A Lisp-Oriented Multiprocessor Architecture
for Digital Parity Simulation

by
Anthony James Courtemanche

BSEE, Massachusetts Institute of Technology
(1986)

Submitted to the Department of Electrical Engineering and
Computer Science in Partial Fulfillment of the Requirements
for the Degree of

Master of Science in Electrical Engineering

at the

Massachusetts Institute of Technology
May 21, 1987

© Massachusetts Institute of Technology 1987

Signature of Author

Department of Electrical Engineering and Computer Science
May 21, 1987

Certified by

Richard D. Thornton
Thesis Supervisor

Accepted by

Arthur C. Smith
Chairman, Department Committee
A Lisp-Oriented Multiprocessor Architecture for Digital Parity Simulation

by

Anthony James Courtemanche

Submitted to the
Department of Electrical Engineering and Computer Science
on May 21, 1987 in partial fulfillment of the requirements for the Degree of Master of Science in Electrical Engineering

Abstract

Successful power-electronic circuit simulations have been achieved with the Par Simulator, an analog computer developed at M.I.T. in 1977 under the guidance of Professor John Kassakian. Advances in digital VLSI processor and arithmetic components and the lack of certain programming flexibility and system reproducibility in the Parity Simulator have motivated the design of a Digital Parity Simulator. The Digital Parity Simulator is a high-performance multiprocessor architecture optimized for high throughput in global data communication and low latency for local memory accesses. In the same vein of the original Parity Simulator, it is programmed to exploit the parallelism in circuit simulation by attempting to keep virtual parity with the circuit being simulated. Architectural enhancements to support Lisp-like linked lists allow flexibility in data representations without dramatic losses in processing performance.

This thesis describes various circuit simulation systems. The Digital Parity Simulator is described at the architectural and logic levels and the motivations for its design are explained. Performance is predicted and analyzed through the tracing of an example microcode routine on the Digital Parity Simulator.

Thesis Supervisor: Richard D. Thornton
Title: Professor of Electrical Engineering

2
Acknowledgments

There are many people who deserve credit for the success of this project. The following deserve special thanks.

Professor Richard D. Thornton of the MIT Laboratory for Electromagnetic and Electronic Systems conceived and supervised this work. He provided valuable aid in many forms at critical moments during this project.

Thomas Masotto has done some excellent work on multiprocessing software for handling the sparsity and non-linearity of power and magnetic circuit simulations. He relieved me of having to become an expert circuit simulator and allowed me to concentrate on the Digital Parity Simulator architecture.

Ka-yeung Cheung worked on parts of this project during the last year. He assisted in printed circuit board layout and was responsible for some of the support hardware for the Digital Parity Simulator.

Peter Caporal provided some useful data on a prototype communication network at the start of this project.

Gino Maa in the MIT Computation Structures Group generously allowed me to use the Circuit Board Design System™ for the DPS printed circuit boards.

Texas Instruments and Advanced Micro Devices made generous and valuable component donations. Special thanks go to Gerald Luecke at Texas Instruments for his interest in this project.

As in all things, my wife Patt provided personal support.

---

1Circuit Board Design System (CBDS) is a trademark of Bell-Northern Research Ltd.
Dedication

O LORD, our Lord, how majestic is your name in all the earth!

You have set your glory above the heavens.
From the lips of children and infants you have ordained praise
because of your enemies, to silence the foe and the avenger.

When I consider your heavens, the works of your fingers,
the moon and the stars, which you have set in place,
what is man that you are mindful of him,
the son of man that you care for him?

You made him a little lower than the heavenly beings
and crowned him with glory and honor.

You made him ruler over the works of your hands;
you put everything under his feet:
all flocks and herds, and the beasts of the field,
the birds of the air, and the fish of the sea,
all that swim the paths of the seas.

O LORD, our Lord, how majestic is your name in all the earth!

Psalm 8
# Table of Contents

## Chapter One: Circuit Design and Simulation

1.1 Introduction \hfill 8
1.2 Methods of Circuit Design \hfill 8
   1.2.1 Breadboarding \hfill 9
   1.2.2 Numerical Simulation \hfill 10
   1.2.3 Parity Simulation \hfill 11
1.3 Circuit Simulation Machines \hfill 12
   1.3.1 Standard Sequential Computers \hfill 12
   1.3.2 Parallel Processors \hfill 19
   1.3.3 The MIT Parity Simulator \hfill 29
1.4 The Need for a Digital Parity Simulator \hfill 32
   1.4.1 The MIT/EC \hfill 33
   1.4.2 The Need for a Multiprocessor with Floating-Point Hardware \hfill 40
1.5 Circuit Simulation Conclusions and Summary \hfill 51

## Chapter Two: The Digital Parity Simulator Architecture

2.1 System Overview \hfill 53
2.2 Micromachine \hfill 56
   2.2.1 Address Unit \hfill 57
   2.2.2 Condition-Code Unit \hfill 59
   2.2.3 Instruction Latch \hfill 60
   2.2.4 Microcode Memory \hfill 60
   2.2.5 Instruction Register \hfill 61
2.3 ALU/Data-Path \hfill 63
   2.3.1 Registered Integer ALU \hfill 64
   2.3.2 Floating Point Unit \hfill 65
   2.3.3 ALU Interconnection \hfill 66
2.4 Data Memory System \hfill 67
2.5 Multiprocessor Communication Interface \hfill 70
2.6 Host Interface \hfill 76
   2.6.1 Intel 310 Host Processor \hfill 81
   2.6.2 Expansion Processors \hfill 82
2.7 Critical Path Analysis \hfill 83
   2.7.1 Micromachine Critical Path \hfill 85
   2.7.2 Data Critical Path \hfill 86
2.7.3 Critical Path Conclusions 87
2.8 Microcode Examples 88
2.8.1 Vector Scale-and-Add with Storage Vectors 88
2.8.2 Vector Scale-and-Add with Linked-Lists 92
2.8.3 Microcode Conclusions 96
2.9 DPS Architecture Summary 96

Chapter Three: Digital Parity Simulator Results 98

3.1 Digital Parity Simulator Construction 98
3.1.1 CBDS 99
3.2 System Results 101
3.3 Conclusions 101
3.3.1 DPS System Conclusions 101
3.3.2 Directions for Further Work 102
3.3.3 Project Conclusions 103

Appendix A: DPS Instruction Set 108

Appendix B: Host Processor Commands 116

6
# Table of Figures

**Figure 1-1:** von Neumann Computer Architecture  
**Figure 1-2:** von Neumann Computer Architecture With Data Cache  
**Figure 1-3:** von Neumann Computer Architecture With Instruction Cache And Pre-Fetch Unit  
**Figure 1-4:** Harvard Architecture  
**Figure 1-5:** Multiprocessor System Architecture  
**Figure 1-6:** Butterfly System Architecture  
**Figure 1-7:** Parity Simulator Architecture  
**Figure 1-8:** Dataflow graph of $\bar{c} = \bar{a} + k \cdot \bar{b}$  
**Figure 1-9:** Proposed MIT/EC  
**Figure 1-10:** MIT/eC  
**Figure 2-1:** Digital Parity Simulator System Architecture  
**Figure 2-2:** DPS Processor Architecture  
**Figure 2-3:** DPS Micromachine  
**Figure 2-4:** Integer ALU and Floating-Point Processor Interconnection  
**Figure 2-5:** Memory System  
**Figure 2-6:** Multiprocessor Communication System  
**Figure 2-7:** Host Interface System  
**Figure 2-8:** Control and Dataflow in the Digital Parity Simulator  
**Figure 2-9:** Representation for Storage Vectors  
**Figure 2-10:** Representation of List Based Vectors  
**Figure 1:** Microinstruction Word
Chapter One

Circuit Design and Simulation

1.1 Introduction

This thesis will explore the issues relating to computer architectures used for the simulation of analog electronic circuits. Although the emphasis is on the peculiarities of analog electronic circuit simulation, many of the ideas presented here are applicable in all areas of simulation. Chapter One will go through circuit simulation and the various architectures that have been used to perform circuit simulation. It ends with the steps leading to the derivation of a new architecture, a digital Lisp-oriented parity simulator, which attempts to combine what are felt to be the best features of previous circuit simulation architectures. Chapter Two describes in detail the architecture of this Digital Parity Simulator and shows by example how the architectural enhancements are well suited for numerical circuit simulation. Chapter Three analyzes the Digital Parity Simulator in terms of its physical construction and draws conclusions about the Digital Parity Simulator System.

