A RECONFIGURABLE MULTI-PROCESSOR CUSTOM INTEGRATED CIRCUIT

FOR FLOATING POINT INTENSIVE ALGORITHMS

by

SHRIRANG MADHAV YARDI

(Under the Direction of JEFFREY W. SMITH)

ABSTRACT

Hardware support for floating-point intensive applications like multimedia and graphics processing is rapidly gaining importance. The use of add-on cards for graphics processing, image rendering and 3D acceleration are now common even in home PCs. Recently, reconfigurable systems, in the form of FPGAs, have gained popularity in construction of such specialized hardware. In this work, we take a look at a particular class of applications involving highly computationally-intense iterative floating-point algorithms for solving large sparse linear systems and present a case for achieving improvement in their execution by using a Custom Integrated Circuit. Our design uses novel methods to overcome precision and numerical instability problems that arise in earlier implementations which used FPGAs. Our design consists of an add-on card which contains a custom Reconfigurable Integrated Circuit with multiple processors on a single die. We believe that our system is the first of its kind to use these algorithms for solving large sparse linear systems in real-time. The cost-effectiveness and computational power provided by our system will allow applications like real-time object modeling, fluid flow modeling and animation to be run even on home PCs by using this add-on card.

INDEX WORDS: floating-point intensive, custom integrated circuit (IC), physically-based modeling, reconfigurable system, standard cell-based VLSI design flow
A RECONFIGURABLE MULTI-PROCESSOR CUSTOM INTEGRATED CIRCUIT
FOR FLOATING POINT INTENSIVE ALGORITHMS

by

SHRIRANG MADHAV YARDI

B.E., Fr. C. Rodrigues College of Engineering, University of Mumbai, India, 2000

A Thesis Submitted to the Graduate Faculty of The University of Georgia in Partial
Fulfillment of the Requirements for the Degree

MASTER OF SCIENCE

ATHENS, GEORGIA

2003
A RECONFIGURABLE MULTI-PROCESSOR CUSTOM INTEGRATED CIRCUIT

FOR FLOATING POINT INTENSIVE ALGORITHMS

by

SHRIRANG MADHAV YARDI

Major Professor: Jeffrey W. Smith
Committee: Thiab Taha
Khaled Rasheed

Electronic Version Approved:

Maureen Grasso
Dean of the Graduate School
The University of Georgia
August 2003
ACKNOWLEDGEMENTS

First, I would like to thank Dr. Bishop and Dr. Kelliher for giving me a chance to work on this project. Their support has been vital for the completion of this project. I would also like to thank Dr. Smith for his suggestions, critique and encouragement during the course of this project.

Special thanks go to the customer support team at MOSIS. With their help, I was able to avoid many pitfalls during the placement and routing stage of the design. I would also like to thank Mr. Chenbo Liu from the Shanghai National Integrated Circuit Design Center and numerous members of the comp.cad.cadence discussion group for their valuable suggestions.
## TABLE OF CONTENTS

<table>
<thead>
<tr>
<th>ACKNOWLEDGEMENTS</th>
<th>iv</th>
</tr>
</thead>
<tbody>
<tr>
<td>LIST OF TABLES</td>
<td>vii</td>
</tr>
<tr>
<td>LIST OF FIGURES</td>
<td>viii</td>
</tr>
</tbody>
</table>

### CHAPTER

1. INTRODUCTION                         .............................................................. 1

2. BACKGROUND                          .............................................................. 4
   2.1 Introduction                       ............................................................. 4
   2.2 Motion Simulation Basics           .......................................................... 5
   2.3 Basics of Ordinary Differential Equations  ........................................ 6
   2.4 Dynamics of Particle Systems       .......................................................... 11
   2.5 Acceleration Using Special Purpose Hardware ..................................... 14
   2.6 Summary                           ................................................................. 17

3. RELATED WORK AND MOTIVATION         ......................................................... 19
   3.1 Introduction                       ................................................................. 19
   3.2 Related Specialized Systems        ............................................................ 20
   3.3 A First Generation System for Deformable Object Modeling ............... 24
   3.4 The Second Generation System       ........................................................... 26
   3.5 Summary                           ................................................................. 28
ARCHITECTURE OF THE CUSTOM INTEGRATED CIRCUIT

4.1 Overview

4.2 Instruction Set Architecture

4.3 Components of a Single Processing Element

4.4 Data and Control Flow within a Single Processing Element

4.5 Summary

A STANDARD CELL-BASED VLSI DESIGN FLOW OF THE CUSTOM INTEGRATED CIRCUIT

5.1 Introduction

5.2 Overview of the Implementation

5.3 System Specification and Simulation

5.4 Logic Synthesis and Optimization

5.5 Placement and Routing

5.6 Adding Bonding Pads

5.7 Tape-Out

5.8 Summary

CONCLUSIONS AND FUTURE WORK

REFERENCES

APPENDICES

A FLOATING POINT REPRESENTATION

B DESIGN HIERARCHY OF THE VHDL REPRESENTATION
LIST OF TABLES

Table 4.1: Compute Mode Instructions .................................................................34
Table 4.2: Override Mode Instructions .................................................................36
Table 4.3: Data-Table for 128x32 Operand Memory and 128x40 Control Memory
generated by the Artisan Memory Compiler .......................................................39
Table 4.4: Fields of the Override Mode Word .....................................................40
Table 4.5: Fields of the Control Memory Word ...................................................40
Table 4.6: External I/O Signals .............................................................................43
Table 4.7: Status Signals .......................................................................................44
Table 4.8: Operand and Control Memory Signals .................................................46
Table 4.9: Control Signals for the Functional Unit ...............................................47
Table 4.10: Program Counter, Communication Unit and Output Unit Control Signals....48
Table A.1: IEEE Single-Precision Format Bit-Field Interpretation .......................87
Table A.2: Range of single precision representation ............................................89
## LIST OF FIGURES

<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Figure 2.1</td>
<td>Example of Cloth Animation</td>
<td>5</td>
</tr>
<tr>
<td>Figure 2.2</td>
<td>Examples of Ray-Traced Images</td>
<td>5</td>
</tr>
<tr>
<td>Figure 2.3 (a)</td>
<td>Path Traced by Euler Integration</td>
<td>9</td>
</tr>
<tr>
<td>Figure 2.3 (b)</td>
<td>Inaccuracy due to approximation in Euler Integration</td>
<td>9</td>
</tr>
<tr>
<td>Figure 2.4</td>
<td>Mass Spring Model for Free-Form Deformation</td>
<td>12</td>
</tr>
<tr>
<td>Figure 2.5</td>
<td>Example of Collision Detection</td>
<td>15</td>
</tr>
<tr>
<td>Figure 3.1</td>
<td>The GRAPE-6 Processor Board with 8 chips</td>
<td>20</td>
</tr>
<tr>
<td>Figure 3.2</td>
<td>Block Diagram of a single GRAPE-6 chip</td>
<td>20</td>
</tr>
<tr>
<td>Figure 3.3</td>
<td>Object-Parallel Rendering by Using Image Composition</td>
<td>22</td>
</tr>
<tr>
<td>Figure 3.4</td>
<td>SPARTA Implementation</td>
<td>24</td>
</tr>
<tr>
<td>Figure 3.5</td>
<td>Spring Force Computation Pipeline</td>
<td>24</td>
</tr>
<tr>
<td>Figure 3.6</td>
<td>A Second Generation System for Floating Point Intensive Algorithms</td>
<td>27</td>
</tr>
<tr>
<td>Figure 4.1</td>
<td>Organization of a production PCB with a PCI Host Interface</td>
<td>29</td>
</tr>
<tr>
<td>Figure 4.2</td>
<td>High-level Organization of a single IC</td>
<td>31</td>
</tr>
<tr>
<td>Figure 4.3</td>
<td>Streaming of Override Mode Instructions</td>
<td>33</td>
</tr>
<tr>
<td>Figure 4.4</td>
<td>Compute Mode Instruction Format</td>
<td>34</td>
</tr>
<tr>
<td>Figure 4.5</td>
<td>Override Mode Instruction Format</td>
<td>36</td>
</tr>
<tr>
<td>Figure 4.6</td>
<td>Datapath of a single PE, Parts I &amp; II</td>
<td>41, 42</td>
</tr>
<tr>
<td>Figure 5.1</td>
<td>Top-down (standard cell-based) Design Methodology</td>
<td>54</td>
</tr>
<tr>
<td>Figure 5.2</td>
<td>NC-VHDL Design Flow</td>
<td>58</td>
</tr>
<tr>
<td>Figure 5.3</td>
<td>NC-ELAB Design Flow</td>
<td>59</td>
</tr>
</tbody>
</table>
CHAPTER 1
INTRODUCTION

There are interesting problems whose solutions require computationally intensive iterative floating point algorithms. Such algorithms are used in a variety of application areas ranging from graphics processing and animation to mechanical modeling and even simulations to predict interaction between galactic objects. A common characteristic of these algorithms is the requirement for massive computational power, which creates an interest in using special purpose hardware to accelerate these applications. Recently, a tremendous amount of effort has gone into building such special-purpose systems, ranging from simple architectural and instruction set extensions for existing processors to using reconfigurable and custom integrated circuits. In this work, we focus on a custom special-purpose hardware implementation targeted towards applications which involve floating point intensive iterative algorithms for solving very large sparse linear systems [1].

Fast linear system solvers are gaining importance in a number of application areas like fluid flow modeling, physical modeling of rigid and deformable objects, animation, mechanical modeling, ray tracing and image rendering, to name a few. The high performance of such systems is of particular interest to the computer graphics community where problems like physically-based modeling of objects can be reduced to the problem of solving a large sparse linear system [2, 3, 4, 5, 6, 7]. Physical modeling and simulation provide powerful tools for creating very realistic animations and have been successfully used in numerous animated feature films. Shrek [8] and Finding Nemo [9] are two recent animated feature films which demonstrate the quality of animations that can be achieved using these methods.
Recently, a lot of research has been directed towards making these applications interactive [10]. This requires the back-end processor running these applications to provide real-time performance. The computational needs of such applications far exceed those provided by the general purpose CPUs available in the market today. Hence there has been increased interest in special purpose architectures that can be cost-effective while providing the required computational power needed for real-time performance. This work presents the design and implementation of such an architecture: a dedicated hardware system for solving large sparse linear systems.

The aim of the proposed architecture is to provide very high performance for floating point intensive iterative algorithms at a low cost. Our system is designed for applications having a large number of simple, but intensely computational, iterations with a significant opportunity for parallelism. The architecture exploits this inherent parallelism by unrolling each iteration in hardware and executing the operations in parallel to achieve speed-up. Additionally, many different algorithms can be supported by reconfiguring the system using a host machine to provide the required instructions and data via a parallel interface. Our system is designed to be scalable so it can handle large problem sizes and provide the required computational power.

Below we present the architecture and the VLSI implementation of this system. A considerable part of this document is devoted to the details about the implementation with a standard cell-based VLSI design flow using CAD tools from the Cadence Tool Suite [11]. The chip is currently being fabricated using the 0.18 µm process technology by MOSIS [12]. The performance evaluation of algorithms to be run on the system is not a part of this work and is a topic of further research in this area.

The rest of the document is organized as follows: Chapter 2 provides some background information about the kind of applications targeted by the system. We take a detailed look at one such application - physically-based modeling of deformable objects. We present two existing methods for solving this problem, viz. explicit integration and implicit integration. Explicit
methods have been popular in software implementations since they involve considerably less computations. However, explicit methods suffer from serious numerical stability problems which make them unsuitable for interactive simulations. Our system has been specifically designed to meet the high computational need of implicit methods. We think that this system is the first of its kind for solving large sparse linear systems using implicit integration techniques.

Chapter 3 takes a look at some related specialized systems for floating point intensive applications. Specifically, we take a detailed look at a first-generation system for deformable object modeling called Simulation of Physics on a Real Time Architecture (SPARTA) [13, 14]. We also explain how the lessons learned during the implementation of SPARTA were helpful in making several design choices of our system.

Chapters 4 and 5 present the tools and methods used to implement the proposed system. Specifically, Chapter 4 includes the details about the instruction set architecture, design of various units and the control and data flow within the system while Chapter 5 describes the standard-cell based VLSI design flow followed to realize the system from its HDL representation to its mask layout.
CHAPTER 2

BACKGROUND

2.1 Introduction

A number of algorithms involving large sparse linear systems with highly iterative, floating point intensive algorithms are of growing interest in various application domains. Hardware solutions for efficient and fast execution of such algorithms are gaining popularity even with chip giants like Intel and AMD [15, 16]. Some areas where such algorithms are encountered include fluid flow modeling, mechanical modeling, and most importantly, physical modeling of objects for animation and interactive simulation [17]. Physical modeling is an especially attractive application domain for the motion picture industry where it is used to model characters in animation movies like Shrek [8] and Finding Nemo [9].

The computational requirements for these applications are too high to be met by available general-purpose CPUs. Researchers are now looking at special-purpose architectures that can provide the required computational power and can be orders of magnitude faster than general purpose systems while being more cost-effective and having a shorter time-to-market.

This chapter provides the background information needed to understand the motivation behind our design. Specifically, we take a detailed look at one application, viz. physical modeling of deformable objects, and explain how we can use a special-purpose hardware system to provide the required computational power and speed to make such applications interactive at a low cost.
2.2 Motion Simulation Basics

Motion simulation involves calculation of acceleration, velocity and position of objects under the influence of certain forces, either in 2D or 3D space. In the context of computer graphics, such objects are represented by physical models, and animation techniques based on certain laws of physics are then applied to these models to simulate their behavior (their motion and response to external and internal forces) as they move over a series of frame updates [13]. Such an animation technique is referred to as the physical modeling of the object of interest. A detailed look at some of the current research in this area is provided in [4, 10, 18]. Figure 2.1 shows an image generated using physical modeling techniques to model cloth movements.

Figure 2.1. Example of Cloth Animation from [18]

Figure 2.2. Examples of ray-traced images from [19]
In addition to solid objects, there is also an interest in realistic animation of fluids and gases [20] which is much more complex. Another highly computationally intensive application is ray tracing which is used for realistic rendering of images by modeling the behavior of individual light rays in a 3D scene. Ray tracing is also used for realistic modeling of shadows, reflections and lighting during animation. Figure 2.2 shows examples of ray traced images which illustrates the high quality of such images. A common characteristic of these applications is their intensely computational and iterative nature. In addition, using certain algorithms, these applications can be reduced to very large sparse linear systems requiring fast linear system solvers.

Simulation of a system of objects can be classified as an initial value problem where the position of each object in the universe is known at simulation time \( t = 0 \). The simulation problem can be formulated as the problem of calculating the state (position, velocity and acceleration) of the system (collection of objects) as a function of time. The problem can be represented using a set of Ordinary Differential Equations (ODE’s, Section 2.3) and each simulation step involves solving this system of ODE's to find the state as time progresses. In this work, we concentrate on the physical modeling of deformable objects.

