clock:     in   Bit;
    clk_en:    in   Bit;
    q_out:     out  Bit_Vector (5 downto 0)
    );
end component;

component xb_sigadd
  port (
    a:      in   Bit_Vector(9 downto 0);
    b:      in   Bit_Vector(9 downto 0);
    func:   out  Bit_Vector(9 downto 0);
    c_out:  out  Bit
    );
end component;

component xb_sigreg
  port (
    d_in:    in   Bit_Vector (9 downto 0);
    clock:   in   Bit;
    q_out:   out  Bit_Vector (9 downto 0)
    );
end component;

component xb_sigreg_clr
  port (
    sync_ctrl: in   Bit;
    d_in:      in   Bit_Vector (9 downto 0);
    clock:     in   Bit;
    clk_en:    in   Bit;
    q_out:     out  Bit_Vector (9 downto 0)
    );
end component;

component xb_boundadd
  port (
    add_sub:  in   Bit;
    a:  in   Bit_Vector(15 downto 0);
    b:  in   Bit_Vector(15 downto 0);
    func:  out  Bit_Vector(15 downto 0)
    );
end component;

component xb_wordreg
```

```
    port (
      d_in:    in   Bit_Vector (15 downto 0);
      clock:   in   Bit;
      clk_en:  in   Bit;
      q_out:   out  Bit_Vector (15 downto 0)
      );
  end component;

  component xb_bitreg
    port (
      d_in:    in   Bit;
      clock:   in   Bit;
      clk_en:  in   Bit;
      q_out:   out  Bit
      );
  end component;

  component xb_xbreg
    port (
      d_in:    in   Bit_Vector (35 downto 0);
      clock:   in   Bit;
      q_out:   out  Bit_Vector (35 downto 0)
      );
  end component;

  component xb_addreg
    port (
      d_in:    in   Bit_Vector (17 downto 0);
      clock:   in   Bit;
      q_out:   out  Bit_Vector (17 downto 0)
      );
  end component;

  for all: xb_expadd   use entity work.xb_expadd(sim);
  for all: xb_expsub   use entity work.xb_expsub(sim);
  for all: xb_expreg   use entity work.xb_expreg(sim);
  for all: xb_expreg_clr  use entity work.xb_expreg_clr(sim);
  for all: xb_sigadd      use entity work.xb_sigadd(sim);
  for all: xb_sigreg   use entity work.xb_sigreg(sim);
  for all: xb_sigreg_clr  use entity work.xb_sigreg_clr(sim);
  for all: xb_wordreg   use entity work.xb_wordreg(sim);
  for all: xb_boundadd   use entity work.xb_boundadd(sim);
  for all: xb_bitreg      use entity work.xb_bitreg(sim);
  for all: xb_xbreg   use entity work.xb_xbreg(sim);
  for all: xb_addreg      use entity work.xb_addreg(sim);