1.2 Methods of Circuit Design

The synthesis, design, and analysis of complex non-linear electronic circuits are usually done by one of several methods:

1. bread-boarding actual circuits,

2. numerical simulation on digital computers, and

3. parity simulation with analog components.
The beginning of this chapter will go over these methods in some detail. This will lead to an overview of the various computational tools required by these methods. The overview of existing work in these methods of circuit simulation will provide motivation for a digital parity simulator. A previous attempt at a digital parity simulator will be discussed and analyzed, and a new digital computer architecture for parity simulation will be proposed. The architecture will be discussed at a system level, emphasizing what it will have to provide in order to perform effective parity circuit simulation.

1.2.1 Breadboarding

Breadboarding is the most basic of circuit design methods. A circuit is proposed, and a prototype is built out of real components. The functionality and performance of the circuit is measured and analyzed, and changes are made to the circuit until it performs as desired.

This may be the most reliable of all design tools. By using the actual components that will eventually be incorporated into the final product, the designer has complete assurance that the eventual circuit will perform as designed. If the prototype works, then the design works. Nevertheless, there are several drawbacks to this method. First, since this method requires actual components, the components used during the design cycle are limited to those that are actually available. This fact may alter the way a design evolves, and alternative solutions to the problems in a circuit may never be considered just because a breadboarding component is unavailable.

The second drawback to this method is that the construction of the breadboarded circuit is often time consuming. Time is taken not only in constructing the initial circuit, but also at every change of the circuit.

In summary, breadboarding is very reliable once the final design is derived, but
the design steps are sometimes awkward and are often time consuming.

1.2.2 Numerical Simulation

For many cases, numerical circuit simulation on digital computers is very successful. Most, if not all, numerical circuit simulation reduces to performing matrix and vector operations on matrices that describe the static and dynamic properties of the circuit being simulated. One of the most common operations is the use of Gaussian elimination (or some equivalent) on a matrix in order to solve a set of linear equations.

One of the major optimizations that have been discovered in this area is that the matrices used to describe circuits often display tremendous \textit{sparsity}. That is, many of the elements in the matrices are zero, reflecting the fact that most components in a circuit are electrically connected to only a handful of other components. Sparsity can be used in order to minimize the amount of numerical operations performed on a matrix. For instance, when scaling a row of a matrix in order to do Gaussian elimination, only the non-zero elements need to be scaled. If scaling a number via multiplication is a time consuming operation, as it often is if the scaling operation is done with numbers in floating-point format, then reducing the number of redundant or unnecessary scalings can lead to a significant improvement in processing speed.

Despite the many optimizations discovered for numerical circuit simulation, complexities of scale, lack of processing power, and problems inherent in numerical methods have often made many problems unwieldy. The most major of these problems has been the lack of processing power. Traditional sequential computers cannot perform the numerical calculations fast enough to come close to real-time simulation. The so-called supercomputers such as the Cray-1 provide tremendous processing power but these are prohibitively expensive and
are normally unsuitable for circuit simulation. The current frontier in computer design is the use of *multiprocessing*, that is, using many processing elements in concert to solve different parts of one problem, and this holds much promise for providing the necessary processing power for numerical simulation, as shall be seen later on in this thesis.

In summary, numerical simulation has provided success, but it is clear that more processing power is needed before simulation can approach real-time speeds.

### 1.2.3 Parity Simulation

A third method of analog electronic circuit simulation is through parity simulation. The most significant example of this type of simulation is the Parity Simulator [14, 13], developed at M.I.T. in 1977 under the guidance of Professor John Kassakian. The Parity Simulator is an analog computer specialized for power-electronic simulation. Each device in the system being simulated is represented by an analog or hybrid subsystem called a component board. These component boards are electrically connected in a network of identical topology to the system being simulated. Currents or voltages in each subsystem can be scaled in order to simulate various values of components. Because analog components are used, the solving of the associated differential equations describing the circuit are performed very quickly; The simulation can sometimes proceed in real time. State information at each node in the system can be sampled and displayed and plenty of data is available to the designer as he or she optimizes the desired circuit. Accuracy and precision in the Parity Simulator is limited by the accuracy and precision of the devices used to scale system voltages and currents.

A drawback of parity simulation is that the analog machine which performs it is time-consuming to maintain, difficult to reproduce, and lacks certain flexibility.
in the components it can simulate. New subsystems are sometimes difficult to produce and may require a significant amount of development time. Also, simulations of magnetic circuits are difficult to perform since the component boards that would be used in modeling magnetic circuits are difficult to produce.

In summary, parity simulation on the Parity Simulator is efficient and accurate for simulations which contain components that already exist as subsystems in the Parity Simulator. Nevertheless, in some cases the system is limited by what subsystems are available and by the reduced availability of a difficult to maintain and reproduce machine.

1.3 Circuit Simulation Machines

This section overviews the machines on which the some of the simulations in the previous section are run. The different types of machines that are looked at in detail are the general purpose sequential computer, several types of parallel processors, and the MIT Parity Simulator. The various strengths and weaknesses of these machines will be discussed in the context of circuit simulations which run on them.

1.3.1 Standard Sequential Computers

The most common and most general computers are the standard sequential von Neumann computers. These machines consist of two units, the processing unit and the memory unit, as represented in Figure 1-1. In this architecture, the processing unit first issues memory requests for instructions to perform. As the processing unit executes an instruction, it may issue other memory requests in order to receive data to operate on or to store results into memory. When an instruction is finished, the next instruction is requested from the memory unit. The processing unit typically contains a program counter, which contains the
address of the current instruction being executed. This program counter increments in order to fetch the next instruction after the current one is executed. Exceptions to this are when branch and jump instructions actually manipulate the contents of the program counter in order to change the flow of control in the processing unit.

![Diagram of von Neumann Computer Architecture](image)

**Figure 1-1:** von Neumann Computer Architecture

This standard architecture suffers from the so-called von Neumann bottleneck. Performance is limited by the speed of the memory unit and its interface to the processing unit. The processing unit cannot compute faster than the speed at which instructions and data can be retrieved from memory. Many optimizations have been used to alleviate this bottleneck, however.

One optimization used to reduce the von Neumann bottleneck is to use caches in the processing unit to store frequently referenced memory locations, as shown in Figure 1-2. If a memory location is frequently referenced, the data in that location is stored in high-speed local memory in the cache. When the processing unit issues a request for that memory location, the cache can respond directly to
the request and the time-consuming memory reference is avoided. Various write strategies are used to make sure the local cache and the memory unit are consistent when cached data is overwritten by the processing unit. Data caches exploit temporal locality of reference by caching recently referenced memory locations. Spatial locality of reference can also be exploited by using data caches which cache lines of data instead of single data items. If a memory location is referenced, adjacent locations are also cached in anticipation that they too will be referenced.

Another optimization to improve performance is to use caches to store instructions and to have an instruction pre-fetch unit as shown in Figure 1-3. Computer instructions in most programs exhibit spatial locality of reference and caches that take advantage of this fact often provide a significant boost to processing performance. Instruction pre-fetch units can have varying degrees of complexity, but all exploit the fact that if instruction i is being executed, instruction i+1 will most likely be executed in the near future. Hence, instruction pre-fetch units typically cache in advance instructions adjacent to the
Figure 1-3: von Neumann Computer Architecture With Instruction Cache And Pre-Fetch Unit

one currently being executed by the processing unit. Depending on the size of the instruction caches, small loops of code containing only a few instructions can execute very quickly if the entire program loop can fit in the instruction cache. Jump and branch instructions do foil the job of the instruction pre-fetch unit however, but more complicated units can often make good guesses as to what the next instruction to be executed will be and they cache these instructions accordingly.

One reason why instruction caches and pre-fetch units can improve performance in standard computers is because these units can operate when the processing unit is operating and the path from the memory unit to the processing unit is unused. When the path to memory is not being used, the pre-fetch unit can use this data path to retrieve likely to be executed instructions. These pre-fetch units allow local parallelism to be exploited. While the processing unit is
executing instruction $i$, the pre-fetch can retrieve instruction $i+1$. Another way of exploiting this instruction-reference/processing parallelism is to use a so-called Harvard architecture, as depicted in Figure 1-4.

**Figure 1-4:** Harvard Architecture

In a Harvard architecture, there are separate instruction and data memories. References to instruction memory for instructions can occur in parallel to references to data memory for data. An instruction pre-fetch unit can run unopposed by requests to main memory. By having separate instruction and data memories, opportunities arise to optimize each for their different applications.

In all these different variations of the von Neumann standard computer architecture, several characteristics of the system are important in analyzing the performance of the system. First of these is system word size. Wider word sized computers require fewer memory references than smaller word sized machines since more data can be retrieved during each reference. Therefore, wider word sized computers usually have a much better performance. Of course, arbitrary
performance improvements cannot be obtained by just increasing the word size. Basic numeric operations such as addition require that data be propagated from lower bits in a word to higher bits in a word. Thus, these operations take longer for wider word sizes. Obviously, there is an engineering tradeoff between small word sizes for fast numeric operations and wide word sizes for fast data retrieval.