In this chapter, we provide the basics of physical modeling of objects and outline the numerical methods for simulation of such systems. We first provide the characteristics, advantages and disadvantages of these methods and then present a case for using special-purpose hardware and hardware/software co-design for solving these problems in real-time.

2.3 Basics of Ordinary Differential Equations

An equation describing the relation between a function and its derivatives is known as a Differential Equation [21]. For example, the equation governing the velocity of an unrestricted sliding mass subject to the frictional effects and Newton's 2\(^{nd}\) law of motion can be given by

\[
m \cdot \frac{dv}{dt} = -D \cdot v(t)
\]
where \( m \) is the mass of the body, \( D \) is the damping coefficient of the surface, \( v \) is the velocity and \( dv/dt \) gives the derivative of \( v \) with respect to time \( t \). All the derivatives in this equation are only with respect to the time \( t \). Such equations where differentiation in the entire equation is with respect to a single variable are called *Ordinary Differential Equations (ODE's)*.

Many models in physics, engineering, economics *etc.* fall within the category of differential equations. We are primarily concerned with a particular class of problems called *Initial Value Problems*. In an initial value problem, the information about the solution is given at a single point. The behavior of such a system can be expressed by an ODE given as

\[
\frac{dx}{dt} = f(x, t)
\]

where, \( x \) is the state of the system at a given point, \( dx/dt \) is the derivative of \( x \) with respect to the quantity \( t \) and \( f \) is a known function. The initial state value is given by \( x(t_0) = x_0 \) and the equation follows \( x \) over \( t \) thereafter. If \( t \) is the time elapsed and \( x(t) \) is the velocity of the particle, then the above equation sweeps out a curve in the 2D space that describes the motion of any particle 'p' in the 2D space [22].

Mass-spring simulation is the dominant method of deformable object simulation. In mass-spring simulation, an object is described in terms of mass points connected by springs [22]. The simulation involves a system of ODE's which describe the relation of point acceleration, velocity and position based on Hooke's Law. We are interested in the numerical methods to solve such a system of ODE's. In this work, we consider the *Explicit* and *Implicit methods* and outline the advantages and disadvantages of each with the point of view of implementing them in hardware.
2.3.1 Explicit Methods - Euler Integration

The simplest method is called Euler integration [21, 22]. This method recursively provides estimates of $x(t)$ in progressive steps (of the variable $t$) starting from an initial value for $x(t)$ given by $x(t_0) = x_0$. Let the estimate at a later time, $(t_0 + h)$ be denoted as $x(t_0 + h)$ where $h$ is known as the stepsize parameter. Euler's method computes an estimate of $x(t_0 + h)$ by simply taking a step in the direction of the derivative and scaling it by the stepsize. Thus

$$x(t_0 + h) = x_0 + h \cdot [dx/dt(t_0)]$$

In general, the estimate of $x_{j+1}$ is given by

$$x_{j+1} = x_j + h \cdot [dx/dt(t_j)]$$

Notice that no differentiation is required to compute the estimate. Thus, in 2D space, if Euler's method is used to compute the motion of a particle 'p', the solution will trace out a polygonal path which approximates the real integral curve.

The main advantages of Euler's method is that it provides for a fast, easy computer implementation and each iteration requires little computation since its cost per step, i.e. the number of derivative evaluations per step, is one.

The major problem with explicit integration is its inaccuracy and instability. Also, since Euler's method consists only of one derivative evaluation per iteration, it results in a large truncation error that is linearly dependent upon the stepsize. Hence the simulation quickly diverges for large timesteps. Damping and adaptive selection of stepsize are two methods to reduce this problem, but these cause the simulation to appear to be unrealistic. The truncation error can be made small by choosing a very small step size which results in a very large number of iterations and renders the method ineffective in software. Figure 2.3(a) illustrates the path traced by Euler’s method. Figure 2.3(b) illustrates the inaccuracy of the Euler’s method when tracing the path of a particle which, ideally, is moving along a circle. Thus, the faint circles represent the actual integral curve that the particle follows starting from the center. The bold line illustrates the path approximated by the Euler’s method which can be seen as an expanding spiral.
The thick bold line plots the divergence of the Euler path from the actual integral curve. The divergence, which increases considerably with each step, results in a large truncation error.

![Graph showing divergence between Euler's Method path and actual integral curve.](image)

(a) Path traced by the Euler’s Method
(b) Increasing divergence due to approximation

Figure 2.3

### 2.3.2 Implicit Methods

The main bottle-neck in explicit integration is that the time-steps must be small to avoid numerical instability. Explicit methods are especially ill-suited for solving "stiff" differential equations [23]. When using explicit methods for solving stiff ODE's, the stepsize parameter needs to be very small in order to avoid instability. This results in the simulation being excruciatingly slow and, although the variables may satisfy all the constraints, the solution is not feasible for real-time or interactive frame updates. In the mass-spring model, a deformable object is modeled as a collection of mass points connected by springs. The motion of mass points is constrained by the compression and decompression forces of the springs as given by Hooke's Law. The mass-spring model is a classic example of stiff ODE's and hence their simulation using explicit methods is unstable for large steps.

The main difference between the two methods is that in explicit integration, the derivative evaluation step is based solely on the conditions at the start of the time interval while in the implicit method, the step depends on the conditions at the end of that step itself. In other
words, the derivative evaluation step in implicit integration consists of a calculation which takes
the simulation one step "backward" in time. For example, to calculate $x(t_0 + h)$, Euler's method
blindly scales the derivative of $x$ at time $t_0$. An implicit method, or backward Euler integration,
calculates an output step whose derivative at least points back (with a backward Euler step) to the
conditions existing at the previous step, thus providing a sanity-check against run-away
divergence.

The main disadvantage of implicit integration is the computation cost per step. This has
been the main reason why implicit methods have not been widely used for motion simulation and
much of the research has been targeted at improving explicit methods. In [18], Baraff et. al.
present a modified implicit integration method which makes it possible to use large step sizes for
cloth simulation. Specifically, Baraff et. al [18] use a modified Conjugate Gradient Method that
generates large sparse linear systems which are then solved to advance the simulation.

A thorough proof and analysis of the Conjugate Gradient Method (CG) is difficult and
beyond the scope of this work. Basics and details about CG and how they are useful in solving
stiff ODE's can be found in [22, 24]. CG provides a fast, memory-efficient, iterative method for
solving the large sparse matrices (for example, in rendering and animation techniques). It will
suffice here to say that CG is a highly iterative method which is useful for solving linear systems
of equations which can be represented as

$$A \cdot x = b$$

where $x$ is an unknown vector, $b$ is a known vector and $A$ is a known square, symmetric, positive-
definite sparse matrix. Every iteration in CG involves intense computation of floating point
numbers which make the sparse linear system solver the main bottle-neck for high performance
of such systems. A special-purpose architecture dedicated to solving large sparse linear systems
would provide considerable speed-up.
2.4 Dynamics of Particle Systems

Particles are objects in 2D or 3D space that have mass, position, velocity and respond to forces, but have no spatial dimension. Since they are simple, particles are the easiest objects for motion simulation. Using particles as basic building blocks, we can build larger objects that typically fall into two categories: rigid bodies and deformable objects.

2.4.1 Rigid Bodies

A rigid body can be defined as a collection of mass points such that the distance between any two mass points is constant over time, i.e. there are no internal degrees of freedom. A rigid body occupies volume and has a particular shape. We define the center of mass of a rigid body as a mass point that lies at its geometric center. The motion of a rigid body can be reduced to the translation of the center of mass and rotation about it.

Interactive simulation of rigid bodies can be implemented efficiently due to the very nature of their definition. For example, the moment of inertia matrix is typically pre-computed to speed up object motion adjustment. It is also possible to use methods like rotation of co-ordinates in such a way that the moment of inertia matrix reduces to a diagonal matrix in the new co-ordinate system which results in simple motion equations. Collision detection/resolution is a more complex problem but can be handled in linear time using bounding boxes [25, 26, 27].

Rigid body simulation, however, cannot model the range of objects that a user would expect to interact in a realistic environment [13]. Soft, bending objects like clouds, fluids, cloth, Jell-O etc. cannot be modeled. It is difficult to model objects that can fracture upon collision. Consequently, modeling of collision and its effects is not very realistic. Due to these limitations, physical modeling of deformable objects has gained popularity in recent years.
2.4.2 Deformable Objects

Modeling of deformable objects requires significantly more computation but the advantage is that a wide range of objects can be modeled and the simulations can be very realistic. Deformable objects are modeled as non-rigid collection of mass points connected by damped springs [22]. Figure 2.4 illustrates this model which is known as the Mass-spring model of deformable objects. Significant amount of computation is required due to the presence of both, internal and external degrees of freedom. Real-time mass-spring simulation of such objects typically involves three steps: spring force computation, collision detection pruning and exact collision detection [28].
2.4.2.1 Spring Force Computation

The spring force computation step determines the velocity increments for each mass point as a result of the connected springs exerting displacement and damping forces on it. The response of these springs due to compression/decompression is given by Hooke's law. The spring forces between any two mass points at positions \( a \) and \( b \) are given by

\[
fa = -\left[ k_s \cdot (|l| - r) + k_d \cdot \left( \frac{dl/dt \cdot l}{(|l|)^2} \right) \right] \cdot \frac{l}{|l|},
\]

\[
b = f_a
\]

where \( f_a \) and \( f_b \) are the forces on \( a \) and \( b \) respectively, \( l = (a - b) \), is the distance between the two mass points, \( r \) is the rest length of the spring, \( k_s \) is the spring constant (stiffness of the spring) and \( k_d \) is the damping constant. The main challenge in spring force computation is minimizing the instability in the approximation. The stepsize parameter and object stiffness have to be carefully selected to get good results. Explicit methods are especially sensitive to these parameters and are prone to instability for large stepsizes.

2.4.2.2 Collision Detection Pruning

Collision detection and the computation of resulting contact forces is the most computationally intensive part in the simulation process. Checking every object against every other object in the object space for collisions has a complexity of the order \( N^2 \) where \( N \) is the total number of objects. Hence this "brute-force" method is rarely used in practice and many collision detection systems use a two-level approach. The first level consists of pruning away the uninteresting objects, for example objects which are too far away or on non-intersecting trajectories or which are not likely to collide for the next \( 'n' \) steps [27]. The second level consists of checking for collision only between the likely objects by checking specific polygons between objects called "bounding boxes". Since this phase is not as computationally intense as other steps, pruning can be carried out on the host CPU or by using a simple embedded processor.
2.4.2.3 Exact Collision Detection and Resolution

Ideas and techniques used for rigid body simulation cannot be extended for deformable objects. In rigid body simulations, objects are assumed to be convex and exact collision detection consists of determining only if the point is contained within each plane defining the object. This approach cannot be used for deformable objects.

A new approach suitable for deformable objects was proposed by Bishop et. al. in [13, 28]. The approach defines "collision detection lines" or cdlines, where each mass point in the object is associated with a line segment in the object. These lines can be visualized as spokes emanating from the interior of the object. The collision detection problem is now reduced to determining when these cdlines intersect a face. Figure 2.5 illustrates this approach with an example from [28].

The final step is to resolve collisions once they are detected. This is accomplished by changing the velocities of the points involved in the collision along the face normal (to model elastic collisions) and along the face (to model realistic friction behavior) [13].

2.5 Acceleration Using Special Purpose Hardware

Recently, even big companies like Intel and AMD have recognized the importance of hardware and software support for floating point computations for multimedia and interactive graphics processing applications. This has resulted in systems like Intel MMX [29] which consists of ISA extensions to support parallel execution of some operations, 3DNow! [15] which accelerates floating-point operations early in the graphics processing pipeline and KNI (Katmai New Instructions) [14] which offers SIMD (Single Instruction Multiple Data) operations on single precision floating point values. We take a detailed look at some related research in this area in Chapter 3.
Figure 2.5. An example of Collision Detection from [28]. Notice how the object is modeled in terms of cdlines for collision detection.

The problem of interactive deformable object modeling provides unique features like processing of large amounts of data with similar computations in successive iterations, floating-point intensive loops and opportunity for a pipelined organization for high utilization of hardware resources. Specifically, modeling using the mass-spring system involves solving of a large sparse system of ODE's. The computational requirements for realistic simulations are too high for general-purpose processors. Software methods cannot achieve the desired performance in real-time when small stepsizes are selected for the simulation. Other applications which fall in a similar category are interactive fluid dynamics modeling and interactive ray tracing. The common
characteristics of these algorithms which make them suitable for a special-purpose hardware implementation are:

- need for processing of large amounts of data
- large number of floating point intensive computations
- relatively simple iterations and similar computations over successive iterations
- significant opportunities to exploit parallelism

We also expect other enhancements by using special-purpose hardware for such applications as opposed to execution on general-purpose CPUs. These include:

- **Better Algorithms:** Explicit integration methods are currently used because of their simplicity and low computation cost per step. An important bottle-neck in using these methods is that the time steps need to be carefully selected to avoid instability. Large steps cause the simulation to quickly diverge due to truncation errors. Special-purpose hardware can minimize these problems by using implicit integration methods which allow for large time steps. Implementing algorithms for implicit integration on general-purpose processors is very difficult due to computational complexity involved. Special-purpose hardware will not only result in speed-up for such computations but also allow the simulations to be realistic.

- **Speed:** General-purpose processors cannot dedicate all the resources and time to these applications. The CPU also has to take care of background processes, operating system overheads, input processing, disk management, etc. Special-purpose hardware, on the other hand, can be dedicated for these computations. Hardware implementations can be optimized for solving large sparse linear systems required in these applications. FPGAs pose a problem for higher floating-point precisions, but custom Application Specific Integrated Circuits (ASICs) can be designed which can offer the full 32-bit precision for these operations.
• *Memory*: General-purpose CPUs are limited in the available on-chip memory for storing opcode and operand values. Some speed-up can be obtained by using extended instruction sets and reusing internal registers, but this will not prove enough for real-time simulation updates. Accessing off-chip memory is the dominant source of delay for these operations. Since these applications involve operations on sparse matrices, the hardware only needs to store a small amount of information. This information can be accommodated on an on-chip SRAM for very high speed/high bandwidth access.

• *Flexible Precision*: The general-purpose CPUs consist of floating-point units which are compliant with the IEEE 745 Floating Point Standard. Some applications do not need the full 32- or 64-bit precision for all computations. Considerable speed-up and area savings can be achieved by using reduced precision floating point units in special purpose hardware.

• *Time-to-Market*: Special purpose architectures for these applications would be much less complex than general-purpose CPUs. The correctness of a hardware implementation can be determined by implementing them first using HDLs. Experimental systems can also be built using off-the-shelf units like FPGAs and memory boards. This allows for an implementation with a low risk and fast time-to-market as compared to general-purpose systems.

2.6 Summary

Linear system solvers are important today in a wide variety of applications including fluid modeling, mechanical modeling and interactive graphics simulations. These applications are typically too computationally intensive for general-purpose CPUs. Many software techniques and optimizations have been employed in the past to obtain real-time interactive simulation. The main drawbacks of these methods are their inability to represent a large variety of objects and their
numerical instability due to the underlying algorithms used. These drawbacks can be overcome by using new algorithms or by executing existing algorithms using careful selection of certain parameters. Both these methods result in computationally intensive loops that are difficult to execute in real-time on general-purpose CPUs.