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

```
-- Definitions for bussing signals
-----------------------------------------------------------------

SIGNAL XBar                   : Bit_Vector(DATAPATH_WIDTH-1 downto 0);
SIGNAL Maddr                  : Bit_Vector(17 downto 0);
SIGNAL MdataIn                : Bit_Vector(15 downto 0);
SIGNAL MdataOut : Bit_Vector(15 downto 0);
SIGNAL MemAdd : Bit_Vector(15 downto 0);
SIGNAL Mtemp : Bit_Vector(15 downto 0);

SIGNAL CtrlWord : Bit_Vector(15 downto 0);
SIGNAL LocalTemp      : Bit_Vector(15 downto 0);

SIGNAL Ainc                   : Bit_Vector(5 downto 0);
SIGNAL Aexp : Bit_Vector(5 downto 0);
SIGNAL Asig : Bit_Vector(9 downto 0);
SIGNAL Abus : Bit_Vector(15 downto 0);

SIGNAL Binc                   : Bit_Vector(5 downto 0);
SIGNAL Bexp : Bit_Vector(5 downto 0);
SIGNAL Bsig : Bit_Vector(9 downto 0);
SIGNAL Bbus : Bit_Vector(15 downto 0);

SIGNAL MathRslt : Bit_Vector(15 downto 0);

SIGNAL LeftIn : Bit_Vector(15 downto 0);
SIGNAL RightIn : Bit_Vector(15 downto 0);

-----------------------------------------------------------------
-- Definitions for Control Signals
-----------------------------------------------------------------

SIGNAL C_LoadLocal            : Bit;
SIGNAL C_LoadCtrl             : Bit;
SIGNAL C_LoadB                : Bit;
SIGNAL C_LoadBound            : Bit;
SIGNAL C_GetLT                : Bit;
SIGNAL C_GetRB                : Bit;
SIGNAL C_MemEnd               : Bit;
SIGNAL C_MemFo                : Bit;
SIGNAL C_MemCon               : Bit;
SIGNAL C_Left2A               : Bit;
SIGNAL C_Rght2A               : Bit;
SIGNAL C_Locl2A               : Bit;
SIGNAL C_Abus2B               : Bit;
SIGNAL C_ResMult              : Bit;
SIGNAL C_LoadRes              : Bit;
```

```
SIGNAL C_MemBound            : Bit;
SIGNAL C_Mwrite              : Bit;
SIGNAL C_Mread               : Bit;

SIGNAL C_WrEn                : Bit;
SIGNAL ConstNo               : Bit_Vector(2 downto 0);
SIGNAL EndConst              : Bit_Vector(1 downto 0);
SIGNAL FoConst               : Bit_Vector(1 downto 0);
SIGNAL C_Bot2x               : Bit;
SIGNAL C_BotZe               : Bit;
SIGNAL C_Rgt2x               : Bit;
SIGNAL C_RgtZe               : Bit;
SIGNAL C_Top2x               : Bit;
SIGNAL C_TopZe               : Bit;
SIGNAL C_Lft2x               : Bit;
SIGNAL C_LftZe               : Bit;

SIGNAL C_IncA                : Bit;
SIGNAL C_IncB                : Bit;
SIGNAL C_ClrA                : Bit;
SIGNAL C_ClrB                : Bit;

SIGNAL C_WrSig               : Bit;

-----------------------------------------------------------------
-- Definitions for the adder unit
-----------------------------------------------------------------

SIGNAL Add_SubAB : Bit_Vector(5 downto 0);
SIGNAL Add_SubAB_C : Bit;
SIGNAL Add_SubAB_O : Bit;
SIGNAL Add_SubBA : Bit_Vector(5 downto 0);
SIGNAL Add_SubBA_C : Bit;
SIGNAL Add_SubBA_O : Bit;

SIGNAL Add_SwapOp : Bit;
SIGNAL Add_SubRes : Bit_Vector(5 downto 0);
SIGNAL Add_ShiftBy : Bit_Vector(5 downto 0);
SIGNAL Add_Exp2reg : Bit_Vector(5 downto 0);
SIGNAL Add_ExpS1 : Bit_Vector(5 downto 0);
SIGNAL Add_SigAsel : Bit_Vector(9 downto 0);
SIGNAL Add_SigBsel : Bit_Vector(9 downto 0);
SIGNAL Add_SigA : Bit_Vector(9 downto 0);
SIGNAL Add_SigB : Bit_Vector(9 downto 0);

SIGNAL Add_Shift : Bit_Vector(3 downto 0);
SIGNAL Add_Shift8 : Bit_Vector(9 downto 0);
```

```
  SIGNAL Add_Shift4 : Bit_Vector(9 downto 0);
  SIGNAL Add_Shift2 : Bit_Vector(9 downto 0);
  SIGNAL Add_Shiftd : Bit_Vector(9 downto 0);

  SIGNAL Add_RawAdd : Bit_Vector(9 downto 0);
  SIGNAL Add_AddCO : Bit;
  SIGNAL Add_ExpRes : Bit_Vector(5 downto 0);
  SIGNAL Add_SigRes : Bit_Vector(9 downto 0);

  ------------------------------------------------------------------
  -- Definitions for the Multiplier unit
  ------------------------------------------------------------------

  SIGNAL Mult_ExpTmp : Bit_Vector(5 downto 0);
  SIGNAL Mult_Exp2Reg : Bit_Vector(5 downto 0);
  SIGNAL Mult_ExpS1 : Bit_Vector(5 downto 0);
  SIGNAL Mult_ExpRes : Bit_Vector(5 downto 0);

  SIGNAL Mult_PP1 : Bit_Vector(10 downto 0);
  SIGNAL Mult_PA1 : Bit_Vector(9 downto 0);
  SIGNAL Mult_PP2 : Bit_Vector(10 downto 0);
  SIGNAL Mult_PA2 : Bit_Vector(9 downto 0);
  SIGNAL Mult_PP3 : Bit_Vector(10 downto 0);
  SIGNAL Mult_PA3 : Bit_Vector(9 downto 0);
  SIGNAL Mult_PP4 : Bit_Vector(10 downto 0);
  SIGNAL Mult_PA4 : Bit_Vector(9 downto 0);
  SIGNAL Mult_PP5 : Bit_Vector(10 downto 0);
  SIGNAL Mult_PA5 : Bit_Vector(9 downto 0);
  SIGNAL Mult_PP6 : Bit_Vector(10 downto 0);
  SIGNAL Mult_PP6Reg : Bit_Vector(9 downto 0);
  SIGNAL Mult_Mcand : Bit_Vector(9 downto 0);
  SIGNAL Mult_Mplier : Bit_Vector(9 downto 0);

  SIGNAL Mult_PA6 : Bit_Vector(9 downto 0);
  SIGNAL Mult_PP7 : Bit_Vector(10 downto 0);
  SIGNAL Mult_PA7 : Bit_Vector(9 downto 0);
  SIGNAL Mult_PP8 : Bit_Vector(10 downto 0);
  SIGNAL Mult_PA8 : Bit_Vector(9 downto 0);
  SIGNAL Mult_PP9 : Bit_Vector(10 downto 0);
  SIGNAL Mult_PA9 : Bit_Vector(9 downto 0);
  SIGNAL Mult_PP10 : Bit_Vector(10 downto 0);
  SIGNAL Mult_SigRes : Bit_Vector(9 downto 0);

  ------------------------------------------------------------------
  -- Miscellaneous definitions
  ------------------------------------------------------------------
```

```
  SIGNAL Yes : Bit;
  SIGNAL No : Bit;
  SIGNAL Zero_6 : Bit_Vector(5 downto 0);
  SIGNAL Zero_10 : Bit_Vector(9 downto 0);
  SIGNAL C_WrEn_D               : Bit;
  SIGNAL C_BoundOp : Bit;

  SIGNAL XBarR : Bit_Vector(35 downto 0);
  SIGNAL MdataInR : Bit_Vector(15 downto 0);
  SIGNAL MaddrR : Bit_Vector(17 downto 0);
  SIGNAL C_MreadR : Bit;
  SIGNAL C_WrSigR : Bit;
  SIGNAL LEDSig : Bit;

begin

  ---------------------------------------------------------------------
  -- Control Signal Breakout - main control from XBar
  ---------------------------------------------------------------------

  C_LoadLocal   <= XBar(0);
  C_LoadCtrl    <= XBar(1);
  C_LoadB       <= XBar(2);
  C_LoadBound   <= XBar(3);

  C_GetLT       <= XBar(4);
  C_GetRB       <= XBar(5);

  C_MemEnd      <= XBar(6);
  C_MemFo       <= XBar(7);
  C_MemCon      <= XBar(8);

  C_Left2A      <= XBar(9);
  C_Rght2A      <= XBar(10);
  C_Locl2A      <= XBar(11);
  C_Abus2B      <= XBar(12);

  C_ResMult     <= XBar(13);
  C_LoadRes     <= XBar(14);

  C_MemBound    <= XBar(15);

  C_Mwrite      <= XBar(16);
  C_Mread       <= XBar(17);

  ---------------------------------------------------------------------
  -- Control Signal Breakout - from control word
```