Another characteristic that affects processing performance is the amount of hardware optimizations for the software that is to be run on these machines. These optimizations move computers from being general purpose to being more specialized. A common hardware optimization is the use of floating-point co-processors or accelerators to improve the performance of numeric floating-point operations in computers intended for scientific, engineering and business applications. Less common but equally valid are hardware optimizations such as tagged memory and efficient pointer indirection hardware in Lisp-language based machines used for Artificial Intelligence applications.

When one examines the circuit simulation software written for general purpose machines one sees that they generally attempt to exploit the little hardware optimizations available in these machines. These software packages are usually written in a language, such as Fortran or C, that can easily be compiled to generate native machine instructions that are very efficient. Important simulation primitives may be hand coded in assembly language to guarantee the fastest and most efficient simulation. Nevertheless, performance is often unsatisfactory, and the performance to cost ratio is often poor. This can be explained by several reasons.

One reason why circuit simulation performance on standard sequential computers is often unsatisfactory is due to the computational natures of the circuit simulation. In circuit simulation the state of a circuit evolves as simulated time advances. Each time increment computed can be viewed as a single iteration in a
loop describing the circuit for all time. If the circuit to be simulated must be analyzed for a long period of time through the circuit’s state evolution, then many of these iterations are necessary. In power-electronic circuits, many circuits are characterized by stiff differential equations in which time constants have varying orders of magnitude. In these cases, small time increments are needed in order to accurately simulate the characteristics of the circuit that are changing rapidly due to a small time constant. Yet, many of these small time increments are needed to move the circuit into the behavior that changes slowly over time which is associated with a larger time constant. Therefore, accurate simulations end up requiring a great number of iterations.

It is often the case that circuits are characterized by non-linear components. In order to solve the set of simultaneous non-linear equations, the state for each simulated time increment is computed using iterative relaxation techniques. Therefore, each iteration in moving from one time increment to the next requires several internal iterations in order to solve the differential equations. Thus, the simulation is often a doubly iterative process that requires a great number of program steps.

Another reason why circuit simulations run on standard sequential computers have unsatisfactory performance is that many general purpose computers are designed with many features to aid in programmability at the expense of processing speed. One example of this type of feature is the use of virtual memory to allow the development of large software systems which utilize enormous address spaces. Virtual memory frees the programmer from worrying about issues such as memory segmentation but it incurs a large monetary price in support hardware. Performance is also reduced when virtual memory systems are used to support multi-tasking. If several processes are run which take advantage of the large address space, each process will attempt to retrieve data from virtual addresses by loading pages of data from the disk to main memory.
If all these pages do not fit into main memory simultaneously, as in the case when many users are running different programs on the system, then the computer is subject to disk thrashing. During disk thrashing, the computer spends much of its time swapping virtual address pages to and from the disk. Little processing is done when this occurs, and performance is poor. On the other hand, general purpose computers with virtual memory and hardware caches are ideal for systems where many users are simultaneously running similar programs that can share virtual address pages. A good example of this is a system where many users are running the same text editing program. The code for this program can be shared in main memory for all the users, and the rest of the virtual address space that can fit into main memory can be used to store the temporary copies of the text that is being processed.

From these observations, it seems that although the general purpose standard sequential computers can run software to simulate circuits, the match between the architectures and the problems being solved is much less than perfect. The generality in a general purpose computer is ideal for writing programs, but the penalty on performance is too high.

1.3.2 Parallel Processors

Since most general purpose computers are too slow for many circuit simulations, attempts have been tried on parallel processors. As the name implies, a parallel processor is a computer system which can do several things in parallel. In one sense, most general purpose computers are parallel to some degree. For example, the VAX 8600 [10, 20] is an instruction pipelined machine. This means that while part of the $i^{th}$ instruction in an instruction stream is being processed, a different part of the $i^{th}+1$ is processed as well. For instance, while one instruction is using the ALU to perform a calculation, the next instruction can use this time to fetch its operands from main memory. This type of instruction
parallelism is quite limited, as only the non-data-dependent instructions in a local area have any chance of executing in parallel\textsuperscript{2}.

When one uses the term parallel processor, however, one is usually referring to a system where the parallelism can be expressed or exploited in more global and distributed senses. For instance, vector processors are parallel processors where the user can express, or the compiler can deduce, that the same operations performed on many data items can be performed in parallel; Here, the data can be considered, and represented, as a vector datum, and the many operations can be considered, and performed, as vector operations. Other parallel processors include the array processors, where data is processed by a structured array of various processing components whose topology roughly matches the functional topology of the operation the array is performing. A more general type of parallel processor is the multiprocessor, in which many processors are programmed to operate on different sets of data. There are two categories of multiprocessors, the first being the Single Instruction Multiple Data (SIMD) type, in which all the processors perform the same operations on different data. The other category is the Multiple Instruction Multiple Data (MIMD) type, in which each processor can be programmed to perform a different operation on the data. All these processors are still considered to be von Neumann systems since they still issue sequential memory requests to memory in order to receive new instructions.

All these different types of parallel processors attempt to speed up computation by allowing different operations to occur in parallel. It is important to note that these processors are only valuable when the computations they are performing

\textsuperscript{2}In fact, besides being limited, this type of instruction pipelining is very expensive in terms of the hardware implementation. A great deal of hardware is necessary to generate and respond to the various resource interlocks that are necessary to make an instruction pipelined system work correctly.
exhibit some sort of intrinsic parallelism. If the computation is intrinsically sequential, as in the case of file I/O, for example, then adding extra processors does not increase processing speed. There just isn’t anything for the extra processors to do, and they end up waiting idly. But, if there is parallelism in the computations being performed, as there is in the case of circuit simulation where most interactions occur locally between adjoining components, then the parallel processors can provide an increase in processing speed. This only happens however if the parallelism can be expressed in the software or deduced by the compiler and if the architecture can efficiently handle the parallelism.

Some of these parallel architectures will now be examined to see how effectively circuit simulations can run on them.

Vector Processors (a.k.a. Super Computers)

Probably the most famous of the vector processors is the Cray family of super computers pioneered by Seymour Cray [15, 7]. The Cray-1 is a wide-word vector pipelined machine. The processor and memory data paths are 64 bits wide. The memory is composed of 16 interleaved banks; References to successive memory locations are actually directed to separate memory banks, thus allowing the different memory banks to process requests in an overlapped fashion. The Cray-1 processor contains several functional units that can operate in parallel and contains an incredible amount of local storage. There are 8 24-bit address registers, 64 24-bit intermediate address registers, 8 64-bit scalar registers, 64 64-bit intermediate scalar registers and 8 64-element vector registers with 64 bits per element. It is this last set of registers, the 8 vector registers, that supplies a tremendous processing boost to the Cray-1.

Consider two arbitrary length vectors \( \vec{a} \) and \( \vec{b} \) that are in vector storage. Suppose a scale-and-add operation\(^3\), \( \vec{c} = \vec{a} + k \cdot \vec{b} \), is to be performed. By using the

\[^3\text{This operation is common is many direct methods of performing Gaussian elimination.}\]
vector registers and a process called chaining, this operation can run on the Cray-1 in approximately the same number of machine cycles as the lengths of the vectors. This is possible because there are separate functional units for multiplication and addition, and because results of one functional unit can be passed as inputs to another functional unit. Hence, to perform the calculation, the first element of $\mathbf{b}$ and the scalar $k$ are first feed into the multiplier unit. The result of this operation is immediately fed into the adder unit concurrently with the first element of $\mathbf{a}$. This chained addition causes no intermediate storage for the vector result of the multiplication. Also at this time, the second element of $\mathbf{b}$ and the scalar $k$ are fed into the multiplier. The result coming out of the adder unit can be stored into a vector register to collect the elements of $\mathbf{c}$. Once this pipeline is full, results are returned to the vector register once each clock cycle, and so the vector operation $\mathbf{c} = \mathbf{a} + k \cdot \mathbf{b}$ has an effective throughput of one element per cycle. With a machine cycle time of $12.5\, ns$, the Cray-1 can execute this code very quickly. Although there is latency incurred for filling up the pipeline, even more complicated operations are possible with this high throughput. With the 8 vector registers, vector operations can be run without referencing data in main memory. It is also important to note that these type of operations take very few programming instructions. The Cray-1 is equipped with vector instructions which perform the proper pipelining and chaining of operations. Thus, the operation $\mathbf{c} = \mathbf{a} + k \cdot \mathbf{b}$ can be executed by issuing just three instructions (assuming the lengths of the vectors have been set up in the internal Vector-Length register).