We have provided a case for using special-purpose hardware for accelerating these applications. The chapter outlines the characteristics of these applications which make them attractive to be implemented in hardware and describes advantages of using such hardware over general-purpose systems. The problem of physical modeling of deformable objects was presented as a representative problem of this class. Other applications include interactive fluid dynamics modeling and interactive ray tracing. To date, these applications have been largely confined to the motion picture industry. We expect that the speed-ups provided by the proposed special-purpose architecture will allow these applications to be run even on home PCs.

The next chapter (Chapter 3) takes a detailed look at some existing systems for speeding up floating-point intensive applications. We will explore a first-generation system, called Simulation of Physics on a Real Time Architecture (SPARTA), designed by Bishop et. al [35]. We will outline the algorithms and design decisions of SPARTA and take a look at some of its disadvantages. We will then provide the motivation for building a much more accurate and faster implementation using a custom ASIC.
CHAPTER 3
RELATED WORK AND MOTIVATION

3.1 Introduction

In this chapter we take a detailed look at some related work in this area, though there is little closely related research in the implementation of dedicated sparse linear system solvers. The importance and applicability of special-purpose hardware for problems involving floating point intensive iterative algorithms has recently been accepted by the semiconductor industry.

We describe a first-generation special-purpose system for deformable object modeling called SPARTA. This system was realized on an Altera high-density FPGA. We will outline the design choices, implementation, and limitations of this system and explain how the lessons learned from SPARTA were helpful in the design of the proposed architecture which we term as a second generation system.

The rest of the chapter is organized as follows: Section 3.2 discusses two related specialized systems, viz. the GRAPE project at The University of Tokyo and the PixelFlow system developed at The University of North Carolina, Chapel Hill. We also briefly mention other "coarse-grain reconfigurable systems" which use a similar design strategy but are not optimized for solving sparse linear systems. Section 3.3 gives a detailed description of SPARTA while Section 3.4, presents the design choices and overview of the second-generation system.
3.2 Related Specialized Systems

We take a detailed look at some related specialized systems to achieve speed-ups for iterative floating point algorithms.

3.2.1 GRAPE

The GRAPE (GRAvity piPE) is a special-purpose architecture to perform N-body simulations. N-body simulations are used in numerical methods for problems which try to answer questions like creation of black holes, the relation of formation of black holes to the formation of galaxies and to analyze the behavior of neighboring stars during black hole formation. To obtain a reliable result, the application needs to consider the effects of a large number of neighboring stars which involves large number of computations at every step.

Figure 3.1 The GRAPE-6 Processor Board with 8 chips

Figure 3.2 Block Diagram of a single GRAPE-6 chip
The GRAPE system consists of special-purpose pipelined hardware which takes the positions and masses of particles from the host machine and calculates the interactions between these particles. A single GRAPE-6 processor chip integrates six pipeline processors for force calculations, one pipeline processor to handle the prediction, a network interface and memory interface. One force pipeline can evaluate one particle-particle interaction per cycle. Details about the GRAPE hardware and its performance details can be obtained from [30]. Figure 5.1 illustrates the GRAPE-6 processor board and Figure 5.2 shows the block diagram of a single GRAPE-6 chip.

3.2.2 PixelFlow

The PixelFlow system is part of the Pixel-Planes project at the University of North Carolina, Chapel Hill [31]. This project was initiated almost two decades ago and its culmination is PixelFlow – a high-speed, highly realistic image generation system that was designed for realistic graphics rendering techniques like shading, texturing, anti-aliasing, and shadows [32].

Pixel-Planes [33] is a parallel image generation architecture which consists of up to 32 processors optimized for mathematical operations, up to 16 rendering units - each rendering unit containing a 128x128-pixel array of processors with on-chip memory - and a 1280x1024-pixel frame buffer, interconnected by a 5 gigabit ring network. The basic idea of the system is to increase the performance of interactive graphics rendering using parallel rendering engines. The required computational power required for such rendering is provided by the MIMD array of math-oriented processors. The processors and the rendering units together work on various sub-divisions of the screen in parallel to achieve the desired performance.

The PixelFlow system is the successor of Pixel-Planes [32]. The main problem with Pixel-Planes is that it cannot be scaled due to the presence of the ring network. PixelFlow uses a technique called "image composition" to overcome this problem. In this system, "display primitives" (for example, image sub-divisions) are distributed over an array of identical renderers
which is a SIMD array of 8192 byte-serial processing elements. Each rendering unit processes the screen-space descriptions of its fraction of primitives into pixel values and performs sophisticated shading calculations. The front-end floating-point processing and control for the rendering units are provided by HP PA-RISC PA-8000 processor running at 180 MHz. A dedicated high-speed communication network called the "Image Composition Network" then merges these images in real time to render the image on the screen.

![Diagram of PixelFlow rendering system]

Figure 3.3 Object-Parallel Rendering by using Image Composition from [32]

PixelFlow thus follows an object-parallel approach to achieve parallel computation on screen sub-divisions by individual rendering units. Figure 3.3 illustrates how this approach can be used for image generation on a screen. Each renderer processes its own screen sub-division and calculates a full-screen image of its fraction of primitives which are then composed to render the entire image. The main advantages of the system are its flexibility and scalability. It can be implemented in many forms, from desk-side systems for home applications to multiple-rack systems for very high performance graphics rendering.
3.2.3 MATRIX

MATRIX [34] consists of an array of identical, 8-bit functional units overlaid with a configurable network. Each functional unit consists of a 256x8-bit memory, an 8-bit ALU and some control logic including a 20x8 NOR plane. The connection network is hierarchical and supports three levels of interconnect - nearest neighbor connection, bypass connection, and global connection. Functional unit ports, which are the basis of the flexibility of the architecture, can be statically configured or dynamically switched. The basic philosophy behind the design of MATRIX is to allow the application to dictate the allocation of resources and their division between computation and control. This differs from traditional reconfigurable systems where devices are configured for a particular application based on its instruction stream. MATRIX allows the application itself to determine the specifics of the instruction stream and hence controls the division of resources. Due to this, MATRIX can be used as a systolic array, a custom VLIW processor or a Multiple-SIMD/VLIW hybrid processor depending upon the application.

Each of the above systems makes use of multiple processing elements or reconfigurable units to provide the required computational power for particular application domains. These systems underline the importance of special-purpose architectures for specific applications which require large amounts of computational power. All these systems heavily exploit the inherent parallelism in these applications to perform operations in parallel for high performance.

Our proposed architecture differs from these systems in the sense that it is designed as a dedicated sparse linear system solver for particular application domains. It is optimized to exploit parallelism in highly iterative floating point intensive algorithms while also providing reconfigurability so that a variety of algorithms can be mapped to it. We think that the design philosophy and the resulting implementation is the first of its kind for applications like deformable object modeling and fluid modeling.
3.3 A First Generation System for Deformable Object Modeling

The first generation system was developed by Bishop et. al. [14, 35] at the Pennsylvania State University. SPARTA was a proof-of-concept system implemented as a high density FPGA on a custom printed circuit board. It was then connected to a host machine with a parallel interface. The aim of the project was to design a hardware-software experimental system for real time physical modeling. A CPU-based physical modeling application was first designed to
demonstrate the algorithms used. The instruction set architecture for the system was synthesized from this software implementation and a pipeline was designed to support spring force and collision detection/resolution computations.

The aim of the SPARTA project was to implement the spring force computation and collision/detection algorithms in hardware to prove that specialized hardware implementation can provide considerable speed-ups beyond what is possible in software implementations. The final implementation was intended to be a single chip ASIC with specialized pipelines for spring force computation and collision detection/resolution. In addition, the design contained on-chip SRAM for fast, high-bandwidth memory access and relevant logic for the pipelines and host CPU interface.

The design decisions in SPARTA were heavily influenced by the implementation choices. Altera EPF10K250 was selected as the FPGA and a custom printed circuit board with a parallel interface to a host machine was designed [13]. FPGAs have poor efficiency when implementing floating point arithmetic [36, 37]. The main problems when using FPGAs for implementing floating point operations are the FPGA utilization and interconnections. Typically, most of the area and interconnect in full 32-bit precision floating point arithmetic is taken up by barrel shifters in the normalization unit. This leads to low performance in the normalization steps in the floating point operations. Some alternatives followed by researchers are to use deeply pipelined arithmetic units [37], using reduced precision floating point numbers or using fixed-point or custom formats which provide high performance for specific applications [36]. All these approaches sacrifice precision or performance and are hence unsuitable for applications in graphics processing where the accumulated error may result in very different visuals than expected.

In the final SPARTA implementation, FPGA utilization was 79% of the 250K gates. A number of simplifications were made to the pipeline for implementation on the FPGA. Specifically, only spring force computation using the Euler's method was implemented. The
collision detection/resolution pipeline was found to be too complex to be easily implemented in hardware. Reduced precision floating point arithmetic was used to minimize the area required. Figure 3.4 illustrates the custom printed circuit board of the SPARTA chip. Figure 3.5 outlines the spring force computation pipeline which was implemented in hardware.

3.4 The Second Generation System

The second generation system is in many ways a follow-up to the SPARTA project. Most of the design choices of this system have been dictated by the lessons learnt during the SPARTA implementation [35, 38]. Specifically, due to the instability of Euler integration for large stepsize, it is seldom used in commercial projects. Hence, the second generation system was designed to support implicit integration. An advantage of implicit methods is that they operate on local data which can be stored on the on-chip SRAM. This leads to less complicated hardware. Secondly, due to the numerous problems with FPGAs when implementing floating point arithmetic, a coarse-grained system consisting of several processing elements (PEs), each with an optimized 32-bit IEEE compliant floating point arithmetic unit, was considered more suitable.

The aim of the second generation system is to provide high performance at low cost for highly iterative floating point intensive computations. The system consists of a custom VLSI chip mounted on a custom PCI card and will act as a dedicated sparse linear system solver. Since the computations do not change much over successive iterations, the data required can be stored on on-chip SRAM. The resulting design is a coarse-grained 2D architecture of independent processing elements (PEs). Each PE consists of a fast floating point unit, control memory, operand memory, relevant control logic and a programmable communication interface to neighboring PEs. The system will communicate with the host machine via a parallel interface. Nearest neighbor interconnections strategy was selected due to its low complexity. Control instructions and data would be downloaded into the private memories of individual PEs from the host machine. Each PE would then process instructions from its own control memory and the
result would be communicated back to the host CPU (for example, data required for screen updates).

![Diagram of a Second Generation System for Floating Point Intensive Algorithms](image)

**Figure 3.6 A Second Generation System for Floating Point Intensive Algorithms**

Figure 3.6 provides a high level view of the system. The entire system was first represented in VHDL to verify its correctness. A suite of CAD tools from Cadence were used to synthesize the design and to perform the placement and routing. Standard cell libraries and memory compilers for generating high-speed memory units were obtained from Artisan Components Inc. and a standard cell-based design flow was followed.

The final system will consist of several custom ICs to populate a custom printed circuit board. The system will communicate with a host machine using a parallel interface implemented
as a FPGA PCI core. Currently, the design of a single chip which contains a 2x2 array of processing elements has been completed.

3.5 Summary

There are many application areas which involve large iterative floating point intensive computations. There have been several specialized systems to address this problem. Typically, these systems provide high floating-point performance by parallelizing the operations on an array of processing elements. We took a look at two such specialized systems viz., GRAPE-6 and PixelFlow.

There is little related research in the implementation of dedicated sparse linear system solvers for applications like physically-based modeling. SPARTA was a first generation system to support spring computations for deformable object modeling and was implemented as a high-density FPGA, but the performance was limited due to routing and area constraints on the FPGA. The current system is the second generation system and aims to provide high floating point performance to support better, more stable algorithms for physically-based modeling. The speed-ups achieved by our design would enable such applications to be run even on home PC's by using our system as an add-on PCI card.
4.1 Overview

Our system will consist of a custom Printed Circuit Board (PCB) containing a custom integrated circuit (IC) with a Parallel Communication Interface (PCI) implemented as an FPGA. The custom IC consists of multiple, independent processing elements designed to provide very high performance for floating point arithmetic. The PCI interface provides the necessary fast medium for communication with a host machine for real-time display updates. A production system with sufficient die area will consist of a number of these custom ICs populating the PCB to provide faster processing. The current prototype supports only one IC for the PCB due to area, cost, and time constraints. Figure 4.1 illustrates the organization of the PCI board which supports multiple ICs.

Figure 4.1 Organization of a production PCB with a PCI Host Interface
4.1.1 Design Choices

The primary goal of this design is to achieve the highest performance at a low cost for highly iterative floating point intensive applications. The codes for the target applications typically perform a large number of relatively simple iterations. High performance can be achieved by distributing these computations to a number of independent processing elements. Each processing element (PE) is designed to support local data, control, and single-precision floating-point arithmetic. At the board level, each IC is designed to support communication with the PCI FPGA core and other ICs (if any) through a number of 32-bit buses. This will be a limiting factor on the number of ICs that can be included on a single board due to concerns like packaging difficulties, increase in the heat dissipation and an increase in chip-to-chip communication delay because of routing congestion.

The most important constraint in the design of the present IC was the available die area. The IC is being fabricated by Taiwan Semiconductor Manufacturing Corporation (TSMC) [39] through MOSIS. The available die area for the IC was 7.5 sq. mm. The initial design included a 4x4 array of processors, but given the pin-count and dimensions of the I/O standard cell library, this could not be fit into the given area. Hence, the design had to be restricted to a 2x2 array of processing elements to be accommodated in the available area. Some additional multiplexing hardware was also required to reduce the pin-count.

4.1.2 Structure of the Processing Element

Each PE consists of a single-precision floating point functional unit, a simple control unit and local SRAM for operand and control memory [40]. Memory cell design and area is of critical importance because memory is usually a dominant source of die area and power consumption. Although DRAM offers an advantage in this regard, SRAM memories were chosen for this design due to their robustness and high bandwidth which offers a very high memory performance.
Each PE consists of four banks of 128 32-bit words for the operand memory and one bank of 128 40-bit words as the control memory. The control logic for each PE is relatively simple.

## 4.1.3 On-chip Communication Strategy

The IC consists of multiple PEs connected in a NxN array. The array size can be scaled up depending upon the available die area. The current implementation contains a 2x2 array of PEs. Each PE is connected to its cardinal neighbors (east, west, north, south) using a Nearest Neighbor interconnect strategy. Each PE in the array identifies itself with its hard-coded X (row) and Y (column) co-ordinates.

![Figure 4.2 High-Level Organization of a single IC](image)

The primary concern in selecting an interconnect strategy was to allow for efficient inter-PE communication without causing a significant area overhead. A nearest neighbor interconnection results in a simple and regular layout with an added advantage during the
placement and routing stage being that connections can be made by abutment rather than complicated wiring as in other strategies like hierarchical interconnect. Since the PE organization is relatively coarse as compared to an FPGA, using a hierarchical tree-based approach for interconnect will cause an excessive and unnecessary wiring overhead. Figure 4.2 illustrates the high-level view of the organization of a single IC.