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

C_WrEn   <= CtrlWord(0);
ConstNo  <= CtrlWord(3 downto 1);
EndConst <= CtrlWord(5 downto 4);
FoConst  <= CtrlWord(7 downto 6);

C_Bot2x  <= CtrlWord(8);
C_BotZe  <= CtrlWord(9);
C_Rgt2x  <= CtrlWord(10);
C_RgtZe  <= CtrlWord(11);
C_Top2x  <= CtrlWord(12);
C_TopZe  <= CtrlWord(13);
C_Lft2x  <= CtrlWord(14);
C_LftZe  <= CtrlWord(15);

-------------------------------------------------------------------
-- Generate Composite Control Signals
-------------------------------------------------------------------

C_IncA   <= (C_Lft2x AND C_GetLT) OR (C_Rgt2x AND C_GetRB);
C_IncB   <= (C_Top2x AND C_GetLT)  OR (C_Bot2x AND C_GetRB);

C_ClrA   <= (C_LftZe AND C_GetLT) OR (C_RgtZe AND C_GetRB) OR not(Abus(9));
C_ClrB   <= (C_TopZe AND C_GetLT)  OR (C_BotZe AND C_GetRB) OR not(Bbus(9));

-------------------------------------------------------------------
-- Data Bussing
-------------------------------------------------------------------

Abus <= LeftIn        when C_Left2A = '1' else
        RightIn       when C_Rght2A = '1' else
        LocalTemp when C_Locl2A = '1' else
        MathRslt;

Bbus <= Aexp & Asig   when C_Abus2B = '1' else
        MdataIn;

-------------------------------------------------------------------
-- Registers for local temperature and the control word and bound
-------------------------------------------------------------------

LOCREG: xb_wordreg
  port map(MdataIn, XP_Clk, C_LoadLocal, LocalTemp);

CTRREG: xb_wordreg
  port map(MdataIn, XP_Clk, C_LoadCtrl, CtrlWord);
```

```
  WRENREG: xb_bitreg
    port map(C_WrEn, XP_Clk, C_LoadCtrl, C_WrEn_D);

--  BOUNDREG: xb_wordreg
--    port map(MdataIn, XP_Clk, C_LoadBound, LeftIn);

  ---------------------------------------------------------------------
  -- Instantiate the incrementer and registers for A and B
  ---------------------------------------------------------------------

  AEXPINC: xb_expadd
    port map(Abus(15 downto 10), Zero_6, C_IncA, Ainc );

  AEXPREG: xb_expreg_clr
    port map(C_ClrA, Ainc, XP_Clk, Yes, Aexp);

  ASIGREG: xb_sigreg_clr
    port map(C_ClrA, Abus(9 downto 0), XP_Clk, Yes, Asig);

  BEXPINC: xb_expadd
    port map(Bbus(15 downto 10), Zero_6, C_IncB, Binc );

  BEXPREG: xb_expreg_clr
    port map(C_ClrB, Binc, XP_Clk, C_LoadB, Bexp);