Because of the tremendous processing performance offered by the the Cray-1, it is in many senses ideal for numerical circuit simulation. Since numerical circuit simulation essentially reduces to performing matrix and vector operations and since the Cray-1 is set up to perform vector operations very quickly, there is a good match between the architecture and the problem being operated on. In
fact, even the programming environment is well matched to the problem. This is because CFT, the optimizing ANSI 66 Fortran compiler for the Cray, allows standard Fortran code to take advantage of the architectural enhancements in the Cray-1 computer. CFT is clever enough to generate vector instructions and to take advantage of chaining in fortran DO loops. Where a ordinary Fortran program would code the \( \vec{c} = \vec{a} + k \cdot \vec{b} \) operation as
\[
\text{DO 10 I = 1,N} \\
10 \ C(I) = A(I) + k*B(I)
\]
the CFT compiler would be able to deduce that this code is vectorizable since there are no iteration crossing data-dependencies. The appropriate vector instructions would be generated, and the original source code would not have to be changed (and potentially obscured) in order to take advantage of the vector processor.

From this discussion, it would seem that the Cray-1 is the perfect machine for circuit simulation. It has very high performance, and it can run standard software with this increased performance. Nevertheless, the case is not closed. For all its 138 Mflop performance, the Cray-1 is an incredibly expensive machine. The Cray-1 is so expensive in fact that there are very few of them, and competition for programming time on the available Cray-1’s is stiff. Hence, although the Cray-1 is well suited for circuit simulation, its availability is so limited due to economic reasons that it outside the realm of feasibility in many cases.

**Array Processors**

Array processors are difficult to describe since they are ill-defined. Depending on who is asked, an array processor can be a collection of adders and multipliers topologically arranged to efficiently perform Fast Fourier Transforms (FFTs), or a crossbar arrangement of functional units which allow all possible connections between the output of one functional unit to the input of another functional unit, or a SIMD hypercube array of one bit processors such as in the Connection
Machine\textsuperscript{TM} 4 [11]. The only thing somewhat universal about array processors is that they are all difficult to program. The applicability of the array processor architectures to circuit simulation varies widely with the different types of array processors. Fixed topology array processors are too specialized to be useful for simulating general classes of circuits. Crossbars of functional units are much more general, but these are very difficult to program. The Connection Machine is very general and perhaps more straightforward to program, but like the the Cray-1, it is a very expensive machine.

For circuit simulations the array processor class of parallel processors can provide some increased performance but at a great expense in terms of software complexity and price.

\textbf{Multiprocessors}

Another class of parallel machines is the general purpose multiprocessor. A general purpose multiprocessor can be characterized as a collection of processor and memory components that can communicate to each other through some sort of interconnection network. The details of this interconnection can vary considerably from system to system. One type of system organization is depicted in Figure 1-5. This is an idealization of a general-purpose shared-memory multiprocessor where each processor is associated with some portion of the global memory space. Memory accesses from a processor to the portion of the global memory that is associated with that processor can occur very fast. Global references to memory on another processor node must go through the interconnection network and subsequently take longer. This idealized model applies to several multiprocessors such as the MIT Concert computer [2] and the BBN Butterfly\textsuperscript{TM} 5 Multiprocessor [8, 6].

\footnote{Connection Machine is a trademark of Thinking Machines, Incorporated.}

\footnote{Butterfly is a trademark of Bolt Berenak and Newman, Incorporated.}
Figure 1-5: Multiprocessor System Architecture

An example of circuit simulations that have been run on a general purpose shared-memory multiprocessor are the experiments of Deutsch and Newton of the University of California at Berkeley [9]. They ran their simulations on the BBN Butterfly computer and used a method called Iterated Timing Analysis (ITA). The ITA method uses the Newton successive-over-relaxation algorithm to solve the circuit differential equations and event-driven analysis to exploit the temporal-sparsity of the electrical network\(^6\). Being event-driven, this method naturally lends itself to efficient implementation on a data-driven computer. Several characteristics of the Butterfly multiprocessor allow it to perform like a

\(^6\)The event-driven analysis takes advantage of the fact that circuit state variables can be modeled as processes. Changes in state variables cause processes to "wake up" to respond to these changes. The processes are not run when the state of the circuit is not changing. Thus, processor utilization is not wasted simulating the circuit when the circuit state is not changing.
data driven-computer, as detailed below.

![Diagram of Butterfly System Architecture](image)

**Figure 1-6:** Butterfly System Architecture

The Butterfly is a shared-memory multiprocessor with processor nodes containing a Motorola 68000 microprocessor, up to 1 megabyte of memory and an Amd2901 based processor node controller. Each node is connected via an omega network as shown in Figure 1-6. The omega network is constructed with 4-by-4 "perfect-shuffle" switching elements. These switches allow the messages on any of 4

---

7 A 2-by-2 "perfect-shuffle" switch is sometimes called a butterfly switch. It is from this that the Butterfly multiprocessor gets its name.
inputs to be routed to any of the four outputs. For an \( N \) processor system, \( \log_4 N \) switches are needed per processor. Thus, in an \( N \) processor system, the complexity of the switching hardware goes up as \( N \cdot \log_4 N \), while the length of time to send a message between two processors goes up as \( \log_4 N \).

The 2901 based processor node controller plays an crucial part in the multiprocessor communications in the Butterfly multiprocessor. When the 68000 attempts to read or write an address which is on a remote processor node, the 2901 controllers on each node work in concert to perform the communication of the data across the switch. The 2901 does not need to establish a connection over the switch for references to memory local to the processor node, so local accesses can occur much faster than global accesses. On the Butterfly, the ratio of global access time to local access time varies with the number of processors in the system, but it is approximately 8 to 1.

The Chrysalis\textsuperscript{TM} \textsuperscript{8} operating system [16] which runs on the Butterfly contains several primitives that facilitate multiprocessor programming. Dual queues, which are queues that can contain pointers to computational tasks to be performed or pointers to available processors which can perform those tasks, allow efficient multitasking communication without explicit semaphore locking and testing. At any instant, a Butterfly processor can be either a producer or consumer of tasks. Producers add tasks to the dual queue, while consumers remove and operate on tasks from the dual queue. If the dual queue is empty when a consumer processor needs a task to work on, the processor "sleeps" and is put on the dual queue. Sleeping processors consume no memory bandwidth while sleeping. When an attempt is eventually made to add a task to the dual queue, a sleeping processor is awakened and it immediately receives that task. If all processors end up on the dual queue, then there is no more work to be done.

\textsuperscript{8}Chrysalis is a trademark of Bolt Berenak and Newman, Incorporated.
Thus, processors usually check to see if they are the last one to be put on the dual queue. If so, then the program is complete and that processor can run some termination routine.

Reports made by Deutsch and Newton indicate that the Butterfly multiprocessor can very efficiently perform the ITA method on test circuits of MOS transistors. With 10 processors, they report a 70% processor efficiency, which is fairly close to the 81% theoretical limit they compute for 10 processors.

One item that is of particular interest in their report concerns circuit partitioning. There are three basic methods of partitioning a circuit for multiprocessor simulation. They are:

- random allocation, where circuit nodes are allocated to processors at random, with the only constraint being that an attempt is made to have the same number of nodes per processor,

- trace allocation, where nodes that are connected serially are kept on one processor in order to minimize the amount of remote memory references, and

- minimum distance allocation, where nodes are distributed in such a way to take advantage of the interconnection network. Adjacent nodes in a circuit are arranged on processors that are adjacent in the interconnection network.

Deutsch and Newton report that there is less than 10% difference in processing speed for these particular node allocation methods. Hence, the easiest strategy, random allocation, was chosen for their experiments. However, they report that partitioning circuits into strongly connected sub-circuits and distributing these subcircuits among the processors can improve efficiency.

The results from this analysis are encouraging, as they show that circuit simulation on a general purpose multiprocessor can be effectively performed. However, several questions should be asked about the match between the
problem to be solved, that is, the circuit simulation, and the architecture used to solve that problem, namely the Butterfly. Deutsch and Newton report that local memory references on the Butterfly are completed in 625\(ns\), while references through the network take about 4 or 5\(\mu s\). By modern standards, the local memory transfer time is somewhat slow, and the global transfer time through the switch is terrible. Block mode transfers provide some palliation by providing 500\(ns\) transfers between processor nodes, but all these timings must be taken in the context that they are for 16 bit transfers. The 68000's communicate to memory via 16 bit memory ports, and the bus width in the interconnection network is 16 bits as well. In circuit simulation, it is presumed that most quantities will be represented in floating-point format. Thirty-two bits are a minimum to provide single precision calculations, and for more accurate simulations 64-bit double precision floating-point numbers would be necessary. With this in mind, the above transfer times should be multiplied by at least a factor of 2, if not a factor of 4. This indicates a severe performance penalty. It appears that by using a general purpose machine such as the Butterfly, at the outset performance is cut in half in comparison to a machine that is specialized for numerical calculations by having longer word sizes.