### 4.1.4 Modes of Operation

The system has two modes of operation: the *Override or Pipe Mode*, when the instructions and data are downloaded to the IC from the host machine and the *Compute Mode* when the processing elements execute the instructions in their control memory. A global signal, denoted as *PC-Enable* (Pipe-Compute) is used to switch between these modes.

In the Override Mode, each PE receives two 56-bit control words from its North and West neighbors. The PE forms a single 56-bit word, called the *Override Word* or *OvrWrd*, by bitwise OR-ing the words received from its North and West neighbors. Each *OvrWrd* consists of a 16-bit control field and a 40-bit data field. The control field includes information like the *PE coordinates* for which the *OvrWrd* is meant, the *opcode of the operation* to be performed and the *operand locations* (operand memory, control memory or program counter) in the target PE. The data field contains the actual instruction to be written to or executed by the PE. Every PE compares its own hard-coded co-ordinates with the co-ordinates from the control field of the received *OvrWrd*. If there is a match, then the PE executes the instruction, else it simply passes the *OvrWrd* to its South and East neighbors. Thus the I/O commands in Override mode fan-out in a 2-D wave starting from the north-west corner of the array towards the south-east corner. This novel scheme dramatically reduces the number of pins required for the entire array.

In the Compute Mode, each PE executes the instruction pointed to by the *Program Counter (PC)*. These include arithmetic, control and communication instructions which are detailed in Section 4.2.1. The results of the computation are passed on to the South and East
neighbors by each PE and are read off the array at the south-east corner. Figure 4.3 illustrates the fan-out of override mode words.

![Diagram of communication between cells with labels: 56-bit Control word, PC-Enable=0, Control word fans-out from NW corner]

**Figure 4.3 Streaming of Override Mode Instructions**

### 4.2 Instruction Set Architecture

There are two different instruction sets for the two modes of operation. These are termed (not surprisingly!) as *Override Mode Instructions* and *Compute Mode Instructions*. The override mode instructions are used primarily to write instructions to the control memory and to initialize the program counter. The compute mode instructions are used to read the control memory and perform certain computational tasks. The compute mode instructions are divided into three classes, *viz.* arithmetic, control, and interconnect.

#### 4.2.1 Compute Mode Instruction Set

Figure 4.4 illustrates the format of the compute mode instructions. The instructions are 40-bit words read from the control memory. Each instruction consists of the opcode (OP) which specifies the operation and four operand fields. Each operand field independently specifies an operand SRAM bank (2 bits) and a location within the bank (7 bits). The instructions are restricted to access an operand bank only once per instruction.
### Table 4.1 Compute Mode Instructions

<table>
<thead>
<tr>
<th>CLASS</th>
<th>OPCODE</th>
<th>MNEMONIC</th>
<th>OPERATION</th>
</tr>
</thead>
<tbody>
<tr>
<td>Arithmetic</td>
<td>0000</td>
<td>Nop</td>
<td>no operation</td>
</tr>
<tr>
<td></td>
<td>0001</td>
<td>add r1, r2, r4</td>
<td>r1 + r2 → r4</td>
</tr>
<tr>
<td></td>
<td>0010</td>
<td>sub r1, r2, r4</td>
<td>r1 - r2 → r4</td>
</tr>
<tr>
<td></td>
<td>0011</td>
<td>mult r1, r2, r4</td>
<td>r1 * r2 → r4</td>
</tr>
<tr>
<td></td>
<td>0100</td>
<td>mac r1, r2, r3, r4</td>
<td>r1 * r2 + r3 → r4</td>
</tr>
<tr>
<td></td>
<td>0101</td>
<td>div r1, r2, r4</td>
<td>r1 / r2 → r4</td>
</tr>
<tr>
<td>Control</td>
<td>0110</td>
<td>blz r1, PC</td>
<td>if (r1 &lt; 0.0) then r4[6..0] → PC. If the value stored at r1 is less than zero, then load the PC with seven rightmost bits of r4</td>
</tr>
<tr>
<td>Interconnect</td>
<td>0111</td>
<td>load dir, r4</td>
<td>NEIGHBOR(r1[1..0]),COMM_REG → r4. where NEIGHBOR is east, west, south or north depending upon the DIR field given by the two rightmost bits of r1</td>
</tr>
<tr>
<td></td>
<td>1000</td>
<td>store r1</td>
<td>r1 → COMM_REG. Stores the value pointed by r1 to its own COMM_REG</td>
</tr>
</tbody>
</table>

The arithmetic instruction class includes *NOP*, a zero operand instruction, *ADD, SUB, MULT, DIV*, which are three operand instructions and *MAC* which is a four operand instruction.

The control class consists of the conditional branch instruction *BLZ* which loads the new program counter value (PC) if the value stored at the location pointed by *R1* is less than zero. The interconnect class consists of the *LOAD* and *STORE* instructions. Inter-PE communication is accomplished by using the 32-bit Communication Register (*COMM_REG*) within each PE. A *LOAD* instruction reads data off the *COMM_REG* of the specified PE while a *STORE* instruction
causes a PE to write to its own COMM_REG. The instructions and the operations they perform are detailed in Table 4.1. The design allows every instruction to access an operand SRAM bank only once. This removes the need for any additional decode logic for operand memory access and allows all instructions to be completed in a single clock cycle. The goal is to keep the design simple and compact enough to be accommodated into the given area while correctly performing the intended operations.

4.2.2 Override Mode Instruction Set

Figure 4.5 illustrates the format of the override mode instruction set. Each instruction is a 56-bit word which consists of a 2-bit opcode, a 14-bit location field and a 40-bit value field. The location field specifies the destination for the 40-bit value field in terms of the (X, Y) coordinates of the target PE, the operand or control memory bank, and the location within the bank. The override instructions are streamed from the north-west corner of the array, (PE 0,0), toward the south-east corner, (PE (n-1), (n-1)). Each PE forms the next instruction by bit-wise OR-ing the override messages received from its north and west neighbors. Simultaneously, it transmits its current override word to its south-east neighbors. Hence override mode instructions fan-out in a 2-D wave from the upper left corner of the array toward the bottom right corner.

The override mode instruction set consists of the LOOKUP and PUT instructions. On receiving a LOOKUP command, a PE examines the (XCoord, YCoord) fields to determine if the instruction is meant for it. If the command is not for the PE, the PE propagates it unchanged to its south and east neighbors. However, if the command is meant for the PE, then the PE performs a SRAM read from the specified location and prepares a FOUNDIT response in the following cycle. This response propagates to the south-east end of the array from where it can be read by the host machine. Similarly, a PUT instruction is used to write data to the specified memory bank. No response is generated by a PUT instruction.
Table 4.2 provides the details of the override mode instructions. A \textit{PUT} instruction targeting the operand memory right-aligns the 32-bit data in the 40-bit value field. PEs responding to a \textit{LOOKUP} instruction also right-align the 32-bit data in the \textit{FOUNDIT} response.

<table>
<thead>
<tr>
<th>Field</th>
<th>OPCODE</th>
<th>LOCATION</th>
<th>VALUE</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>2</td>
<td>14</td>
<td>40</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Field</th>
<th>XCoord</th>
<th>YCoord</th>
<th>Bank</th>
<th>Offset</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>2</td>
<td>2</td>
<td>3</td>
<td>7</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Bank</th>
<th>Usage</th>
</tr>
</thead>
<tbody>
<tr>
<td>000 – 011</td>
<td>Operand Memories</td>
</tr>
<tr>
<td>100</td>
<td>Control Memory</td>
</tr>
<tr>
<td>101, 110</td>
<td>Unused</td>
</tr>
<tr>
<td>111</td>
<td>Program Counter (PC)</td>
</tr>
</tbody>
</table>

Figure 4.5 Override Mode Instruction Format

<table>
<thead>
<tr>
<th>Opcode</th>
<th>Mnemonic</th>
<th>Operation</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>Nop</td>
<td>No operation</td>
</tr>
<tr>
<td>01</td>
<td>lookup XCoord, YCoord, LOC</td>
<td>Lookup the value at the location LOC in the PE at (XCoord, YCoord). Prepare a FOUNDIT response by setting the opcode to 11 to indicate a successful lookup</td>
</tr>
<tr>
<td>10</td>
<td>put XCoord, YCoord, LOC, val</td>
<td>Store the value “val” at the location LOC in the PE at (XCoord, YCoord)</td>
</tr>
<tr>
<td>11</td>
<td>foundIt XCoord, YCoord, LOC</td>
<td>Signifies a successful lookup (refer description of lookup instruction)</td>
</tr>
</tbody>
</table>

Table 4.2 Override Mode Instructions
4.3 Components of a Single Processing Element

This section provides the design details of the various components of a single processing element. Figure 4.6 shows the complete datapath of a single PE. The main components of a PE are the floating point functional unit, memory unit and the control unit. The entire design was represented using VHDL. The details about the analysis, compilation and synthesis of the VHDL representation are given in Chapter 5.

4.3.1 The Floating Point Arithmetic Unit (FPU)

The floating point unit supports various operations on single-precision floating point numbers and is strictly compliant with the IEEE Standard for Floating Point Arithmetic. Each PE includes a floating point Add (FPADDSUB), Subtract, Multiply (FPMULT) and Divide (FPDIV) units. Multiply-and-accumulate (MAC), which is a very important operation for processing of matrices, is supported by using the FPADDSUB and FPMULT units. The entire floating point unit has been designed as a complete combinational block by using unrolling of the algorithms in hardware. This ensured that one floating point operation (FLOP) was performed per clock cycle by a single PE.

Considerable time and effort were spent on verifying the correctness of the FPU as this is the most crucial unit of the PE. Specifically, each individual unit was tested with approximately 5 million randomly generated test vectors. The results computed by the hardware were then compared with those produced by GCC on an x86 machine running Red Hat Linux. Additionally, separate test benches were used to verify correctness at boundary values.

The details about floating point representation, algorithms, design and schematics of each individual unit are provided in Appendix A. Only a high-level view of the implemented hardware is provided. For a thorough treatment of floating point arithmetic, the reader can refer to [41, 42, 43].
4.3.2 The Memory Unit

Each PE consists of four SRAM banks of 128 32-bit words each, for the operand memory and one SRAM bank of 128 40-bit words for the control memory. Typically, the full-custom design of memory units is a long process and it is difficult to do hand-layout of robust, compact and fast memory units in a short time-frame. In the standard cell-based design flow, memory units are generally designed using memory compilers. Memory compilers are software programs which, given the specifications, can generate VHDL/Verilog representations, timing library files and even complete layouts in GDSII or CIF format of memory units. The compilers are supplied by specific vendors and usually support only a particular fabrication technology. We were able to obtain the memory compiler for a High-Speed, Single-Port, Synchronous SRAM (SRAM-SP-HS) from Artisan Components Inc. [44] under the MOSIS Educational Program. The compiler is geared to achieve the full-speed, high density and low power operation for fabrication technologies ranging from 0.10 to 0.25 microns. It is capable of generating an ASCII data-table, VHDL model, Verilog model, Synopsys Model, PrimeTime model, a Timing Library Format (TLF) model and a Library Exchange Format (LEF) footprint. Since this version was provided for educational use only, we could not generate a GDSII or CIF layout using the compiler.

The compiler needs to be supplied with some generic memory parameters like the instance name to be used in the VHDL/Verilog model, number of words, number of bits per word and the multiplexer width for address decoding. Other parameters that can be tweaked include the top metal layer which is process dependent and the horizontal and vertical metal ring layers for the power and ground connections. The ASCII data-table provides data about the cell area, access times, voltage (V) and current (milliamps) values, access times and the capacitance values on the address and data lines. An example of the data-table for the 128x32 operand memory and 128x40 control memory is given in Table 4.3.
Table 4.3 Data Table for 128x32 Operand Memory and 128x40 Control memory generated by the Artisan memory compiler

<table>
<thead>
<tr>
<th></th>
<th>Operand Memory Bank</th>
<th>Control Memory Bank</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>128x32 SRAM</td>
<td>128x40 SRAM</td>
</tr>
<tr>
<td>geomx</td>
<td>563.820</td>
<td>685.180</td>
</tr>
<tr>
<td>geomy</td>
<td>154.835</td>
<td>154.835</td>
</tr>
<tr>
<td>Ring_size</td>
<td>5.200</td>
<td>5.200</td>
</tr>
<tr>
<td>icc</td>
<td>0.114</td>
<td>0.139</td>
</tr>
<tr>
<td>ice_r</td>
<td>0.104</td>
<td>0.126</td>
</tr>
<tr>
<td>ice_w</td>
<td>0.124</td>
<td>0.152</td>
</tr>
<tr>
<td>icc_peak</td>
<td>384.211</td>
<td>478.602</td>
</tr>
<tr>
<td>icc_desel</td>
<td>0.000</td>
<td>0.000</td>
</tr>
<tr>
<td>icc_standby</td>
<td>0.000</td>
<td>0.000</td>
</tr>
<tr>
<td>tcyce</td>
<td>0.788</td>
<td>0.809</td>
</tr>
<tr>
<td>Ta</td>
<td>0.736</td>
<td>0.753</td>
</tr>
<tr>
<td>tas</td>
<td>0.239</td>
<td>0.244</td>
</tr>
<tr>
<td>tah</td>
<td>0.038</td>
<td>0.038</td>
</tr>
<tr>
<td>tcs</td>
<td>0.272</td>
<td>0.272</td>
</tr>
<tr>
<td>tch</td>
<td>0.000</td>
<td>0.000</td>
</tr>
<tr>
<td>tws</td>
<td>0.272</td>
<td>0.275</td>
</tr>
<tr>
<td>twh</td>
<td>0.000</td>
<td>0.000</td>
</tr>
<tr>
<td>tds</td>
<td>0.133</td>
<td>0.133</td>
</tr>
<tr>
<td>tdh</td>
<td>0.000</td>
<td>0.000</td>
</tr>
<tr>
<td>thz</td>
<td>0.428</td>
<td>0.434</td>
</tr>
<tr>
<td>tlz</td>
<td>0.377</td>
<td>0.383</td>
</tr>
<tr>
<td>tckh</td>
<td>0.083</td>
<td>0.083</td>
</tr>
<tr>
<td>tckl</td>
<td>0.112</td>
<td>0.112</td>
</tr>
<tr>
<td>tckr</td>
<td>4.000</td>
<td>4.000</td>
</tr>
<tr>
<td>load_q</td>
<td>0.271</td>
<td>0.271</td>
</tr>
<tr>
<td>icap_a</td>
<td>0.053</td>
<td>0.053</td>
</tr>
<tr>
<td>icap_d</td>
<td>0.004</td>
<td>0.004</td>
</tr>
<tr>
<td>icap_clk</td>
<td>0.265</td>
<td>0.265</td>
</tr>
<tr>
<td>icap_cen</td>
<td>0.014</td>
<td>0.014</td>
</tr>
<tr>
<td>icap_wen</td>
<td>0.015</td>
<td>0.015</td>
</tr>
<tr>
<td>icap_oen</td>
<td>0.010</td>
<td>0.010</td>
</tr>
<tr>
<td>ocap_q</td>
<td>0.019</td>
<td>0.019</td>
</tr>
<tr>
<td>pwn_ck</td>
<td>10.000</td>
<td>10.000</td>
</tr>
<tr>
<td>vn_ck</td>
<td>0.822</td>
<td>0.822</td>
</tr>
<tr>
<td>vn_pwr</td>
<td>0.198</td>
<td>0.198</td>
</tr>
<tr>
<td>vn_gnd</td>
<td>0.198</td>
<td>0.198</td>
</tr>
</tbody>
</table>
4.3.3 The Control Unit

This section provides a detailed description of the various control signals shown in the datapath of Figure 4.6. The control unit is responsible for providing control signals to the program counter, operand and control memories, the functional units and the communication unit.