  BSIGREG: xb_sigreg_clr
    port map(C_ClrB, Bbus(9 downto 0), XP_Clk, C_LoadB, Bsig);

  ---------------------------------------------------------------------
  -- Adder - exponent subtractors
  ---------------------------------------------------------------------

  EXPSUBA: xb_expsub
    port map(Aexp, Bexp, Add_SubAB, Add_SubAB_C, Add_SubAB_O);

  EXPSUBB: xb_expsub
    port map(Bexp, Aexp, Add_SubBA, Add_SubBA_C, Add_SubBA_O);

  ---------------------------------------------------------------------
  -- Adder - Decide if swap needs to be done...
  -- do it, register results
  ---------------------------------------------------------------------

  Add_SwapOp <= Add_SubAB_C;

  Add_SubRes <= Add_SubAB when Add_SwapOp = '1' else Add_SubBA;
```

```
SHFTREG: xb_expreg
  port map(Add_SubRes, XP_Clk, Add_ShiftBy);

Add_Exp2reg <= Aexp when Add_SwapOp = '1' else Bexp;

EXPREGS1: xb_expreg
  port map(Add_Exp2reg, XP_Clk, Add_ExpS1);

Add_SigAsel <= Asig when Add_SwapOp = '1' else Bsig;
Add_SigBsel <= Bsig when Add_SwapOp = '1' else Asig;

SIGAREGS1: xb_sigreg
  port map(Add_SigAsel, XP_Clk, Add_SigA);

SIGBREGS1: xb_sigreg
  port map(Add_SigBsel, XP_Clk, Add_SigB);

-------------------------------------------------------------------
-- Adder - Shift smaller significand right
-------------------------------------------------------------------

Add_Shift <= "1111" when (Add_ShiftBy(5) = '1' or Add_ShiftBy(4) = '1')
                    else Add_ShiftBy(3 downto 0);

Add_Shift8 <= Add_SigB when Add_Shift(3) = '0' else
              "00000000" & Add_SigA(9 downto 8);

Add_Shift4 <= Add_Shift8 when Add_Shift(2) = '0' else
              "0000" & Add_Shift8(9 downto 4);

Add_Shift2 <= Add_Shift4 when Add_Shift(1) = '0' else
              "00" & Add_Shift4(9 downto 2);

Add_Shiftd <= Add_Shift2 when Add_Shift(0) = '0' else
              '0' & Add_Shift2(9 downto 1);

-------------------------------------------------------------------
-- Adder - Do addition and normalize result if necessary
-------------------------------------------------------------------

SIGADD: xb_sigadd
  port map(Add_SigA, Add_Shiftd, Add_RawAdd, Add_AddCO);

EXPNORM: xb_expadd
  port map(Add_ExpS1, Zero_6, Add_AddCO, Add_ExpRes);
```

```
Add_SigRes <= '1' & Add_RawAdd(9 downto 1) when Add_AddCO = '1' else
              Add_RawAdd;

-----------------------------------------------------------------
-- Multiplier - exponent add, normalize and register
-----------------------------------------------------------------

EXPMULT: xb_expadd
  port map(Aexp, Bexp, No, Mult_ExpTmp);

Mult_Exp2Reg <= "000000" when (Asig(9) = '0' or Bsig(9) = '0') else
                not(Mult_ExpTmp(5)) & Mult_ExpTmp(4 downto 0);

EXPMULTREG: xb_expreg
  port map(Mult_Exp2Reg, XP_Clk, Mult_ExpS1);

-----------------------------------------------------------------
-- Multiplier - first 5 stages of the multiplication
-----------------------------------------------------------------
Mult_PP1(10) <= '0';
Mult_PP1(9 downto 0) <= Asig when Bsig(0) = '1' else Zero_10;

Mult_PA1 <= Asig when Bsig(1) = '1' else Zero_10;

SIGMULT1: xb_sigadd
  port map(Mult_PP1(10 downto 1), Mult_PA1,
           Mult_PP2( 9 downto 0), Mult_PP2(10) );

Mult_PA2 <= Asig when Bsig(2) = '1' else Zero_10;

SIGMULT2: xb_sigadd
  port map(Mult_PP2(10 downto 1), Mult_PA2,
           Mult_PP3( 9 downto 0), Mult_PP3(10) );

Mult_PA3 <= Asig when Bsig(3) = '1' else Zero_10;

SIGMULT3: xb_sigadd
  port map(Mult_PP3(10 downto 1), Mult_PA3,
           Mult_PP4( 9 downto 0), Mult_PP4(10) );

Mult_PA4 <= Asig when Bsig(4) = '1' else Zero_10;

SIGMULT4: xb_sigadd
  port map(Mult_PP4(10 downto 1), Mult_PA4,
           Mult_PP5( 9 downto 0), Mult_PP5(10) );

Mult_PA5 <= Asig when Bsig(5) = '1' else Zero_10;
```

```
SIGMULT5: xb_sigadd
  port map(Mult_PP5(10 downto 1), Mult_PA5,
           Mult_PP6( 9 downto 0), Mult_PP6(10) );