1.3.3 The MIT Parity Simulator

A radically different computer architecture that is used for analog circuit simulation is the Parity Simulator. The parity simulator is a synthetic breadboard which uses digitally controlled analog elements as shown in Figure 1-7. These analog elements, called “component boards”, are electrically floating with respect to all other parts, and can be connected in arbitrary networks via an electro-mechanical crossbar that is set up by a digitally controlled robot. Each component board is synthesized to have the same terminal relationships as the device it is intended to simulate. A resistor board is synthesized as a hybrid system with one terminal pair that has the property that:

29
\[ i = \frac{v}{R} \]

An inductor board is likewise synthesized as a hybrid system with one terminal pair that has the property that:

\[ i = \frac{1}{L} \int v \, dt \]

The implementation of these boards can vary widely as long as the terminal properties hold. The component boards for energy storing devices such as inductors and capacitors will contain some sort of electronic integrators if not actual inductors and capacitors themselves.

\[ \text{Front End Computer with Keyboard, CRT, Lightpen, and Disk} \]

\[ \text{Breadboard Connection Matrix with Patching Robot} \]

\[ \text{Component Boards} \]

\[ \text{Data Acquisition and Digital Control of Analog Component Boards} \]

\[ \text{Peripherals (Printer and Penplotter)} \]

**Figure 1-7:** Parity Simulator Architecture

The value of \( R \) and \( L \) for particular component boards of the above types can be programmed by a digital computer that serves as the front-end interface to the Parity Simulator. Besides being able to program the various component boards, the front-end computer controls the robot that configures the crossbar and effectively sets up the topological parity with the circuit being simulated. The front-end also provides a friendly interface the Parity Simulator. Circuits are
entered schematically using the color CRT display, and the simulation schematic and crossbar interconnection can be saved to disk for later use. Also, the front-end processes inputs from A/D converters that can probe various nodes of the circuit being displayed and can plot the changes in node voltages and currents as functions of time.

Elements more complicated than simple resistors, capacitors and inductors can be represented as component boards. A non-linear inductor can be synthesized as a basic inductor board with a scaling factor $L$ that is either under control of the front-end or under control of a pre-programmed PROM on the component card itself. To use the PROM to control the inductance, an A/D converter samples the terminal pair input to the non-linear inductor card. The digital output of the A/D is used as an address into the PROM, and the output of the PROM determines the current value of $L$ of the non-linear inductor. By changing the values in the PROM, different saturation curves can be attributed to the particular non-linear inductor.

The Parity Simulator has been used very effectively to run circuit simulations. Since its architecture can change to match the circuit being simulated, it seems to be ideally matched to the problems that it is trying to solve. Accuracy is limited by the accuracy of the components in the system, but simulations can sometimes proceed in real time. Nevertheless, there are strong weaknesses associated with the Parity Simulator. The first of these is obvious: component boards are dedicated in their use. A resistor board can never be used to function as a inductor board, and vice versa. Even worse than this is the fact that many circuit elements are difficult to synthesize as component boards. Even the common capacitor and resistor boards must complicated in order to ensure local stability. Also, the boards must maintain electrical isolation except for the interconnected terminals so that bias and noise is not introduced into the simulation; This requires that all the boards be separately powered by individual
batteries on each board which further complicates and increases the cost of the system.

In summary, the Parity Simulator is an effective circuit simulation architecture with good performance, but it has the drawback of lacking flexibility in what systems it can simulate. Because not all possible component boards are available on the Parity Simulator, it is not a machine for truly general purpose circuit simulation.

1.4 The Need for a Digital Parity Simulator

There is a obvious gap between the two types of machines presented above. On the one hand, the various digital machines, while providing more or less hardware support for various operations, all have a tremendous amount of built in generality. Because all are programmed through software, they can handle a wide variety of simulation requirements. Yet, their topologies, and word sizes are less than ideal for analog circuit simulation. On the other hand, the Parity Simulator, since its processing topology can change to match the circuit being simulated, has an ideal way of handling the simulation of various circuits. Being an analog machine, it can solve the differential equations associated with the circuit being simulated very quickly. Yet, its processing components, namely the component boards, are not flexible in the types of components they can simulate.

When given two alternatives with disjoint strengths, the best choice is usually a synthesis of the two alternatives. It is therefore not surprising that attempts at combining digital processing power with parity simulation have been tried. The questions almost beg themselves: Isn't there some way of using the generality of digital processors in a network in parity with a circuit being simulated in order to achieve an effective but inexpensive simulation? Isn't it possible to build a digital parity simulator using several digital processors? In order to answer these
questions, a previous attempt at a digital parity simulator will be examined. After a discussion of its strengths and weaknesses, the motivation of a fresh attempt at a new architecture will be presented.

1.4.1 The MIT/EC

Thomas Sterling, in his master's thesis report in 1981 [17], describes the various issues regarding using multiprocessors to perform parity simulation. Of his many conclusions, one of the most important relates to the use of multiprocessors to perform parity simulation. Three reasons are given why topological parity is meaningless in relation to digital simulation of analog components. These reasons are

1. In a digital multiprocessor, signals can be characterized as either inputs or outputs. In an analog circuit, there is little distinction between input and output signals, and in either case, it is not obvious whether the signals of interest should be considered as voltages or as currents.

2. In a digital multiprocessor, signals are associated with data paths. In a circuit, specific signals cannot be associated with specific wires in most cases. For example, a voltage “signal” in an analog circuit does not flow through wires.

3. In a digital multiprocessor, the interconnection topology is a passive network whose sole job is to propagate information from one processor to another. In an analog circuit, the interconnection topology is an active one which besides directing the flow of information also establishes equilibrium constraints with respect to branch variables.

It is for these reasons that actual circuit parity was discarded as a means for digital circuit simulation. Instead, the realization was that circuit simulation on digital computers is equivalent to solving simultaneous sets of equations derived from Kirchoff's current and voltage laws.
In order to create a multiprocessor system which could solve simultaneous equations associated with a circuit and take the maximum advantage of the available parallelism, Sterling chose an architecture that is different from the three types of digital parallel architectures described in section 1.3.2. This architecture is the \textit{dataflow architecture} [4].

\begin{figure}
\centering
\includegraphics[width=\textwidth]{dataflow_graph.png}
\caption{Dataflow graph of $c = \bar{a} + k \cdot \bar{b}$}
\end{figure}

The dataflow computer is one of the few non von Neumann computers. It does not issue sequential memory requests in order to receive new instructions in the
standard von Neumann fashion. The dataflow computer does not have a program counter. Instead, each instruction is part of a dependency graph as shown in Figure 1-8. When instruction $i$ finishes executing, all the instructions that depended on it are then available to be run. If there is more than one processor, the different available instructions can all be executed in parallel. As these instructions execute, more instructions that depend on them are enabled, and so on. Complicated hardware schemes which utilize tokens to propagate the values associated with the operation of various instructions have been proposed and in a few select cases, actual general purpose dataflow machines have been built. Instructions produce tokens as values, and when an instruction has all the required input tokens, it is then able to run. Work on intelligent compilers which can take ordinary programs and generate the program graphs that run on dataflow machines is also showing some progress [19].

The dataflow computer that Sterling proposed consisted of tasker units which performed the actual interpretation of the dataflow program graph. Each tasker would be built out of a Z8001 16-bit microprocessor, 64K bytes of data memory, 8K bytes of instruction memory, 4K bytes of stack memory, interfaces to global data buses, and interfaces to global command buses. Up to 62 taskers would be connected with a host processor, a global multiport memory, and a dispatching unit via 6 system busses as shown in Figure 1-9.

The taskers contain the proper logic to interpret the data flow graph. They request new instructions from the dispatcher when they have indicated that they are done processing the current instruction by returning a value token. When the dispatcher receives a value token for a particular instruction, it knows that any instructions that are waiting on that value can then proceed. These instructions are therefore enabled and put into a queue containing instructions to execute. These instructions in the queue are distributed to the taskers as they request new instructions.
Figure 1-9: Proposed MIT/EC
An important part of the proposed MIT/EC is the multiport memory unit which is called the "tank". The tank was proposed as an efficient mechanism for the transfer of argument variables between taskers. The memory in the tank is pipelined and interleaved into several banks so that it can process several memory requests in parallel. Arbitration is needed for access to the shared busses that allow the tank to communicate with the taskers, but if two taskers need objects that are in different memory banks in the tank, then the processors will contend only for the bus, and not for the actual memory retrieval. Once one processor obtains access to the bus and sends a memory request, it can release the bus as it waits for the request to be filled. During this time, the other tasker can obtain the bus and send its request, and the tank can also begin processing this request as well.