As designed, the system has only two global signals: Clock (Clk) and PC-Enable. Each PE identifies itself by its $X$ (row) and $Y$ (column) co-ordinates which are hard-coded. The inputs of each PE include the two global signals, the X and Y co-ordinates required for identification, the 32-bit Communication words from its South ($SCommIn$) and East ($ECommIn$) neighbors, and the 56-bit Override mode words from its North ($NOvrWrdIn$) and West ($WOvrWrdIn$) neighbors. The outputs of each PE are the 56-bit override mode word ($OvrWrd$) to its South and East neighbors, and the 32-bit communication word ($CommWrd$) to its North and West neighbors. The input and output pins are multiplexed to reduce the I/O pin-count.

<table>
<thead>
<tr>
<th>OvrWrdData[39..0]</th>
<th>40-bit Data field</th>
</tr>
</thead>
<tbody>
<tr>
<td>OvrWrd.Op[55..54]</td>
<td>2-bit Override Mode Opcode</td>
</tr>
<tr>
<td>OvrWrd.PE[53..50]</td>
<td>PE Co-ordinates</td>
</tr>
<tr>
<td>OvrWrd.Bank[49..47]</td>
<td>Bank (000 – 011, 100, 111)</td>
</tr>
<tr>
<td>OvrWrd.Addr[46..40]</td>
<td>Offset</td>
</tr>
</tbody>
</table>

Table 4.4 Fields of the Override Mode Word

<table>
<thead>
<tr>
<th>CM.Op[39..36]</th>
<th>4-bit Opcode</th>
</tr>
</thead>
<tbody>
<tr>
<td>CM.R1Bank[35..34], CM.R1Addr[33.27]</td>
<td>R1 – Bank, R1 – Address</td>
</tr>
<tr>
<td>CM.R2Bank[26..25], CM.R2Addr[24..18]</td>
<td>R2 – Bank, R2 – Address</td>
</tr>
<tr>
<td>CM.R3Bank[17..16], CM.R3Addr[15.9]</td>
<td>R3 – Bank, R3 – Address</td>
</tr>
<tr>
<td>CM.R4Bank[8..7], CM.R4Addr[6..0]</td>
<td>R4 – Bank, R4 – Address</td>
</tr>
</tbody>
</table>

Table 4.5 Fields of the Control Memory Word
Figure 4.6 – Part I. Datapath of a single PE
Figure 4.6 – Part II. Datapath of a single PE
The values of the control signals depend upon the mode of operation of the system. In the override mode, the values depend upon the OvrWrd received from the North and West neighbors while in the compute mode, the control signal values are determined by the control word (CMWrd) read from the local control memory. The various fields of the OvrWrd and the CMWrd are detailed in Tables 4.4 and 4.5. Note that the fields are represented by $FIELD\_NAME[NN..NN]$, where $FIELD\_NAME$ refers to the name of the field and $NN..NN$ represents the bit-boundaries of the field in the OvrWrd or CMWrd.

The control signal values are determined using these fields. The signal equations are represented in pseudo-code. The various control signals can be classified as External I/O signals, Status signals, Operand Memory Signals, Control Memory Signals, Functional Unit Signals and Program Counter signals.

### 4.3.3.1 External I/O Signals

The external I/O signals provide the interface for connection to other PEs and to the host interconnect for the boundary PEs. The various external I/O signals are:

<table>
<thead>
<tr>
<th>SIGNAL NAME</th>
<th>DESCRIPTION</th>
</tr>
</thead>
<tbody>
<tr>
<td>Clk</td>
<td>Global Clock Signal</td>
</tr>
<tr>
<td>PCEnable</td>
<td>Provides switching between override and compute modes</td>
</tr>
<tr>
<td>XCoord[1..0]</td>
<td>X (row) co-ordinate for PE identification</td>
</tr>
<tr>
<td>YCoord [1..0]</td>
<td>Y (column) co-ordinate for PE identification</td>
</tr>
<tr>
<td>WOvrWrdIn[55..0]</td>
<td>56-bit OvrWrd from West neighbor</td>
</tr>
<tr>
<td>NOvrWrdIn[55..0]</td>
<td>56-bit OvrWrd from North neighbor</td>
</tr>
<tr>
<td>SCommIn[32..0]</td>
<td>32-bit CommWrd from South neighbor</td>
</tr>
<tr>
<td>ECommIn[32..0]</td>
<td>32-bit CommWrd from East neighbor</td>
</tr>
</tbody>
</table>

Table 4.6 External I/O Signals
4.3.3.2 Status Signals

The status signals are asserted or de-asserted to indicate the status of certain operations. These signals include PE-Hit, Unique and WriteBack.

PE-Hit is asserted to indicate an exact match between the $X$ and $Y$ co-ordinate values received in the OvrWrd with the own co-ordinates of the PE. The PE-Hit is used to disable all the other operations if the received OvrWrd is not meant for the current PE ($PE\text{-Hit} = 0$).

The Unique signal provides a sanity check of operand memory bank access by any CMWrd during the compute mode. Specifically, this signal disables multiple operand memory bank accesses by the same instruction. This policy has been enforced to maintain simplicity so that all instructions can be completed in one clock cycle.

The WriteBack signal is de-asserted during a read operation when the memory bank drives its output bus. It is de-asserted for a memory write when the bus is driven by an external source.

<table>
<thead>
<tr>
<th>SIGNAL</th>
<th>LOGIC</th>
<th>DESCRIPTION</th>
</tr>
</thead>
<tbody>
<tr>
<td>PE-Hit</td>
<td>!(PC-Enable) &amp;&amp; OvrWrd PE == (( XCoord &lt;&lt; 2)</td>
<td></td>
</tr>
<tr>
<td>WriteBack</td>
<td>(CM.Op &gt; 0 &amp;&amp; CM.Op &lt; 6)</td>
<td></td>
</tr>
</tbody>
</table>

Table 4.7 Status signals
4.3.3.3. Operand and Control Memory Signals

From the datapath design of Figure 4.6, the operand and control memory interface consists of the Address Select (AddrSel), Output Enable (OE_n), Write Enable (Wrt_n) and the Chip Enable (CE_n) control signals.

The CE_n signal is an active low signal used to enable/disable a particular memory bank. It is asserted (CE_n = 0) at all times for both operand and control memories during the compute mode. In the override mode, however, the CE_n signal is asserted for the particular banks only if there is a write instruction to the bank. When CE_n is de-asserted, the memory bank ignores any transition on its input lines and maintains its output bus at tri-state. The OE_n signal is used to control the direction of the internal output line drivers in the memory units. The OE_n signal is asserted (OE_n = 0) for all read operations when the memory drives the output bus and it is de-asserted during a write operation when the output bus is driven by an external source. The Write Enable (Wrt_n) signal is used to distinguish between a read and a write operation to the memory bank. On assertion (Wrt_n = 0), a write operation is performed and a read operation is performed when it is de-asserted.

Table 4.8 provides the calculation of these signals in the override and compute modes. In Table 5, the entity ‘x’ refers to the particular bank being addressed in the calculation. ‘x’ can take the values 0, 1, 2 and 3. Also, OM refers to the operand memory banks and CM refers to the control memory bank. For e.g. OMxCE_n refers to the calculation of the CE_n signal for the operand memory specified by ‘x’.
<table>
<thead>
<tr>
<th>SIGNAL</th>
<th>LOGIC</th>
<th>DESCRIPTION</th>
</tr>
</thead>
<tbody>
<tr>
<td>OMxCE_n</td>
<td>if (PCEnable = 1) then 0 elsif (!PCEnable &amp;&amp; OvrWrd.Bank == x) then 0 else 1</td>
<td>CE_n signal is asserted at all times in the compute mode. It is asserted in the override mode if there is a write command to the bank specified by x.</td>
</tr>
<tr>
<td>CMCE_n</td>
<td>if (PCEnable = 1) then 0 elsif (!PCEnable &amp;&amp; PE-Hit &amp;&amp; OvrWrd.Op == 2 &amp;&amp; OvrWrd.Bank == 4) then 0 else 1</td>
<td>CE_n signal for the control memory bank is asserted at all times in the compute mode. It is asserted if there is a write command to the control memory in the override mode.</td>
</tr>
<tr>
<td>OMxOE_n</td>
<td>if (PCEnable &amp;&amp; CM.R4Bank == x) then 1 elsif (!PCEnable &amp;&amp; OvrWrd.Op == 2) then 1 else 0</td>
<td>OE_n signal is de-asserted for an operand memory write. It is asserted at all other times.</td>
</tr>
<tr>
<td>CMOE_n</td>
<td>!PCEnable &amp;&amp; OvrWrd.Op == 2</td>
<td>OE_n is de-asserted on a write to the control memory during override mode. It is asserted at all times during the compute mode (when there are no writes to the control memory)</td>
</tr>
<tr>
<td>OMxWrt_n</td>
<td>if (Unique&amp;&amp; WriteBack &amp;&amp; PCEnable &amp;&amp; CM.R4Bank == x) then 0 elsif (!PCEnable &amp;&amp; PE-Hit &amp;&amp; OvrWrd.Op == 2 &amp;&amp; OvrWrd.Bank == x) then 0 else 1</td>
<td>Wrt_n is asserted for an operand memory write during compute and override mode. It is de-asserted at all other times.</td>
</tr>
<tr>
<td>CMWrt_n</td>
<td>!(!PCEnable &amp;&amp; PE-Hit &amp;&amp; OvrWrd.Op == 2 &amp;&amp; OvrWrd.Bank == 4)</td>
<td>Wrt_n for control memory is asserted only in the override mode</td>
</tr>
<tr>
<td>OMxAddrSel</td>
<td>if (!PCEnable) then 4 elsif (CM.R1Bank == x) then 0 elsif (CM.R2Bank == x) then 1 elsif (CM.R3Bank == x) then 2 else 3</td>
<td>AddrSel is the control signal for the external mux to select the proper read/write address for any operand memory bank. It is calculated by the value of ‘x’.</td>
</tr>
<tr>
<td>CMAaddrSel</td>
<td>!PCEnable</td>
<td>In override mode, the control memory address is obtained from the OvrWrd while in the compute mode, it is obtained from the Program Counter (PC).</td>
</tr>
</tbody>
</table>

Table 4.8 Operand and Control Memory signals
4.3.3.4 Functional Unit Signals

The functional unit signals are used to control the operation to be performed and to determine the final result depending upon the operation. The details of these signals are provided in Table 4.9.

<table>
<thead>
<tr>
<th>SIGNAL</th>
<th>LOGIC</th>
<th>DESCRIPTION</th>
</tr>
</thead>
<tbody>
<tr>
<td>MACOp</td>
<td>CM.Op == 4</td>
<td>Control signal to select between the MUL and MAC operations</td>
</tr>
<tr>
<td>Sub/Add_n</td>
<td>CM.Op == 2</td>
<td>Control signal to select between the ADD and SUB operations. This bit is input to the FPAADD/SUB unit as initial CARRYIN bit.</td>
</tr>
<tr>
<td>SignBit</td>
<td>R1[31:31]</td>
<td>Sign bit of the data stored at R1. Used in the blz instruction to make a branching decision</td>
</tr>
<tr>
<td>R1SrcSel</td>
<td>!PCEnable</td>
<td>Used to select the correct address to access the R1 bank in override and control modes</td>
</tr>
<tr>
<td>ResultSel[2..0]</td>
<td>if (!PCEnable) then 011</td>
<td>Control signal for selecting the correct output depending upon the operation denoted by CM.Op. All functional units much on the input data, hence result has to be selected from the correct functional unit</td>
</tr>
<tr>
<td></td>
<td>elsif (CM.Op == 3) then 000</td>
<td></td>
</tr>
<tr>
<td></td>
<td>elsif (CM.Op == 1)</td>
<td></td>
</tr>
<tr>
<td></td>
<td>elsif (CM.Op == 5) then 010 else 100</td>
<td></td>
</tr>
</tbody>
</table>

Table 4.9: Control Signals for the Functional Unit

4.3.3.5 Miscellaneous Control Signals

Other miscellaneous control signals are used to control the program counter (PC) and the communication and output units. Specifically, the PC uses three signals: PCInSel, to select the correct input address to the PC, PCLd which loads the PC with a new address when it is asserted and PCCnt which causes an automatic increment of the address in the PC when it is asserted. Care is taken to ensure that PCCnt and PCLd are not asserted at the same time. These signals, their logic and operation are detailed in Table 4.10.
<table>
<thead>
<tr>
<th>SIGNAL</th>
<th>LOGIC</th>
<th>DESCRIPTION</th>
</tr>
</thead>
<tbody>
<tr>
<td>PCInSel</td>
<td>!(PCEnable)</td>
<td>Controls input address to the PC. In override mode, PC address is obtained from the OvrWrd. The PC can be overwritten by a successful jump in the compute mode</td>
</tr>
<tr>
<td>PCLd</td>
<td>(PCEnable &amp;&amp; CM.Op == 6 &amp;&amp; SignBit)</td>
<td></td>
</tr>
<tr>
<td>PCCnt</td>
<td>PCEnable &amp;&amp; (!PCLd)</td>
<td>This is the increment signal of the PC which is asserted every cycle in the compute mode if PCLd is de-asserted</td>
</tr>
<tr>
<td>CommLd</td>
<td>PCEnable &amp;&amp; CM.Op == 8</td>
<td>This is the enable signal for the Communication Register. It is asserted in the compute mode for a “STORE” instruction.</td>
</tr>
<tr>
<td>OWC[55:55]</td>
<td>(PE-Hit &amp;&amp; OvrWrd.Op == 1 &amp;&amp; (OvrWrd.Bank &lt; 5</td>
<td></td>
</tr>
<tr>
<td>OWrdoOutSel</td>
<td>if (PCEnable) then 100 elsif (PE-Hit &amp;&amp; OvrWrd.Op == 1 &amp;&amp; OvrWrd.Bank &lt; 4) then 001 elsif (PE-Hit &amp;&amp; OvrWrd.Op == 1 &amp;&amp; OvrWrd.Bank == 4) then 010 elsif (PE-Hit &amp;&amp; OvrWrd.Op == 1 &amp;&amp; OvrWrd.Bank == 4) then 011 else 000</td>
<td>This signal is used to select the lower 40-bits of the final output OvrWrd of the PE.</td>
</tr>
</tbody>
</table>

Table 4.10 Program Counter, Communication Unit and Output Unit control signals
4.4 Data and Control Flow within a Single Processing Element

We outline the data and control within an individual PE by considering two instructions, one from each operating mode. To understand the data and control flow correctly, we will divide the PE into different blocks: the OvrWrd Block, which consists of the 56-bit OvrWrd register which stores the computed OvrWrd (it should be recalled that the OvrWrd is computed by the bit-wise OR-ing of the OvrWrds received from the North and West neighbors), the PC Block, which consists of the PC and its supporting control logic, the CM block which consists of the control memory, the Functional Unit which consists of the operand memory banks and the FPU and finally, the Output Block, which consists of the output selector logic. We consider a PUT instruction and a MAC instruction to describe the data and control flow in the Override and Compute modes respectively.