SIGMULTREG: xb_sigreg
  port map(Mult_PP6(10 downto 1), XP_Clk, Mult_PP6Reg );

SIGMULTREGA: xb_sigreg
  port map(Asig, XP_Clk, Mult_Mcand);

SIGMULTREGB: xb_sigreg
  port map(Bsig, XP_Clk, Mult_Mplier);

-----------------------------------------------------------------
-- Multiplier - last 4 stages of the multiplication
-----------------------------------------------------------------

Mult_PA6 <= Mult_Mcand when Mult_Mplier(6) = '1' else Zero_10;

SIGMULT6: xb_sigadd
  port map(Mult_PP6Reg, Mult_PA6,
           Mult_PP7(9 downto 0), Mult_PP7(10) );

Mult_PA7 <= Mult_Mcand when Mult_Mplier(7) = '1' else Zero_10;

SIGMULT7: xb_sigadd
  port map(Mult_PP7(10 downto 1), Mult_PA7,
           Mult_PP8( 9 downto 0), Mult_PP8(10) );

Mult_PA8 <= Mult_Mcand when Mult_Mplier(8) = '1' else Zero_10;

SIGMULT8: xb_sigadd
  port map(Mult_PP8(10 downto 1), Mult_PA8,
           Mult_PP9( 9 downto 0), Mult_PP9(10) );

Mult_PA9 <= Mult_Mcand when Mult_Mplier(9) = '1' else Zero_10;

SIGMULT9: xb_sigadd
  port map(Mult_PP9(10 downto 1), Mult_PA9,
           Mult_PP10(9 downto 0), Mult_PP10(10) );

-----------------------------------------------------------------
-- Final normalization of multiplication result
-----------------------------------------------------------------

Mult_SigRes <= Mult_PP10(10 downto 1) when Mult_PP10(10) = '1' else
```

```
                Mult_PP10( 9 downto 0);

  EXPMULTNORM: xb_expadd
    port map(Mult_ExpS1, Zero_6, Mult_PP10(10), Mult_ExpRes);

  ------------------------------------------------------------------
  -- Send back math results and register if the final result
  ------------------------------------------------------------------

  MathRslt <= Add_ExpRes & Add_SigRes when C_ResMult = '0' else
              Mult_ExpRes & Mult_SigRes;

  RESREG: xb_wordreg
    port map(MathRslt, XP_Clk, C_LoadRes, MdataOut);

  ------------------------------------------------------------------
  -- Memory address Generation
  ------------------------------------------------------------------

--  MemAdd <= "0000000000000001" when C_MemBound = '1' else
    MemAdd <= "0000000000000000";


  Mtemp(15 downto 3) <= XBar(33 downto 21);
  Mtemp( 2 downto 0) <= '0' & EndConst    when C_MemEnd = '1' else
                        '0' & FoConst     when C_MemFo  = '1' else
                        ConstNo           when C_MemCon = '1' else
                        XBar(20 downto 18);

  BOUNDADD: xb_boundadd
    port map(C_BoundOp, Mtemp, MemAdd, Maddr(15 downto 0));

  Maddr(17 downto 16) <= XBar(35 downto 34);

  ------------------------------------------------------------------
  -- Miscellaneous signal assignments
  ------------------------------------------------------------------

  Yes       <= '1';
  No     <= '0';
  Zero_6    <= "000000";
  Zero_10   <= "0000000000";
  C_WrSig   <= (C_Mwrite AND C_WrEn_D);
  C_BoundOp <= XP_Broadcast;

  ------------------------------------------------------------------
  -- Registers and resolution for off-chip signals
```

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

  XBINREG: xb_xbreg
    port map(XBarR, XP_Clk, XBar);

  MDATAINREG: xb_wordreg
    port map(MdataInR, XP_Clk, Yes, MdataIn);

  MADDRREG: xb_addreg
    port map(Maddr, XP_Clk, MaddrR);

  MRDREG: xb_bitreg
    port map(C_Mread, XP_Clk, Yes, C_MreadR);

  MWRREG: xb_bitreg
    port map(C_WrSig, XP_Clk, Yes, C_WrSigR);

  Pad_Input  (XP_XBar, XBarR);
  Pad_InOut  (XP_Mem_D, MdataOut, MdataInR, not(C_MreadR));

  Pad_Output (XP_Mem_A, MaddrR);
  Pad_Output (XP_Mem_RD_L, not(C_MreadR));
  Pad_Output (XP_Mem_WR_L, not(C_WrSigR));

  XP_Xbar_EN_L <= "00000";

  Pad_Input(XP_GOR_Valid, LEDSig);
  XP_LED <= not(LEDSig);

  XP_Left(15 downto 0) <= TriState(XP_Left(15 downto 0));
  LeftIn <= RB3VtoBV(XP_Left(15 downto 0));

  XP_Right(31 downto 16) <= TriState(XP_Right(31 downto 16));
  RightIn <= RB3VtoBV(XP_Right(31 downto 16));

  Pad_Output (XP_Left(31 downto 16),  LocalTemp);
  Pad_Output (XP_Right(15 downto  0), LocalTemp);

end heat_proc;
```