A property that the MIT/EC exhibits is that it can tolerate latency in memory references without a loss in performance. This is true of most, if not all, dataflow computers [5]. Since different instructions from different threads of computation can be executed sequentially on any given tasker as it demands new instructions, the delay between the result of one instruction enabling another instruction is inconsequential, as long as there are enough enabled instructions in the queue. A dataflow computer such as the MIT/EC supports the ability to process switch at the instruction level. Just as a multitasking main frame computer achieves good performance by only scheduling those tasks that are not blocked by things such as I/O, a dataflow computer achieves good performance by being able to run any instruction that is available. However, a mainframe computer has a great deal of storage and time overhead in switching the process state from one task to another. In a dataflow computer, there is little or no state that needs to be stored away as it switches from one instruction to another.

Because of the grand scale and extreme complexity of the proposed MIT/EC computer, only a much smaller and simpler computer, the MIT/eC was actually
implemented. The MIT/eC contains only up to 8 taskers. Because the implementation of the taskers out of microprocessing elements was the main focus of Sterling's research, the tank and dispatching unit were eliminated altogether. Instead, the 8 MIT/eC taskers were connected on a standard common S100 bus which interfaced the taskers to a CP/M based Z80 host processor, 64K bytes of dynamic memory and a disk drive as shown in Figure 1-10. The host computer performed as the dispatcher, coordinating the actions of the taskers. The dynamic memory in the host processor could be used as a type of tank to distribute instructions and data for the taskers.

In analyzing the proposed MIT/EC and the implemented MIT/eC, several conclusions were drawn. First, it was shown that although microprocessors could be utilized in the taskers, their interface to the system required a great deal of support components. It was eventually reasoned that by their nature, microprocessors were not very good as processing elements in a multiprocessor structure. This is especially true if the multiprocessor is expected to perform wide-word floating-point calculations. A 16-bit microprocessor such as the Z8001
is not set up to do floating-point calculations efficiently.

Another conclusion drawn was that the way to take advantage of the local interactions that are typical in circuit simulation is to provide enough local memory so that costly global memory accesses can be minimized. Even though the proposed tank was designed with efficient multiport communication in mind, it was determined that the best method for obtaining efficiency was to have redundant memories so that frequently accessed data could be stored locally on each tasker. Also, it was noted that global circuit state could effectively be communicated if the system had provisions for global broadcast mode communications. That is, when global system variables needed to be communicated to all the taskers in the system, multi-point broadcast communication could allow system parallelism and efficiency in the the receiving of the global variables. All the taskers could receive the data at the same time.

Yet another conclusion derived by Sterling was that in order to maximize the exploitation of the concurrency of the algorithms to be executed, a dataflow computer architecture should be used.

This last conclusion has yet to be confirmed. Dataflow computers have been proposed for more than a decade, and yet few successful machines have been built. To this day, no commercially available dataflow machines are in existence. Dataflow has not yet provided a widespread revolution in computer architecture.

In summary, this first attempt at a digital parity simulator, although providing very useful conclusions and having explored a very radical architecture, has not provided a machine with enough processing performance.
1.4.2 The Need for a Multiprocessor with Floating-Point Hardware

Having looked at several digital machines for circuit simulation, one conclusion is perfectly clear: there must be hardware floating-point support. The Cray-1 provides this but it is too expensive. The other digital machines provide varying levels of floating-point support, but other problems such as too small word sizes inhibit floating-point performance. Nevertheless, simply having floating-point hardware will not necessarily solve all the performance problems associated with circuit simulation.

One topic that is of particular interest that has only been briefly alluded to is the method for reducing communication in a circuit simulation. As concluded by Sterling in his analysis of the MIT/EC, having enough local memory to hold redundant copies of information can do a great deal to take the load off of the multiprocessor communication network. Deutsch and Newton in their report on their experiments on the Butterfly multiprocessor have mentioned in passing that there are advantages to partitioning circuit elements to be simulated such that strongly connected networks are simulated on one processor.

If it is the case that circuits can be partitioned in such a way that global communication is minimized, a great deal of the optimization of a multiprocessor for circuit simulation can be done through software. Assuming that the basic hardware support for floating-point operations and some sort of multiprocessor communication is available on the multiprocessor in question, then it should be possible to divide the circuit into partitions that can be processed by each processor with a minimum of communication overhead. As already described in section 1.4.1, physical parity between processing components and the circuit is not possible. But virtual parity in the simulation software is still open.

In order to see how virtual parity could work in a digital parity simulator, the patterns of multiprocessor communication in circuit simulation will be examined.
By making some assumptions about the typical design cycle and by using some ideas from the Lisp programming language, a derivation of a new digital parity simulation architecture (henceforth called the Digital Parity Simulator or DPS) will evolve.

**Multiprocessor Communication in Circuit Simulation**

It has already been concluded that wide word sizes and floating-point hardware are necessary in order to perform numerical circuit simulation efficiently, so all that is left is an understanding of what types of memory systems and multiprocessor communications are needed. First, Sterling’s conclusions that local memory and broadcast communications are essential for a multiprocessor optimized for circuit simulations will be used. The need for local memory is somewhat obvious, as it is clear that the simulation of a partition of the circuit will generate intermediate results that do not need to be stored in slow global memory. If the circuit is divided into partitions, then the local memory on each processor can contain many of the data structures describing that partition. These data structures do not have to be global, unless circuit partitions change as would happen if the topology of the circuit changed. This is a real possibility in power-circuits, where transistors and mechanical switches do change the topology of the network. We will assume, however that circuit topology changes can be made local to one circuit partition residing on one processor. This therefore sets a limit to the granularity of parallelism that we can support. Parallelism will be at the level of partitions, and these partitions will have to be large enough to have constant topology interfaces between them.

The communication between processors in this multiprocessor system will be state variables which are common to topologically connected partitions. These state variables will have to be communicated to other partitions as the variables change through the course of the simulation. The changes will occur as the circuit reaches a new equilibrium state from one simulated time increment to the
next. Therefore, the processors will be performing time-step communication. As new state variables for a partition are computed by one processor for the current time increment other processors will be computing new state variables for the current time increment for their circuit partitions. Therefore, we can imagine that the pattern of multiprocessor operation will be

- **Step 1:** Each processor computes new state variables as functions of the global state variables of the previous time increment.

- **Step 2:** Some sort of multiprocessor communication updates the global state variables.

- **Step 3:** The simulated time increment advances, and all processors go back to step 1.

Therefore, the proposed system will have to support multiprocessor communication in order to update global state variables. It is important to understand the scale of the global state variables. Consider an analog circuit divided into circuit partitions; The global state variables are those at the interface of the partitions. For a circuit with about 100 components partitioned into 10 sub-networks that attempt to minimize the interactions between partitions, there certainly should be no more than 100 total global state variables shared between the partitions. There will probably be much less. Even considering a magnetic circuit simulation where magnetic fields are being simulated with a simulation grid of 1,000 by 1,000 points for a total of one million mesh points, the amount of global state variables is still reasonable. If the magnetic circuit is divided into physical regions, then only the boundary mesh points constitute global state variables that must be communicated between processors. For this example partitioned into 9 pieces, only about 1300 mesh points constitute global state variables. In both examples, system performance will be maximized if this small amount of global data can be updated quickly. A communication network that is optimized for this task will be needed. Note,
however, that where these global variables will reside has yet to be specified. Multiprocessor communication networks will now be examined to see how well they will fit into this model of multiprocessor communication and where global state variables could reside in each case.

**Multiprocessor Communication Networks**

The most simple of communication networks is the shared bus. This is similar to that used in Sterling's MIT/eC computer. Here, arbitration for the bus is needed in order to guarantee that two or more processors do not attempt to put data on the bus simultaneously. The time for sending data from one processor to the next depends on the time needed to get control of the bus. The amount of bus hardware is relatively independent with the number of processors.

The shared bus is an inexpensive system, but it's main drawback is it's lack of expansibility. If too many processors are put on the bus, then the drivers which send data on the bus will overload and reliable communication will be lost.

Many options for the storing of global state variables are possible in a shared bus system. There could be a special memory unit dedicated for this purpose on the bus, or global state variables could be distributed among the processors themselves. In either case, the reads and writes to the global data would necessitate going through the bus arbitration. One very positive characteristic that the shared bus has is the ability to efficiently do broadcast communications. When one processor gets control of the bus, it can send the same data at the same time to all processors.

Another type of communication network is the generalized hypercube. In the hypercube, each processor can send data directly to $N$ other processors, each of which is said to be one one unit away in a particular spatial dimension. In a 3-cube configuration, 8 processors can be connected such that each processor is connected to 3 others. This arrangement can be visualized as a cube with
processors at the vertices. To route data from one processor to another, at most 3 steps are needed: one step in each dimension of the cube. In general a \( M \)-cube has a maximum communication delay proportional to \( M \), while the hardware for communication is proportional to the number of processors times the number of dimensions, \( N \cdot M \).