Override Mode - PUT 1, 0 4, 0 <instruction>

This instruction is used to write the value <instruction> into the CM. The logical flow of steps for this instruction is as follows:-

- **OvrWrd Block** – The OvrWrd is obtained by bit-wise OR of the words from North and West neighbors. The OvrWrdCtrl and OvrWrdData fields are identified. For this instruction, OvrWrdCtrl = PUT 1,0 4,0 and OvrWrdData = <instruction>.
- **PC Block** – In override mode, PCInSel = 0, so input to the PC is OvrWrdData[6..0]. The PC is viewed as bank 7. Since this is a write to bank 4, PCLd = 0.
- **CM Block** – This instruction performs a write to the control memory (bank 4). Hence, CMAAddrSel = 1, CMOE_n = 1, CMWS_n = 0, CMWrt_n = 0. The value <instruction> gets written to the CM at the address given by OwdCtrlAddr (which in this case is 0).
• **Functional Block** – Since this instruction does not concern any of the operand memory blocks, $OM_x_addrSel = 0, OM_xOE = 1, OM_xWrt_n = 1, OM_xWS_n = 1$, where $x=0,1,2$ or 3. Also, in override mode, $PCEnable = 0$, so the functional unit is disabled.

• **Output Block** – In override mode, the PE does not compute anything but just passes on the received data to its East and South neighbors. Hence $OvrWrdOutSel = 0$.

**Compute Mode - MAC R1, R2, R3, R4 where R1=0,0 R2=1,7 R3=2,12 R4=3,5**

This instruction calculates $R1 \times R2 + R3$ and places the result into $R4$. The bank and addresses pointed to by each of $R1, R2, R3$ and $R4$ are as mentioned above. The control and data flow are as:

• **OvrWrd Block** – Since PE is in compute mode, OvrWrd Block is disabled.

• **PC Block** – Say that this instruction is written at address 0 in the CM. Hence the PC output is $(0000000)_b$. $PCCnt = 1$, so that PC is auto-incremented to point to the next instruction.

• **CM Block** – In compute mode, $CM_addrSel = 0$ so that address given to CM is the PC output. $CMOE_n = 0, CMWrt_n = 1, CMWS_n = 1$. The instruction is read from $(0000000)_b$ and then decoded to obtain the values of $CM.Op$ - the opcode, $CM.R1Bank$, $CM.R1Addr$, $CM.R3Bank$, $CM.R2Addr$, $CM.R3Addr$, and $CM.R4Bank$, $CM.R4Addr$.

• **Functional Unit**
  
  o The operands are first read from the banks 0, 1 and 2. Hence $OM0_addrSel = 0$ (since $R1=0,0$), $OM1_addrSel = 2$ (since $R2=1,7$), $OM2_addrSel = 3$ (since $R3=2,12$) and $OM3_addrSel = 4$ (since $R4=3,5$).
For banks 0, 1, 2 $OM_{xOE_n} = 0, OM_{xWS_n} = 1, OM_{xWrt_n} = 1$ and for bank 3 $OM_{3OE_n} = 1, OM_{xWrt_n} = 0, OM_{xWS_n} = 0$. Let us denote output of $OM_0$ as $D_0$, $OM_1$ as $D_1$ and $OM_2$ as $D_2$.

In compute mode, $R1SrcSel = 0$, so output of that multiplexer is $CM.R1Bank$ (i.e. 0) and consequently, output of the R1-multiplexer is $D_0$. Similarly, output of the R2-multiplexer is $D_1$.

Since this is a MAC instruction, $MACOp = 1$, so output of the bottom multiplexer is $CM.R3Bank$. Consequently, the output of the $R2orR3$-multiplexer is $D_2$. Now we have all the operands to execute the instruction.

$R1$ and $R2$ are provided as inputs to the multiplier and the inputs to the add/subtract block are the output of the multiplexer and $R2orR3$.

For MAC operation, $ResultSel = 1$, so that output of the add/subtract block is selected as the result to be written back.

During write-back, $OM_{3Wrt_n} = 0, OM_{3WS_n} = 0, OM_{3OE_n} = 1$ so that the result is written into $OM_3$.

- **Output Block** – In compute mode, $OwrdrOutSel = 4$, so that output of the $COMM$ register is appended to $OvrWrdCtrl$ and given as output to East and South neighbors.

All the operations are completed in a single clock cycle. The worst case execution time for a single PE was determined by the floating point divide unit. In the current implementation, the minimum clock period to complete all operations was found to be 110 ns or a clock rate of 9.1 MHz. Since all the 4 PEs perform operations in parallel, total speed of the system will be 36.4 MHz. Since each PE performs exactly one FLOP per clock cycle, we get a floating point performance of 36 MFLOPS. Clearly, this speed will increase as more PEs are accommodated on the die.
4.5 Summary

The design and instruction set architecture of the proposed custom integrated circuit were presented in this chapter. The system is a 2D array of processing elements, each with private operand and control memories, scalar processing and control. The final system will consist of a number of these ICs populating a custom PCB with a parallel interface to a host machine. The processing elements communicate between themselves using a programmable interconnect and the Nearest Neighbor interconnection strategy.

This IC has been designed as a prototype. The initial emphasis was on correctness with speed only the second concern. Simulations of the current system clearly show a potential for large speed-ups in floating point performance as more processing elements are accommodated on a single die. Most of the implementation choices were constrained by the available area. Several modifications were made in order to reduce the pin-count to accommodate all the logic onto a single die of 7.5 sq. mm. The number of processing elements would be much higher in a production system.
CHAPTER 5
A STANDARD CELL-BASED VLSI DESIGN FLOW OF THE CUSTOM INTEGRATED CIRCUIT

5.1 Introduction

A VLSI design flow typically consists of a series of steps, starting from the system specification to the mask layout of the chip. There are two primary methodologies to VLSI design: bottom-up or full-custom design and top-down or standard cell based design.

The bottom-up design methodology is based on the manual construction of the circuit building blocks, i.e. logic gates, at the transistor and the mask-layout level. These blocks are then merged together to form the complete design hierarchy according to the functionality of the system. The bottom-up design flow provides a lot of flexibility at the lower levels since the designer has full control over issues such as transistor size optimization, parasitic minimization, meeting timing and power constraints, etc. Hence this methodology is better suited for the design of very dense, high-performance digital, analog and mixed-circuit modules. The disadvantage of full-custom design is its design complexity and long time-to-market even for chips of moderate sizes which makes it a highly specialized and time-consuming process.

The top-down design methodology is based on computer-aided synthesis of gate-level netlists using a behavioral or structural HDL description. The functionality of the system is specified using a VHDL or Verilog description and Computer Aided Design programs are then used to obtain a generic gate-level netlist. The netlist is then mapped to a standard cell library which consists of the basic building blocks and is usually provided by vendors. This
methodology is well-suited for digital designs with short time-to-market and moderate area-performance requirements.

In this chapter, we describe a generic VLSI design flow using standard cell libraries. The design flow is not restricted to this particular design but can be applied, as is, for any other design which starts with a HDL representation and uses standard cell libraries. The current design has been completely represented in VHDL. We have used the TSMC 0.18 um technology standard cell libraries for logic, I/O and the SRAM memory provided by Artisan Inc. under the MOSIS Educational Program.

Figure 5.1 Top-down (standard cell based) Design Methodology
The Cadence CAD Tool Suite was used for the HDL-to-layout design flow. Note that the information about the various commands to be used can be readily found using the Cadence documentation. Our purpose in outlining the flow is to give an overview of the tools used and mention some subtle points which are not included in any documentation but come only through the experience and usage of these tools. We hope that these points will be beneficial for future designers who plan to follow a similar flow.

5.2 Overview of the Implementation

This section provides an overview of the various steps and the CAD tools used in this design. Each of these steps is then elaborated in later sections with illustrations wherever necessary. Figure 5.1 provides a flow-diagram of the design flow from HDL-to-layout.

5.2.1 HDL Representation and Simulation

A highly modular VHDL representation was used to describe the design. Specifying the entire design using smaller and simpler modules made the design and, more importantly, the debugging a lot easier. We were also able to predict the performance of the entire system by analyzing the performance of important modules like the floating point arithmetic unit. Each module was thoroughly tested and debugged separately before being integrated into the larger design. This helped minimize mistakes and saved debugging time in the later stages. Appendix B contains a design hierarchy tree which shows the various modules and their dependencies.

NCVHDL™ and NCSIM™ [45] were used for the compilation and simulation of the system using test-benches generated automatically using a C program. Considerable effort was spent on verifying the correctness and fine-tuning the floating point unit as its performance is critical for the entire system. For testing the PE array, we simulated a well-known algorithm, Conway's Game of Life [46] which provided a good test of the inter-PE communication and the
memory units. Along with the HDL representation, we developed an input-driven simulator in C
to model the behavior of the system. The hardware computations were compared against the
simulator results to verify the correctness of the system. To aid in quick generation of test
benches we have also developed an assembler using LEX and C for the PE array.

5.2.2 Logic Synthesis and Datapath Design

Automatic logic synthesis of the HDL representation and its mapping to the TSMC 0.18
um technology was accomplished by using Ambit BuildGates™ [45]. In this step, the HDL
representation is first converted into a generic netlist which is then mapped to the target
technology viz. TSMC 0.18 um. BuildGates™ performs this mapping using the library
information from the cell library and user-defined timing, area and power constraints. The output
of this stage is a technology-specific netlist represented either in Verilog or in the Design
Exchange Format (DEF).

5.2.3 Placement and Routing

This step involves floorplanning of the die area and the placement & routing of the
standard cells and macro blocks. This was done using Silicon Ensemble-Physically
Knowledgeable Synthesis (SE-PKST™) [45] platform within the Cadence Tool Suite. SE™ was
also used for clock tree generation, power connections and the timing analysis of the design. The
primary input for this stage is the synthesized netlist in Verilog or DEF format. In addition,
design data like component names, their areas, power and timing values etc. are obtained using
various data files provided by the vendor.
5.2.4 Adding Bonding Pads

The final step before tape-out is the addition of I/O and bonding pads to the design. Cadence provides no automated tool for this step. Hence bonding pads were added and connected manually using a layout editor. Since the design was pad-limited and had to be fit into an area of 7.5 sq. mm., we had to modify the design slightly to lower the pin-count. The final design, after design rule check, fit into an area of 7.234 sq. mm. and had 84 pins. The generation of the bonding diagram and selection of the package was done by MOSIS. The chip was then submitted to MOSIS for fabrication.

The following sections elaborate each of the above steps in detail in terms of the tools, their design flows and some commands used. The Cadence Tool Suite consists of a collection of tools that perform the required operations to get through the design flow. Each of these tools is optimized for a certain task and it is helpful to represent the design in a certain format for the tool to perform efficiently. Other file formats are used to specify the timing details, power consumption details and the area details for each standard cell used in the design. These files are typically provided by the vendor (Artisan Inc. in our case) along with the standard cell libraries.

5.3 System Specification and Simulation (NC-VHDL™, NC-ELAB™, NC-SIM™)

The simulation flow in the Cadence environment consists of three stages - analysis and compilation (NC-VHDL™), design elaboration (NC-ELAB™) and simulation (NC-SIM™). Each of these steps is elaborated in the following sub-sections.
5.3.1 HDL Analysis and Compilation

This was accomplished using NC-VHDL™, an event-driven, native compiled code (NCC) compiler. Internally, NC-VHDL™ contains a parser which performs syntactic and static semantic checking on the input VHDL design units and produces an intermediate representation of the input source text. If no errors are found, these intermediate objects are stored in a single packed library database file in the LIBRARY.TOP_CELL:VIEW format in the work library directory. The compiled code and the simulation snapshot is stored in the work library (usually the current working directory). The NC-VHDL™ design flow is illustrated in Figure 5.2.

5.3.2 Elaboration

Before the generated library database can be simulated, the design hierarchy defining the model must be elaborated using NC-ELAB™. NC-ELAB™ constructs a design hierarchy based on the instantiation and configuration information in the design, establishes signal connectivity, and computes initial values for all objects in the design. The elaborated design hierarchy is stored in a simulation snapshot file (again in the LIBRARY.TOP_CELL:VIEW format). The snapshot is the representation of the design at event time t = 0. The NC-ELAB™ design flow is illustrated in Figure 5.3.

![Figure 5.2 NC-VHDL™ Design Flow](image-url)
Figure 5.3 NC-ELAB™ Design Flow

Figure 5.4 NC-SIM™ GUI
5.3.3 Simulation of the snapshot

NC-SIM™ loads the simulation snapshot generated by NC-ELAB™ as its primary input along with other intermediate objects referenced by the snapshot. In the case of interactive debugging, HDL source files and script files may also be loaded. The outputs of simulation are controlled by the model or the debugger. These outputs can include result files generated by the model, Simulation History Manager (SHM) databases, or Value Change Dump (VCD) files. One of the main advantages of NC-SIM is its interface with the SimVision™ environment as illustrated in Figure 5.5. This allows the user to view the current design hierarchy in a graphical tree representation, analyze the waveforms of various signals and trace backwards through the design to detect any signals with questionable values. Thus NC-SIM™ provides a comprehensive simulation environment and serves as a powerful debugging tool.

![Figure 5.5 The SIMVISION™ waveform and debug environment](image)

Figure 5.5 The SIMVISION™ waveform and debug environment
5.4 Logic Synthesis and Optimization

Automatic logic synthesis and optimization of the HDL representation was performed using Ambit BuildGates. BuildGates™ is a high performance synthesis tool which can also perform full chip static timing analysis, automatic time budgeting, supports both top-down and bottom-up synthesis flows and can generate information in a variety of file formats which makes it easy to integrate with the physical design tools which come next in the design flow. The synthesis steps using BuildGates™ are outlined in Figure 5.6.