Appendix B

Control Processor Source Code

```
--------------------------------------------------------------------------------
--
-- Splash-2 Based Finite Difference Heat Transfer Analysis - Control PE
--
-- Copyright (C) 1996 by Kevin J. Paar and
-- Virginia Polytechnic Institute and State University
-- All rights reserved.
--
-- Do not distribute further without the concurrence of the authors.
--
--------------------------------------------------------------------------------

library SPLASH2;
use SPLASH2.TYPES.all;
use SPLASH2.SPLASH2.all;
use SPLASH2.COMPONENTS.all;
use SPLASH2.ARITHMETIC.all;

--------------------------------------------------------------------------------
--  Splash 2 Simulator v1.5 Xilinx_Control_Part Entity Declaration
--------------------------------------------------------------------------------
entity xilinx_control_part is
  Generic(
    BD_ID               : Integer := 0; --  Splash Board ID
    PE_ID         : Integer := 0  --  Processing Element ID
  );
  Port (
    X0_SIMD         : inout DataPath; --  SIMD Data Bus
    X0_XB_Data         : inout DataPath; --  Crossbar Data Bus
    X0_Mem_A         : inout MemAddr; --  Splash Memory Address Bus
    X0_Mem_D         : inout MemData; --  Splash Memory Data Bus
    X0_Mem_RD_L         : inout RBit3; --  Splash Memory Read Signal
(low-true)
    X0_Mem_WR_L : inout RBit3; --  Splash Memory Write Signal (low-true)
    X0_Mem_Disable : in  Bit; --  Splash Memory Disable Signal
    X0_GOR_Result_In    : inout RBit3_Vector(1 to XILINX_PER_BOARD);--
Global OR Result Bus
    X0_GOR_Valid_In : inout RBit3_Vector(1 to XILINX_PER_BOARD);--
```

```
Global OR Valid Bus
    X0_GOR_Result : out  Bit; --  Global OR Result Signal
    X0_GOR_Valid         : out  Bit; --  Global OR Valid Signal
    X0_Clk         : in  Bit; --  Splash System Clock
    X0_XBar_Set         : out  Bit_Vector(2 downto 0);--  Crossbar Set
Signals
    X0_X16_Disable : out   Bit; --  X16 Disable Signal
    X0_XBar_Send : out   Bit;
    X0_Int         : out  Bit; --  Interrupt Signal
    X0_Broadcast_In : in Bit; --  Broadcast Input Signal
    X0_Broadcast_Out   : out Bit; --  Broadcast Output Signal
    X0_Reset         : in Bit; --  Reset Signal
    X0_HS0                  : inout RBit3; --  Handshake Signal Zero
    X0_HS1                 : in    Bit; --  Handshake Signal One
    X0_XBar_EN_L : out   Bit;  --  Crossbar Enable (low-true)
    X0_LED         : out   Bit --  LED Signal
  );
end xilinx_control_part;

architecture x0_control OF xilinx_control_part IS
 --                                                    765432109876543210
  CONSTANT C_StateA : Bit_Vector(17 downto 0) := "100000000000000000";
  CONSTANT C_StateB : Bit_Vector(17 downto 0) := "100000000000000000";

  CONSTANT C_State1 : Bit_Vector(17 downto 0) := "101010000000000001";
  CONSTANT C_State2     : Bit_Vector(17 downto 0) := "100011000000000110";
  CONSTANT C_State3     : Bit_Vector(17 downto 0) := "100000000000001000";
  CONSTANT C_State4 : Bit_Vector(17 downto 0) := "100100001100010100";
  CONSTANT C_State5 : Bit_Vector(17 downto 0) := "000000010000100100";
  CONSTANT C_State6 : Bit_Vector(17 downto 0) := "010000000000000100";
  CONSTANT C_State7 : Bit_Vector(17 downto 0) := "100000000001000000";
  CONSTANT C_State8 : Bit_Vector(17 downto 0) := "100001000010000100";
  CONSTANT C_State9 : Bit_Vector(17 downto 0) := "100000100000000100";
  CONSTANT C_State10    : Bit_Vector(17 downto 0) := "100000000000000100";

  CONSTANT C_StateZ : Bit_Vector(17 downto 0) := "000000000000000000";

  CONSTANT Mem_A_Temp : Bit_Vector(1 downto 0) := "00";
  CONSTANT Mem_B_Temp : Bit_Vector(1 downto 0) := "01";
  CONSTANT Mem_Config : Bit_Vector(1 downto 0) := "10";
  CONSTANT Mem_Const : Bit_Vector(1 downto 0) := "11";

  CONSTANT Const_Conv : Bit_Vector(15 downto 0) := "0000000000000000";
  CONSTANT Const_End    : Bit_Vector(15 downto 0) := "0000000000001000";
  CONSTANT Const_Fo     : Bit_Vector(15 downto 0) := "0000000000010000";

  SIGNAL State : Bit_Vector(3 downto 0);
```

```
  SIGNAL Config : Bit;
  SIGNAL Top_Node : Unsigned(15 downto 0);
  SIGNAL Curr_Node : Unsigned(15 downto 0);
  SIGNAL Bot_Node : Unsigned(15 downto 0);
  SIGNAL Wrt_Node : Unsigned(15 downto 0);