Hypercube systems have been built, such as the Connection Machine which is connected in 16 dimensions. The one universal fact about hypercube architectures is that they are very complicated from an engineering standpoint. High dimension hypercubes have a tremendous number of wires and switching units, and therefore are often expensive.

The storing of global state variables in a hypercube would have to be done in a distributed fashion. There is no central repository where they could be stored. One organization would be to have the variables stored on the processor nodes where they are computed. Thus, stores would not have to propagate through the network. However, reads of global state variables would have to be directed through the network at the processors where they are stored.

Another type of communication network is the packet switch, as in the Butterfly multiprocessor described in section 1.3.2. Here, any processor can communicate to any other processor through a switching unit that routes memory requests. It has already been stated that for the Butterfly, the amount of switching hardware for a \( N \) processor system is proportional to \( N \cdot \log_4 N \). The communication time between processors for a \( N \) processor system is proportional to \( \log N \). Here there is less communication hardware in comparison to the hypercube, although there are longer communication times.

The packet switch communication system is very general, but like the hypercube, it is complicated. Separate switching units must be constructed and this adds to
the cost of the multiprocessor system.

Systems like the Butterfly support a global address space, so the storage of global variables could go into this global address space. In actuality, the storage would go into the memory on some particular processor nodes, and so references to global state variables on that processor could be completed much more quickly than requests by processors on the other side of the packet switch network. Optimization could be performed to make sure that the storage for global variables occurs on those processor nodes that reference those variables most often.

A third, much simpler communication network is the ring. In a ring network, each processor is connected to only two other processors: one processor that it receives input from, and one processor that it sends output to. When looked at as a whole, the multiprocessor system looks like a circle. An example of a ring-network multiprocessor is the Digital Orrey [3] built to run solar system simulations. For a \( N \) processor system, a ring communications network has switching hardware proportional to \( N \), but the switching hardware is very simple since each switch has just one input, one output, and a connection to the processor. The time to send data from one processor to another is proportional to \( N \). In the worst case, a processor must send data to the processor before it, and the data must go around the entire ring. In the best case, a processor is sending data to the next processor in the ring. The ring network is much less complicated than the other two communication networks, but it has poorer performance.

As in the hypercube, there is no centralized place for the storage of global state variables, so they would have to be distributed among the memory units in each of the processor nodes in the ring. One option that has not yet been considered is to have a copy of the global state variables in each node. This is feasible since
the number of state variables is relatively small. Also, the updating of state variables can be done easily because of the broadcast possibilities in the ring structure. As one processor updates a global state variable in its local memory, this update can propagate down the ring updating that global state variable at each node. If we can synchronize the processors to have a common update period, then the entire update of the system state variables would be proportional to the number of variables updated. While one processor would be sending updates into the ring, the other processors could be receiving these updates while waiting to put their data into the ring. Note that the ring network has the similar broadcast possibilities as in the shared bus, but does not have the unfortunate lack of expansibility that the shared bus has. There are no electrical losses associated with adding arbitrarily many processors in the ring.

Since the ring communication network is so much easier to implement when compared with the hypercube and packet switch networks, it was chosen as the communication network for the DPS. The ramifications of this decision will now be examined.

**Static Load Balancing and the Typical Design Cycle**

Chief among the problems of the ring structure is the fact that communications can take time of the order of the number of processors. This can be accepted, however because the system will be performing time-step simulations with periodic broadcast communications. During the processing stage of a simulation, all processors will be computing data for the current time increment. In order to perform these computations, they will use the copies of global state variables that are in their local memories. Multiprocessor communication will not be needed at this time. When all the processors have computed new state variables, then these state variables will be broadcast to all processors in the ring at the same time.

The other problem that must be contended with is the nature of the system
having to wait for all processors to finish before beginning the global multiprocessor communication. This is a problem of processor utilization. If the simulation is distributed such that one processor is simulating a circuit partition that requires much more processing time than the other partitions, then the entire system will have to wait for that processor to finish. The time between global communication cycles is constrained to be at least as long as the time to operate on the most time-consuming partition. Thus there is a serious possibility of many processors waiting idly for a communication cycle while one processor is performing a disproportionally large amount of the simulation. In order to avoid this, the processing load must be guaranteed to be fairly distributed among all the processors in the ring.

It is impossible to maintain this guarantee in general for the static case. There is no general way to partition and distribute the circuit at the start of the simulation such that a guaranteed balanced load will be achieved. Nevertheless, the fact that the circuit simulation will be a part of circuit design can provide some reprieve. As designs evolve and change, opportunities arise to balance the processing load. The load does not have to be perfectly balanced at the start of the first simulation. Metering can be performed to analyze the simulation as it proceeds, and the processing load can be further balanced, either automatically by the computer, or by hand, at the start of the next simulation. As a better understanding of the issues of load balancing for circuit simulation is obtained, eventually software can be written that causes the load balancing to occur automatically by the computer at the start of every simulation. It is even conceivable that dynamic load balancing could be possible. As the simulation proceeds, the processors could determine that the distribution of circuit partitions is less than optimal, and the circuit could be redistributed in the midst of the simulation.
Lisp and Numerical Circuit Simulation

In the previous discussion, little has been mentioned about the implementation of data structures for numerical circuit simulation. One data structure which has been greatly overlooked due to some of its performance penalties has traditionally been very important for Lisp implementations. This data structure is the linked-list. The advantages that these data structures can provide will now be discussed. Also it will become clear that there is a strong motivation when doing circuit simulation to use not only lists as data structures but also Lisp as the proper way to describe how to manipulate these data structures. Many Lisp implementations allow the blurring of the distinction between programs and data, and it a fact that this can be used to to describe circuits elegantly and efficiently. As some programming issues are addressed, it will become obvious that there is an ultimate need for a high performance Lisp machine that can simultaneously provide the advantages of higher-order procedures without sacrificing the speed of floating-point operations.

In order that the terms used in the following discussion are clear and unambiguous, the following definitions will be used throughout this thesis.

List
Also called a linked-list, this data structure is composed of logical pairs. Each pair has two components. The head of the pair contains the address of a data element which is said to be contained in the list\(^9\). The tail of the list contains the address of the next pair. The last pair has as its tail a unique identifier indicating the end of the list.

Storage Vector
This is a data structure composed of a contiguous block of memory locations. Each memory location contains a piece of data, and the length of the data structure is stored in some convenient location, such as the first memory location of the vector.

\(^9\)Alternatively, it could contain the actual data item itself, if it can fit in the space allocated for the pointer.
Numerical Vector  This is a mathematical abstraction, used to denote a multi-dimensional value.

Concurrent with the design of the Digital Parity Simulator architecture, a member of our group, has been pursuing the use of linked-lists in algorithms for working with sparse matrices. This work indicates that lists, as compared with storage vectors, provide some advantages. As an example, the use of lists to store numerical vectors allows several ways of referencing the same data locations. Addressing structures for the elements in a vector can be constructed out of lists. Each element of the list could point to one of the elements in a vector in a matrix. These lists could be organized by both rows and columns, and because of the indirection, the effects of scaling a row in a matrix will also be visible to the list structure that addresses a column in the matrix. The use of lists as addressing structures causes about twice as much storage to be used since the lists will contain a pointer to the data instead of the data itself.

The price of using list structures as a general data structure is an approximately two-fold increase in storage size and an apparent two-fold increase in latency for data references to elements in lists. In order to dismiss the increase in storage size, one merely needs to consider the types of circuit simulation problems that are to be solved. First, analog circuit problems will deal primarily with a moderate number of complex components and will therefore contain relatively small matrices. Second, most circuit topologies exhibit sparsity. Each component is connected to only a few other components so the matrices will contain mostly zeros. Algorithms exist to take advantage of these sparse structures in order to reduce storage space and reduce the number of operations used to manipulate these structures. Considering the optimizations that can be taken advantage of, it becomes apparent that the size of memory needed to solve a problem can be made small enough so that the doubling of the storage imposed by using lists is not an issue. Also, since multiprocessors will be used to perform
the simulations, the storage of the matrices can be distributed, thus further reducing the total amount of storage that is needed on each processor.

Architectural enhancements can do a great deal to reduce the cost of the extra memory accesses required when lists are used for data structures. The most crucial of these enhancements are the ability to issue memory requests quickly and to be able to do memory indirection efficiently. The Digital Parity Simulator that is being proposed in this thesis has both of these enhancements at a minimal monetary expense, and the effect on performance will be demonstrated in Chapter Two with an example.