BuildGates™ accepts the hardware representation in VHDL, Verilog, or EDIF HDLs and generates the corresponding hardware implementation in the form of a generic, technology-independent netlist. The resulting generic netlist can then be optimized and mapped to a target technology. A GUI version of the BuildGates™ software was used for the design. Figure 5.7 illustrates the BuildGates™ GUI. The following sub-sections provide a walkthrough of the various design steps with relevant screen-shots illustrated wherever necessary.

![Logic Synthesis Flow using Ambit BuildGates™](image)

Figure 5.6 Logic Synthesis Flow using Ambit BuildGates™
5.4.1 Reading Technology Libraries

To make architectural decisions about the hardware, BuildGates™ needs information about the components in the cell library used for the design. Hence reading the technology library information is the first step in the design process. BuildGates™ can read library information in ALF (Ambit Library Format) or TLF formats. This is accomplished by the `read_alf` and `read_tlf` commands.

5.4.2 Reading Design Data

The next step is to read the design data in the form of HDLs. This is done by the `read_vhdl` and the `read_verilog` commands. In VHDL, for any one design, its entity must be read prior to its architecture and all packages and package bodies must be read prior to reading in the entities and their architectures.
5.4.3 Building a Generic Netlist

A generic netlist can now be generated using the `do_build_generic` command. On execution of this command, BuildGates™ generates a control/data flow graph to analyze the input design, determines the number of latches and flip-flops to store data, determines the sizes and types of components required to implement the logic and generates the appropriate control and interconnect logic. In the current system, memories were part of the Artisan Intellectual Property (IP) and hence were treated in the synthesis as “black boxes” i.e. only their entity declarations were read into BuildGates™ as details about their architecture were hidden. The generated netlist is completely generic and can be mapped to any target technology. Figure 5.8 illustrates the schematic of the generic netlist of the design.
5.4.4 Setting Constraints

This is the first step in mapping the generic netlist to the process technology. In this case, process data for the TSMC 0.18um technology was provided using the TLF files provided by Artisan. BuildGates™ allows timing constraints to be set only at the top level of the design. Hence the first step is to specify top timing module explicitly by using the `set_top_timing_module` command. The typical constraints to be set include the clock periods, the input clock delay which signifies the delay from the clock source to the clock port of the design and the setup and hold times of I/O pins. Timing constraints for black-box units, like the memory unit, have to be specified explicitly, either by using a GUI form or by sourcing a TCL script into BuildGates™. The GUI form for setting the timing constraints is illustrated in Figure 5.9.
5.4.5 Optimizing the generic netlist

The final step in the mapping process of the generic netlist to the target technology is optimization using the \textit{do\_optimize} command. This comprises of several intermediate steps which are known as "transformations". Transformations are commands which change the structure of a logic block. The optimization command repeatedly performs these transformations depending upon a global cost function which is set by the various constraints set in the previous step and by the technology-specific data. When \textit{do\_optimize} is executed, several optimizations like logic optimizations, structural optimizations, clock tree optimizations and timing optimizations are performed. These optimizations may involve dissolving the hierarchy in the generic netlist for efficient mapping to technology-specific cells, adding buffers to maintain signal integrity, adding buffers and drivers to propagate the clock signal, reclaiming area freed during optimization, fixing multi-port nets, performing timing correction to remove hold and setup time violations and fixing any design rule violations. The output of the optimization step is an efficient technology-specific netlist subject to certain user-defined and technology-specific constraints.
5.5 Placement and Routing

Placement of standard cells and routing the design is the most complex and time-consuming step of the design process. In the Cadence environment, this was done using the Envisia™ Silicon Ensemble™ (SE) place-and-route software. SE™ is a complete system that can perform all the complex tasks needed to create the physical layout of an IC. SE™ is composed of various sub-systems illustrated in Figure 5.11. Specifically, it consists of tools to perform floorplanning, block and cell placement, clock-tree generation, timing analysis, routing and RC extraction.

![Figure 5.11 Components of SE™ – A Place-and-Route Tool](image)

Some of the features of SE™ include timing-driven support using the high-level timing constraints provided by the TLF file, placement compaction before routing, area-based routing for denser designs, Engineering Change Option (ECO) support for accurate incremental changes and an ultra-fast router for multi-million-gate designs. A timing-driven SE™ design flow illustrated in Figure 5.12 was used in our design. The following sub-sections elaborate each step in detail.
5.5.1 Importing Design Data

The first step is to import the library and design data. The library data must be in the LEF format. SE™ does not accept any other formats for standard cell library information. The design data is imported in the form of Verilog description of the cells, I/O pads and the design. Usually, I/O, power and corner pad connections are not included during the logic synthesis phase. Hence the netlist obtained from the synthesis tool has to be appended with a new top-level module which explicitly specifies the connections of the I/O pads to all the input/output pins.

Figure 5.12 Timing-Driven Place-and-Route using SE™

For a timing-driven design flow, timing and boundary constraints need to be imported using the TLF and GCF files respectively. The latest SE™ release supports reading of these constraints using a single GCF file since a GCF file can contain pointers to the relevant CTLF (compiled TLF) files. Any incremental information like corner pads (which cannot be included in
the netlist), has to be read in using a DEF file for the corner pads. Filler cells (described in Section 5.6.8) are not part of the design data.

5.5.2 Initialization

The next step is to initialize the floorplan. Initialization sets up the physical design space (or the die area) based on the library and design data read in. The floorplanner creates a core area with rows or columns, sets up I/O rows around the core area and sets up tracks for cell placement. The floorplanner can also be used to specify the clearance between the macro block boundaries and the cell tracks, known as the block halo. This is crucial to make reliable power and ground connections to the corresponding rings around the macro blocks. We found that SE™ supports only primitive floorplanning capabilities. Making any changes to the floorplan suggested by SE™ is a cumbersome process and usually involves making manual adjustment of the co-ordinates. Figure 5.13 illustrates a GUI dialog box to set parameters for the floorplan of the current design and provides details about number of standard cells, I/O pads, corner pads, the aspect ratio of the design and the total die area.

Figure 5.13 Floorplan initialization dialog box
5.5.3 I/O Pins and Macro Blocks Placement

After initialization of the basic floorplan, the next step is the placement of I/O pads and macro blocks. The I/O pad placements can be dictated by an I/O constraint file. The simplest way is to let SET™ generate a dummy I/O constraint file (no constraints) and then edit the file to specify the required constraints. The macro blocks, for example SRAM units, can then be placed. Timing-driven placement usually provides much better results in terms of reduction in congestion during placement of standard cells and should be followed whenever possible. To make any changes in the block placement by SET™, the designer can either manually edit the DEF file or use the MOVE command to move the blocks. The final step in this phase is to trim or update the rows surrounding the placed blocks. This is critical for connecting the block ring to the power and
ground lines. Figure 5.14 illustrates the die area after placement of the macro blocks. The black boxes are the macro blocks for the SRAM while the rectangles around the die periphery are the macro blocks for the I/O pads. The figure also illustrates the rows where the standard cells will be placed in the cell placement stage and the power and ground rings of each macro block.

5.5.4 Planning Power

This step adds the power rings and stripes for power and ground connections. Additionally, a quick cell placement can be done to perform static power analysis. This allows the user to create a custom power plan and dictate cell placement to optimize power. Power and ground rings can be added using any metal layer, but the defaults are METAL1 (horizontal) for power and METAL2 (vertical) for ground connections. In addition, to ensure uniform distribution, power and ground stripes may also be added though they are usually a source of higher area without much gain. Power stripes were not used in our design.

5.5.5 Cell Placement

Internally, SE™ uses the Ultra Placer™ for placement of standard cells, I/O pins and blocks. It performs concurrent optimization of placement and timing and can resolve signal and design integrity issues. Some additional features of the Ultra Placer™ include

- The timing data from the timing analyzer is accessible to the placer. This dramatically shortens the design time by eliminating the need for the generation of stage-based constraints in the system-level constraints flow and provides greater accuracy.
- It provides flexibility in the form that the designer can place some select cells close to the I/O cells and fix their locations so that they will not be moved later.
- It supports complete timing driven placement by using the timing information from the GCF files. It also has the ability to run timing optimizations during or after placement where it can remove setup and hold violations.
- It reduces congestion and improves routability
- It has the ability to read a Reduced Standard Parasitic Format (RSPF) which allows the placer to perform optimizations based on parasitics extracted after final routing.

![Figure 5.15 Cell Placement in the core area](image)

The optimizations are performed by the placer using the simulated annealing technique. The placer performs a global placement in the first pass and then performs optimization in successive passes until the timing and area constraints are met. The placer also checks for design rule violations during placement. Certain timing optimizations and signal integrity checks are
performed at every pass to maintain the signal strength. It also simultaneously checks and fixes any setup, hold, minimum transition and maximum transition violations.

We found that the default environment variable values worked well for global placement. An important point to be remembered is that sufficient area must be left around the macro blocks after block placement to allow for power and ground connections. The power and ground wires are typically wider and hence need more routing space. To avoid manual connection of power/ground wires after placement, it is advisable to leave enough space around blocks. Figure 5.15 illustrates the die area after the placement of standard cells. All the standard cells are placed in the core area. Additional cells, called filler cells, need to be added to maintain integrity of power and ground rails.

5.5.6 Timing Analysis

At this stage, all the I/O pads, cells and blocks are placed in the die area. Timing analysis at this stage provides details about the timing of the individual paths of the design. This enables the designer to determine if the design needs any modification to meet the timing requirements. The timing information can also be written out in the Reduced Standard Parasitic Format (RSPF) or Standard Delay Format (SDF). Although timing analysis can be performed at various stages of the design, the accuracy of this analysis depends upon the amount of information available.

5.5.7 Clock Tree Generation (CT-GEN™)

This is an optional step in the design process but is highly recommended for large designs to preserve the signal integrity of the clock signal. In our case, this step was crucial as we were using a single clock signal for the entire chip. Clock tree generation is an iterative procedure and has to be repeated till the timing requirements are met. CT-GEN™ allows the designer to
generate post-placement buffered clock-trees that help minimize problems associated with clock skew. CT-GEN™ -

- Traces through any input pins not declared as CLOCK pins in the TLF or CT-Gen technology files.
- Removes any existing buffers or inverters that were traced through.
- Generates a clock tree topology (that is, the number of buffer levels and the type and number of buffers required at each level) that best satisfies user-specified delay and skew constraints.
- Groups the clock leaf pins (that is, CLOCK input pins) into clusters defined by the clock tree topology so that pin and wire loads are balanced.

The clock tree process is completed by writing out a DEF file of the modified design and doing an ECO using this file. We found that SE™ does not actually recognize the added buffers and inverters until this step is performed.

5.5.8 Adding Filler Cells

Filler cells are dummy cells which are used to preserve the signal integrity in the "empty" spaces of the die. During the placement stage, many areas of the die can remain empty, i.e. no cells or blocks are placed. In these areas, it is necessary to place the filler cells for proper routing of power and ground signals. Ignoring this can be fatal for the chip as the power and ground signals will not be routed properly. Pad filler cells are also needed between the I/O pads to ensure the continuity of the power signals around the pad-ring.

5.5.9 Route

At this stage, the design has been completely placed and is ready for routing. SE™ uses a very fast routing engine called Warp-Route or WROUTE™. The usual process is to perform the
routing of the clock and power signals in a global pass and then perform detailed routing of the cells and I/O pins. A full routing pass consists of global routing, final routing and search-and-repair routing. During global routing, the router finds generalized pathways, without laying down actual wires, and makes iterative passes to optimize the global routing, shorten wire length, and minimize the use of vias and updates the congestion map. The router interconnects the signal nets defined in the NETS section of the DEF file for the design, based on the supply and demand of routing tracks. During final routing, the router follows the global routing plan and lays down actual wires to connect the pins to their corresponding nets. It also removes routing violations and minimizes wire length and the number of vias.

![Figure 5.16 Routing using WROUTE™](image)

The default environment settings in SE™ work well with WROUTE™. If there are some violations after search-and-repair stage then WROUTE™ can be performed once again to remove them. SE™ also allows a stand-alone search-and-repair pass after the routing is complete, but we
found that running WROUTE™ again is a much better although slower way of removing any violations. Figure 5.16 shows a fraction of the die area which illustrates the result of running WROUTE™ on the design. The rectangles marked by yellow are the memory macro blocks. Since we did not possess the internal details of the memory blocks, these were treated as black boxes and connections were made by SET™ only to the power and ground rings of these blocks. The rest of the die area is composed of the standard cells and interconnect.

The final step in the P&R design flow is to verify the connectivity and geometry of the design. If all goes well, a GDSII layer map file can be used to write out the final GDSII file. The Ultra Placer™ and WROUTE™ facilities in SET™ make it a very attractive tool for fast SoC designs. The complete SET™ run-time for the current design was typically 8-11 hours depending upon the number of times WROUTE™ had to be re-run to remove any violations. Using timing optimization, the design time was increased to 14 hours, but the results obtained were much better.

5.6 Adding Bonding Pads

The final step before tape-out of the mask layout for fabrication is the addition of bonding pads. Bonding pads are responsible for the passage of signals between the chip and the outside world. Thus they act as the interface for all I/O, clock, power and ground signals from external sources/sinks to/from the chip. During chip fabrication, thin gold wires are connected from the external pins of the chip to these bonding pads. The silicon nitride substrate layer is breached at the position of each bonding pad to supply the signal on the gold wire to the chip. The proper placement of bonding pads is extremely important for the correct working of the chip. There are two main styles for placement of bonding pads: staggered bonding and inline or linear bonding. These are illustrated in Figure 5.17.

Staggered bonding is used in a pad-limited design to minimize the area required by the bonding pads. As shown in Figure 5.17, staggered bonding allows abutment of I/O pads so that
the placement area is minimized. The disadvantage of staggered bonding is the complexity to bond to these pads during the actual fabrication process.

Inline or linear bonding involves the linear placement of bonding pads to each I/O pad. Due to design rules concerning the top metal layer of the bonding pads, the I/O pads cannot be abutted if linear bonding is used. Inline placement is much easier, results in a relatively simple bonding diagram and is easy to handle during the fabrication process. Hence inline bonding pad placement was followed for the current design.

Figure 5.17 Bonding Styles

There are no automated tools in the Cadence Tool Suite for automatic placement of bonding pads. This was done manually using the Virtuoso Layout Editor™. Each bonding pad
consists of a pad boundary layer which connects to the I/O pad, an outer metal layer that connects to the gold wiring and a glass layer to provide insulation against signal coupling. The spacing between the I/O pads and that between the bonding pads needs to be carefully controlled according to the packaging design rules. This distance is specified in terms of the distance between the centers of the individual pads and is known as the pad pitch.

In this design, the pad pitch was maintained at 80 um. This allowed a spacing of 9 um between the bonding pads at the widest point. The pad pitch for bonding pads was maintained at 60 um with a pad-to-pad clearance of 5 um. In addition, the clearance between the die edge and the corner of the first pad edge was kept at 400 um. These distances were in accordance with the packaging design rules provided by MOSIS.

Figure 5.19 illustrates a snapshot of the design after placement of the bonding pads using both the bonding styles. Figure 5.18 illustrates an overview of the design after it was merged at MOSIS.