  SIGNAL XB_Data : Bit_Vector(35 downto 0);
  SIGNAL MemBlock : Bit_Vector(1 downto 0);
  SIGNAL MemAddress : Bit_Vector(15 downto 0);
  SIGNAL CtrlData : Bit_Vector(17 downto 0);

  SIGNAL TempBank : Bit_Vector(1 downto 0);
  SIGNAL WrtBank : Bit_Vector(1 downto 0);
  SIGNAL IterNum : Unsigned(15 downto 0);
  SIGNAL IterCount : Unsigned(15 downto 0);
  SIGNAL EndNode : Unsigned(15 downto 0);
  SIGNAL CycleWidth     : Unsigned(15 downto 0);
  SIGNAL CycleEnd : Unsigned(15 downto 0);

  SIGNAL LMem_RD_L  : Bit;
  SIGNAL LMem_WR_L : Bit;
  SIGNAL LMem_A : Bit_Vector(17 downto 0);
  SIGNAL LMem_D : Bit_Vector(15 downto 0);
  SIGNAL GOR_Result : Bit;
  SIGNAL Done : Bit;
  SIGNAL ProcMap : Bit;

begin  --  X0_Control

  Curr_Node <= Top_Node  + CycleWidth;
  Bot_Node  <= Curr_Node + CycleWidth;

  process

    VARIABLE NewState : Bit_Vector(3 downto 0);

  begin
    wait until X0_Clk'Event and X0_Clk = '1';

    if (Config = '0') then
      CASE State IS
        WHEN "0000" =>
          LMem_A    <= "000000000000000000";
          LMem_RD_L <= '0';

          Top_Node  <= "0000000000000000";
          IterCount <= "0000000000000000";
```

```
        WrtBank   <= "00";
        Wrt_Node   <= "0000000000000000";
        TempBank  <= Mem_A_Temp;
        ProcMap   <= '0';

        NewState := "0001";

      WHEN "0001" =>
        CycleWidth <= LMem_D;

        LMem_A    <= "00000000000000001";
        LMem_RD_L <= '0';

        NewState := "0010";

      WHEN "0010" =>
        EndNode <= LMem_D;

        LMem_A    <= "00000000000000010";
        LMem_RD_L <= '0';

        NewState := "0011";

      WHEN "0011" =>
        IterNum   <= LMem_D;

        LMem_RD_L <= '1';

        NewState := "0000";
        Config <= '1';

      WHEN Others => NULL;

    end CASE;

    MemBlock   <= Mem_A_Temp;
    MemAddress <= "0000000000000000";
    CtrlData   <= C_StateZ;

  else
    CASE State IS

      ------------------------------------------------
      -- State A:
      --   Setup for read of local temp data
      ------------------------------------------------
      WHEN "0000" =>
```

```
  MemBlock   <= TempBank;
  MemAddress <= Curr_Node;
  CtrlData   <= C_StateA;

  NewState := "1110";

-----------------------------------------------
-- State B:
--   Setup for read of config data
-----------------------------------------------
WHEN "1110" =>
  MemBlock   <= Mem_Config;
  MemAddress <= Curr_Node;
  CtrlData   <= C_StateB;

  NewState := "0001";

-----------------------------------------------
-- State 1:
--   Setup for read of bound data
--   Latch local temp data
-----------------------------------------------
WHEN "0001" =>
  MemBlock   <= TempBank;
  MemAddress <= Curr_Node;
  CtrlData   <= C_State1;

  NewState := "0010";

-----------------------------------------------
-- State 2:
--   Setup for read of top data
--   Latch config data
-----------------------------------------------
WHEN "0010" =>
  MemBlock   <= TempBank;
  MemAddress <= Top_Node;
  CtrlData   <= C_State2;

  NewState := "0011";

-----------------------------------------------
-- State 3:
--   Setup for read of bot data
--   Latch bound data
-----------------------------------------------
WHEN "0011" =>
```

```
  MemBlock   <= TempBank;
  MemAddress <= Bot_Node;
  CtrlData   <= C_State3;

  NewState := "0100";

-------------------------------------------------
-- State 4:
--   Setup for read of conv data
--   Latch top data
-------------------------------------------------
WHEN "0100" =>
  MemBlock   <= Mem_Const;
  MemAddress <= Const_Conv;
  CtrlData   <= C_State4;

  NewState := "0101";

-------------------------------------------------
-- State 5:
--   No memory operation
--   Latch bot data
-------------------------------------------------
WHEN "0101" =>
  MemBlock   <= TempBank;
  MemAddress <= "0000000000000000";
  CtrlData   <= C_State5;

  NewState := "0110";

-------------------------------------------------
-- State 6:
--   Setup for write of new local temp data
--   Latch conv data
-------------------------------------------------
WHEN "0110" =>
  if (WrtBank = Mem_A_Temp) then
    MemBlock <= Mem_B_Temp;
  else
    MemBlock <= Mem_A_Temp;
  end if;

  MemAddress <= Wrt_Node;
  CtrlData   <= C_State6;

  if (IterCount = IterNum) then
    NewState := "1011";
```

```
    else
      NewState := "0111";
    end if;

-----------------------------------------------
-- State 7:
--   Setup for read of end data (data write)
-----------------------------------------------
WHEN "0111" =>
    MemBlock   <= Mem_Const;
    MemAddress <= Const_End;
    CtrlData   <= C_State7;

    NewState := "1000";

-----------------------------------------------
-- State 8:
--   Setup for read of fo data
-----------------------------------------------
WHEN "1000" =>
    MemBlock   <= Mem_Const;
    MemAddress <= Const_Fo;
    CtrlData   <= C_State8;

    Wrt_Node   <= Curr_Node;

    if (Curr_Node = EndNode) then

      Top_Node  <= "0000000000000000";

      if (TempBank = Mem_A_Temp) then
        TempBank <= Mem_B_Temp;
      else
        TempBank <= Mem_A_Temp;
      end if;

      IterCount <= IterCount + 1;

    else

      Top_Node   <= Top_Node + 1;
      WrtBank    <= TempBank;

    end if;

    ProcMap <= not(ProcMap);
```

```
  NewState := "1001";

-------------------------------------------------
-- State 9:
--   Setup for read of local temp data
--   Latch end data
-------------------------------------------------
WHEN "1001" =>
  MemBlock   <= TempBank;
  MemAddress <= Curr_Node;
  CtrlData   <= C_State9;

  NewState := "1010";

-------------------------------------------------
-- State 10:
--   Setup for read of config data
--   Latch fo data
-------------------------------------------------
WHEN "1010" =>
  MemBlock   <= Mem_Config;
  MemAddress <= Curr_Node;
  CtrlData   <= C_State10;

  NewState := "0001";

-------------------------------------------------
-- State X:
--   Ending cycle number 1
-------------------------------------------------
WHEN "1011" =>
  MemBlock   <= Mem_A_Temp;
  MemAddress <= "0000000000000000";
  CtrlData   <= C_StateZ;

  NewState := "1100";

-------------------------------------------------
-- State Y:
--  Ending cycle number 2
-------------------------------------------------
WHEN "1100" =>
  MemBlock   <= Mem_A_Temp;
  MemAddress <= "0000000000000000";
  CtrlData   <= C_StateZ;

  NewState := "1111";
```

```
        ------------------------------------------------
        -- State Z:
        --   Ending cycle number 3
        ------------------------------------------------
        WHEN "1111" =>
          MemBlock   <= Mem_A_Temp;
          MemAddress <= "0000000000000000";
          CtrlData   <= C_StateZ;

          Done        <= '1';

          NewState := "1111";

        ------------------------------------------------
        -- Catch all state
        ------------------------------------------------
        WHEN OTHERS => NULL;

      end CASE;

      LMem_RD_L <= '1';

    end if;

--    if (X0_Reset = '1') then
--      State  <= "0000";
--      Config <= '0';
--    else
      State <= NewState;
--    end if;

  end process;

  X0_GOR_Result  <= not(Done);
  X0_LED <= Done;

  X0_X16_Disable <= '1';
  X0_XBar_Send <= '1';
  LMem_WR_L           <= '1';

  X0_Broadcast_Out      <= ProcMap;

  XB_Data <= MemBlock & MemAddress & CtrlData;

  Pad_Output(X0_Mem_RD_L, LMem_RD_L);
  Pad_Output(X0_Mem_WR_L, LMem_WR_L);
```

```
  Pad_Output(X0_Mem_A, LMem_A);
  Pad_Input(X0_Mem_D,LMem_D);

  Pad_Output(X0_XB_Data, XB_Data);

end x0_control;
```

VITA

Kevin J. Paar was born on February 14, 1971 in Manhassett, NY. Kevin grew up in upstate New York and attended school in the town of Dryden. During his sophomore year of high school, Kevin moved to Ashton, MD and continued school there. He graduated from Sherwood High School in Sandy Spring, MD in 1989, after which he attended college at Virginia Tech. Kevin graduated from Virginia Tech in May, 1993 with a B.S. in Computer Engineering.

After completing his undergraduate degree, Kevin began work with York International Corp. in York, PA. In August, 1994 Kevin returned to Virginia Tech while continuing work for York International on a part-time basis. In May, 1996 he graduated from Virginia Tech with a M.S. in Electrical Engineering.

In July, 1996 Kevin plans to begin work with the Hewlett Packard Corporation in Ft. Collins, CO. There Kevin will be designing graphics-acceleration ASICs for use in HP workstations.