5.7 Tape-Out

The preparation of the design (merging of standard cell layouts and final connectivity checks) was handled by MOSIS. The GDSII file of the final design was submitted to MOSIS along with the information about the standard cell libraries and memory units. Additional dummy filler cells of polysilicon and the top metal layer (METAL 6) were added to make the density of these layers sufficient for the TSMC process. The bonding diagram and package selection was handled by MOSIS. The final die size was 7.234 mm x 7.234 mm. with 84 pins to be packaged in a ceramic LCC (leadless chip carrier) package. The chip was submitted to the fabrication run at MOSIS.
5.8 Summary

A VLSI design flow using standard cell libraries was illustrated in this chapter. The aim was to familiarize the reader with the different aspects of VLSI design, to illustrate some of the CAD tools involved and to mention some points which are important, but are not mentioned in any tool documentation.

The top-down design methodology has become more and more popular in recent years due to the sophistication of CAD tools. This has drastically reduced the time-to-market of chips required for embedded systems. The main reason for the popularity of this approach is the widespread use of low complexity chips in embedded systems like washing machines, calculators, digital cameras, automobiles etc. These chips have moderate area-performance requirements, but their main constraint is the time-to-market. The capability of CAD tools and standard cell libraries and, more importantly, their ready availability has made VLSI design easier. We hope that the details presented here will be helpful to future designers who intend to follow a top-down design approach.

Figure 5.18 Final Die Mask after merging standard cell layouts
Figure 5.19 Final Design with two different bonding styles.
CHAPTER 6
CONCLUSIONS AND FUTURE WORK

This work presents the philosophy, design choices, instruction set architecture and implementation details of a Reconfigurable Multi-Processor System for Floating Point Intensive Algorithms. Many interesting problems involve computationally intensive iterative floating point algorithms. A particular class of such applications involving large sparse linear systems was studied. It was found that current implementations popularly use Explicit Integration Methods to solve these problems. The numerical instability of Explicit Methods for large stepsizes is well-known, but these methods are preferred over Implicit Methods as they involve considerably less computation. A first-generation special-purpose hardware system which implemented forward Euler integration on an Altera FPGA [35] was discussed. The lessons learnt from the implementation of the first-generation system were used in the design of our proposed system. Specifically, the current system is a coarse-grain structure of processing elements optimized to provide high performance for floating point intensive algorithms. The designed architecture is a 2D array of processing elements, each with private operand and control memories, scalar processing and programmable interconnect. A standard cell-based VLSI design flow using CAD tools from the Cadence Tool Suite was presented. This project provides numerous avenues for further research. In the short term, the immediate concern will be the design and assembly of a custom PCB to connect the fabricated chip to a host system. This will enable us to perform testing and performance evaluation of various algorithms to support our claims.
Specifically, we would first like to verify the operation of the chip by testing the local memories, control logic, interconnect between the processing elements and then individual processing elements themselves. The test routines would then map several interactive graphics applications which require solving of sparse linear systems. Such large sparse linear matrices for purposes of testing and simulation are available at Matrix Market [47].

Certain performance optimizations can also be made to reduce the run-time of the system. Specifically, the floating point divide unit determines the worst-case execution time of the system. The divide unit can be made faster by using look-up tables for common operand values rather than computing them. This simple modification would reduce latency at a low price in terms of area and power consumption. To make mapping of applications easier, we would also like to focus on the design of a compiler which will modify the application kernels to take advantage of our system. We also plan to implement a scheduler to aid in determining an optimal schedule of the operations to be performed.

In the long term, we would like to concentrate on designing later generation systems to provide higher performance. The design of the current IC was constrained due to area and cost constraints. Simulations have shown the potential for increased speed-up by integrating more processing elements on a single die. Experimental systems with multiple ICs populating a single PCB and collaborating to handle much larger problems could also be constructed. This would also allow experimentation with other techniques which could not be mapped to the current IC. The primary aim of our research would be to build a system with the capability of rendering animations of a high quality, such as those seen in feature films, to the home PC.
REFERENCES


[46] Wonders of Math – The Game of Life by Dr. John Conway,


A.1 Representation of Single Precision Numbers

In computer arithmetic, floating point representation is used for storage of and operations on real numbers. The floating point system represents real numbers in scientific notation where the numbers are represented using a base and an exponent. We will study the representation and operations of single-precision floating point numbers as per the IEEE 754 Floating Point Standard.

In the IEEE single precision format, a floating point number is represented as a 32-bit word. The division of various bits and their interpretation is as illustrated in Table A.1. We will assume the leftmost bit as bit 31 (MSB) and the rightmost bit as bit 0 (LSB).

<table>
<thead>
<tr>
<th>Bit</th>
<th>31</th>
<th>30 – 23</th>
<th>22 – 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>Interpretation</td>
<td>Sign (S)</td>
<td>Exponent (E)</td>
<td>Mantissa (M)</td>
</tr>
</tbody>
</table>

The Sign bit is used to determine the sign of the number. 0 indicates a positive number while 1 indicates a negative number. Since the exponent needs to represent both negative and positive exponents, a bias is added to the actual exponent to obtain the stored exponent. For single precision numbers, the bias is 01111111_2 or 127_d. Thus, to store a zero exponent, 0 + 127 = 127 is stored in the exponent field. A stored exponent value of 130 indicates an actual exponent value of 130 – 127 = 3.
The mantissa or the significand stores the part of the number after the decimal point. Since all the numbers are stored in binary, the only non-zero leading bit is 1. Hence, this leading bit is assumed implicitly and is known as the “hidden 1”. Thus we can get a resolution of 24 bits while actually storing only 23 bits.

A.2 Range of the Representation

The idea of the floating point numbers is that they are stored as 32-bit words where different bit-fields are interpreted differently by the hardware. Specifically, the value $V$ of a 32-bit word representing a floating point number can be obtained by the following rules:

- If $E = 255$ and $M$ is nonzero, then $V = \text{Nan} ("\text{Not a number}")$
- If $E = 255$ and $M$ is zero and $S$ is 1, then $V = -\text{Infinity}$
- If $E = 255$ and $M$ is zero and $S$ is 0, then $V = \text{Infinity}$
- If $0 < E < 255$ then $V = (-1)^S \times 2^{(E-127)} \times (1.M)$
- If $E = 0$ and $F$ is nonzero, then $V = (-1)^S \times 2^{(-126)} \times (0.M)$. These are “unnormalized” values or “denormals”.
- If $E = 0$ and $M$ is zero and $S$ is 1, then $V = -0$
- If $E = 0$ and $M$ is zero and $S$ is 0, then $V = 0$

Table A.1 illustrates some of the numbers represented by the single precision format. The table also illustrates the range of numbers that can be represented.
Table A.2 Range of single precision representation

A.3 Floating Point Add/Subtract

Figure A.1 illustrates the flow diagram for performing addition and subtraction of floating point numbers. We will briefly outline the algorithm. A more thorough treatment can be found in [41, 42, 43]. For the following algorithms, N1 and N2 are the input floating point operands. S_i, E_i and M_i denote the sign, exponent and mantissa of the number N_i. Note that during unpacking (operation performed by the “Unpack Operands” block) the hidden 1 is made explicit which means that M_i already contains the leading 1. Following steps need to be performed for floating point addition:

1. If E_1 < E_2, swap operands, tentatively set E_result to E_1.
2. If S_1 ≠ S_2, replace M_2 by its 2’s complement.
3. Set D = E_1 − E_2.
4. Pre-normalize M_2 by shifting it D places to the right. Shift in 1’s if M_2 was complemented in step 2. Denote this by M_{2\text{norm}}.
5. From the bits that were shifted out, set GUARD_BIT to the MSB, ROUND_BIT to the next MSB and STICKY_BIT to the OR of the rest of the bits.
6. Set CARRY_OUT, SUM = M₁ + M₂norm where + represents bit-wise addition.

7. If S₁ ≠ S₂, MSB(SUM) = 1 and CARRY_OUT = 0, then SUM is negative. Replace SUM by its 2’s complement. GOTO step 10.

8. If S₁ = S₂ and CARRY_OUT = 1, shift SUM right by 1 and shifting in CARRY_OUT. GOTO step 10.

9. If S₁ ≠ S₂ and CARRY_OUT = 0, then shift left until SUM is normalized. On the first left shift, shift in GUARD_BIT followed by zeros.

10. Adjust ROUND_BIT and STICKY_BIT as follows: If SUM was shifted right in step 8, set ROUND_BIT = LSB(SUM) and STICKY_BIT = (GUARD_BIT OR ROUND_BIT OR STICKY_BIT). If there was no normalizing shift, set ROUND_BIT = GUARD_BIT and STICKY_BIT = ROUND_BIT. If there are two or more normalizing left shifts, set ROUND_BIT = 0 and STICKY_BIT = 0.

11. Round SUM. Round-to-nearest strategy was followed in our hardware. Adjust the exponent accordingly.

12. Compute the correct sign of the result.

13. Pack the result to get the 32-bit sum or difference.
A.4 Floating Point Multiply and Divide

Floating-point multiplication involves multiplication of the unpacked mantissas (M₁ and M₂) and addition of their exponents. A normalizing shift and rounding have to be performed to get the final answer. The multiplication algorithm is as follows:

1. Unpack the input operands to obtain S₁, S₂, E₁, E₂, M₁ and M₂.
2. Multiply M₁ and M₂. In our system, a 24-bit bit-wise unrolled multiplier was used for faster operation. The result will be a 48-bit number, out of which the upper 32 bits are selected as the result. Let this number be denoted as MULT_{upper} and its lower 24-bit portion be denoted as MULT_{lower}.
3. Set GUARD_BIT to the MSB(MULT\textsubscript{lower}).

4. Set ROUND_BIT to the next MSB of MULT\textsubscript{lower}.

5. Set STICKY_BIT to the OR of the rest of the bits of MULT\textsubscript{lower}.

6. If MSB(MULT\textsubscript{upper}) = 0, shift MULT1 left by 1 bit, shifting in GUARD_BIT.

7. If MSB(MULT\textsubscript{upper}) = 1, set STICKY_BIT = ROUND_BIT OR STICKY_BIT and ROUND_BIT = GUARD_BIT.

8. Adjust the exponent. $E_{\text{result}} = E_1 + E_2 - \text{bias}$, where bias = 127. Also, add 1 to the exponent if MSB(MULT\textsubscript{upper}) = 1.

9. Set SIGN\textsubscript{result} = S\textsubscript{1} XOR S\textsubscript{2}. Pack the result to get the 32-bit product.

The main bottle-neck in the floating-point multiply operation is the 24-bit integer multiplier. Floating-point divide operation is similar to multiply. The main difference is that the exponents now need to be subtracted in order to obtain the correct exponent. Also, a 26-bit divider needs to be used in order to generate additional bits for GUARD_BIT and ROUND_BIT values. The block diagrams of the floating point multiply and divide operations look very similar. The block diagram of the multiply unit is illustrated in Figure A.2. The only difference for the divide unit is that the 26-bit divide operation is performed by a “Divide Significands” block. The “Adjust Exponents” block differs in the sense that exponents are subtracted to get the exponent of the result.
Figure A.2 Flow diagram of the Floating Point Multiply / Divide Units
APPENDIX B

DESIGN HIERARCHY OF THE VHDL REPRESENTATION

The entire custom integrated circuit was represented using VHDL. Hence, a succinct and modular representation and eventual testing of the VHDL code was a crucial part of the design process. We found that modularity of the design minimized problems and made debugging a lot easier.

Figure B.1 illustrates the design hierarchy of the VHDL representation. Care was taken to include only those constructs from the VHDL language which can be easily and correctly synthesized by the synthesis tool. A bad VHDL representation often leads to a bad and sometimes oversize netlist. These problems were avoided by keeping the representation simple and re-using design units wherever possible.

The code was developed on a Sun Ultra Sparc-10 machine running Solaris. It was compiled and simulated using the Cadence NC-VHDL compiler and NC-SIM simulator from the Cadence Tool Suite.

Testing formed a very important part of the design process. An assembler and test-bench generator were implemented using LEX and C. This program takes the PE array commands as inputs and generates a VHDL testbench with the relevant test vectors. Test vectors for the floating point unit were generated randomly. Some of the important benchmarks run on the system were Conway’s Game of Life and the Jacobi Iterations. Details about these benchmarks can be obtained from [48].

The hierarchy diagram provides the functionality of each VHDL unit in short. The diagram highlights the modular design strategy used.
pe_array (top-level PE array)

- getCoords.vhd (provides hard-coded co-ordinates of individual PEs)

- getground.vhd (connects the boundary PEs to ground)

- pe.vhd (top level file for a single PE)

  - getOver.vhd  (performs bit-wise OR of the WOvrWrdIn/CommIn and NOvrWrdIn/CommIn)
  - ovwrwrdreg2.vhd (56-bit register to store the OvrWrd)
  - getPEHit2.vhd (compare input co-ordinates with own co-ordinates to determine PEHit)
  - getPEsignals2.vhd (calculate PCLd, PCCnt and PCInSel control signals)
  - programctr2.vhd (top-level file for the PC implementation)
    - add_6.vhd  (implments auto-increment facility of the PC)
    - mux2_7.vhd
    - prgrct2.vhd
  - getCMsignals.vhd (calculates the control signals for CM access)
  - mem40 (Artisan 128x40 memory...code generated by the memory compiler)
  - getWrBack.vhd (calculates the WriteBack signal which control the buffers on memory output lines)
  - OMsignals.vhd (calculates the control signals for OM access)
  - mux5_7.vhd (a 7-bit 5:1 multiplexer)
  - mem32 (Artisan 128x32 memory...code generated by the memory compiler)
  - fu.vhd (top-level file for the functional unit)
    - getselects.vhd (generates the functional unit control signals)
    - mux2_2.vhd
    - mux2_32.vhd
    - mux4_32.vhd
    - mux5_32.vhd
    - buf_31.vhd
  - fpaddsub32.vhd (top-level file for floating point add/subtract operation)
    - isaddop32.vhd (determines the type of operation - +/-)
    - unpack_addsub.vhd (unpacks the input operands)
    - getexpdiff.vhd (calculates the difference between input exponents)
    - getmant.vhd (checks for special values (0’s, infinity) and calculates 2’s complement of the second operand if required)
    - shift24.vhd (24-bit barrel shifter, also calculates guard, round, sticky bits)
    - add_mant24.vhd (24-bit full adder for bit-wise addition of mantissas)
    - sum2compl.vhd (calculates 2’s complement of the sum)
    - find24.vhd (logic for locating leading 1)
    - shft24.vhd (determines the amount by which sum is to be shifted to the left)
      - calcshft16.vhd
      - calcshft8.vhd
      - orgate2.vhd
    - leftshft24.vhd (24-bit barrel shifter)
    - getresult.vhd (logic to select the correct result)
    - rndresult.vhd (implements round-to-nearest policy)
    - checklimit.vhd (checks for exceptions and special values)
Figure B.1 Design Hierarchy Tree for the VHDL Representation