Besides using lists as data structures for circuit simulation, it is also the case that the Lisp language can be effectively used. Besides the advantages of simple and consistent syntax and the ability to express and manipulate higher-order procedures, Lisp environments typically have sophisticated and useful debugging tools that make program development easier than languages which support debugging only at the assembly language level. With first class procedure objects, Lisp allows many different programming methodologies. One that could be useful in circuit simulation is object oriented programming with message passing. Circuit partitions could be described as procedures which take messages from other partitions (procedures). These circuit objects could receive messages in the form of updates to global variables. This could allow a somewhat straightforward mapping between partitions at a software level and distributing the partitions at the hardware level.

To explore the possibilities of using Lisp to describe circuit simulation, algorithms for digital parity simulation were actually developed in a dialect of Lisp called Scheme [1]. The report on this work is being developed concurrently with this thesis.
One problem in using Lisp to run circuit simulations is the fact that machines optimized to run Lisp have often neglected floating-point performance. Most of the hardware optimizations in contemporary Lisp machines are focused around efficient manipulation of tagged objects and efficient and flexible procedure calls. Also, compilation techniques for efficiently optimizing numeric algorithms in Lisp have been slow in coming.

There is an ultimate need for a floating-point Lisp machine. The ease of program development and the expressiveness inherent in Lisp must be combined with powerful numeric hardware and advanced compilation techniques. Preliminary attempts at this are starting, notably the Scheme86 processor [21] developed at M.I.T., but much more work is needed in this area. Because of the enormity of the task of producing a full-fledged floating-point Lisp machine, the DPS architecture is not a full-fledged Lisp machine. It is not even close, at the control level. The DPS architecture, as it will be described in Chapter Two, contains just enough hardware optimizations to be able to deal with Lisp data structures. As the architecture is studied and analyzed, further optimizations of the DPS architecture for Lisp operations may be warranted.

1.5 Circuit Simulation Conclusions and Summary

This chapter looked at analog circuit simulation. A number of computer architectures that have been used to perform circuit simulation have been examined. Many strengths and weaknesses have been uncovered in the match between the problems of circuit simulation and the architectures used to solve these problems. Digital parity simulation was described as a way to use multiprocessor digital computers to efficiently perform numerical circuit simulation by optimizing the software to minimize the amount of intra-processor communication.
The main conclusion of this chapter was that a digital multiprocessor with a wide enough word size and floating-point hardware was essential for performing digital parity simulation. Various multiprocessor communication networks were analyzed, and the simplest expandable network, the ring, was discovered to be useful if certain software issues such as load-balancing could be resolved. It was stated that optimizations for list data structures were available cheaply, and the use of lists was argued to be important for circuit simulation if the lists could be manipulated efficiently. The generality of using Lisp as a language for expressing circuit simulations was argued as desirable, but the complexities of building a multiprocessor to interpret Lisp as well as perform numeric circuit simulation was deemed to be too great at this stage of the research. Instead, a Digital Parity Simulator (DPS) was proposed which would limit itself to just processing lists. A wide word size would be provided as well floating-point hardware.

Next, in Chapter 2, the Digital Parity Simulator architecture will be examined at a much more concrete level. Examples will be shown as to how this architecture will be able to efficiently perform basic primitives in circuit simulation, such as the vector scale-and-add operation used in Gaussian Elimination.
Chapter Two

The Digital Parity Simulator Architecture

This chapter will explore the details of the Digital Parity Simulator. First, the basic system structure of the multiprocessor will be presented. Then, the specifics of each subsystem in the DPS processors will be described. Finally, the entire processor will be analyzed in relation to a microcoded program which is used to perform an important portion of Gaussian Elimination, namely the vector scale-and-add operation.

2.1 System Overview

As described in the previous chapter, the Digital Parity Simulator is a ring connected multiprocessor. Each processor is connected to two neighbors, as shown in Figure 2-1. A processor receives data from one neighbor on its input side, and this processor sends information to the neighbor on its output side. Since the DPS does not contain input/output peripherals such as CRT or disk, the entire system is interfaced to a host processor through a Multibus [12] connection.

As concluded from the analysis in Chapter One, the processors in the Digital Parity Simulator will need wide data paths and floating-point hardware support. The actual data widths and type of floating-point support is dictated to some extent by the available digital building blocks. As the Digital Parity Simulator is described in further detail, it will be shown that the available VLSI processor components motivate a 32-bit architecture. In order to solve the various memory contention problems associated with von Neumann computers described in
Chapter One, the Digital Parity Simulator employs the Harvard architecture. Separate memory units allow the simultaneous references of instructions and data.

Each processor in the ring can be broken down into five major subsystems as shown in Figure 2-2. The subsystems are:

1. the micromachine,

2. the ALU/data-path,

3. the memory system,

4. the multiprocessor communication system, and
5. the host interface system.

Because of advances in VLSI technology, many of the above systems can take advantage of highly integrated VLSI components. These complicated one-chip units provide excellent performance and simplify the design by reducing the total number of components in the system. These components are not without their drawbacks, however, in that they are not particularly flexible and they add complications due to their electrical and thermal properties. Also, printed circuit boards containing these components take longer to design since the components often contain a great number of pins that are concentrated in small area. The following sections will show how highly integrated VLSI components have simplified the design of the DPS system, especially in the micromachine and the ALU/data-path.
For the sake of simplicity and to speed the realization of the Digital Parity Simulator, Advanced Schottky transistor-transistor logic (TTL) [18] is the main technology used in the processor design. Advanced Schottky TTL provides greater speed than most other versions of TTL logic for modest costs and power usages. Emitter-coupled logic (ECL) provides much greater performance, but it is much more expensive and requires much more power.

2.2 Micromachine

The micromachine is the controlling agent in each processor in the DPS system. It contains the program that is executed by a processor, and uses this program to control the other subsystems.

Figure 2-3: DPS Micromachine
The micromachine can be divided into five basic parts, as shown in Figure 2-3. These are:

1. the address unit,
2. the condition-code unit,
3. the instruction latch,
4. the microcode memory, and
5. the instruction register.

2.2.1 Address Unit

The address unit supplies addresses to the microcode memory to fetch instructions. These addresses can come from three sources. These sources are:

1. the microsequencer,
2. the dispatch latch, and
3. the interrupt latch.

These different sources are described in detail below.

Microsequencer

The Texas Instruments SN74AS890 microsequencer can be considered the heart of the micromachine. This 68-pin chip generates 14-bit microcode addresses so that a total of 16 Kwords of microcode memory can be addressed. The chip contains an incrementing program counter, a 9 word stack, two input/output ports, and two decrementing counters. Control inputs allow this microsequencer to select various sources for the address of the next microinstruction to be executed. The possibilities include the address in the program counter, an address from the top of the stack, an address from one of the two input/output ports, an address from one of the two counters, or a four bit offset from any one of these sources. These sources allow the implementation of sequential, return
from subroutine, branch and jump, fixed loop, and multiway branch instructions, respectively. Also, the device contains an interrupt return register which allows the device to process real-time vectored interrupts.

This chip is a major component saver in the DPS system. Containing approximately 2400 gate equivalents, it would take approximately 30 TTL components to implement the functionality of this chip.

In the Digital Parity Simulator, the microsequencer accepts addresses to its I/O ports from two sources. One input port accepts addresses from the current microinstruction word. This is accomplished by feeding part of the microinstruction word, the branch field, into one of the I/O ports of the microsequencer and forcing the microsequencer to supply that address to the memory unit. This allows support for branch and subroutine instructions. The microsequencer can conditionally select the branch address as the address of the next instruction. The selection is a function of the condition-code input that is provided by the condition-code unit. Thus, conditional branch and conditional subroutine instructions are also possible.

The other I/O port on the microsequencer is interfaced to one of the busses in the ALU/data-path. This allows microinstruction addresses to be passed back and forth from the microsequencer to the registers in the integer ALU. This is useful for several reasons. First, it allows the ALU/data-path to compute new addresses at which the DPS processor can execute instructions. Thus, computed branch addresses are possible. Second, it allows the microsequencer to send the address of the current instruction to one of the registers in the integer ALU. This is useful in the case where the 9-word microinstruction stack overflows. This would occur if more than 9-levels of subroutines were called. In this case, the overflowing return addresses could be stored in the registers in the ALU or in the main-memory system. Return addresses would not be lost.
Dispatch Latch

Addresses to the microcode memory can also be supplied by the 14-bit dispatch latch. The dispatch latch is loaded with addresses from the Multibus interface, and it serves two purposes. First, it provides the address in microcode memory where new microcode instructions are stored when programs are loaded from the host processor to one of the DPS processors. Second, it provides an address at which the DPS processor can start executing instructions as directed by the host processor. The dispatch latch is implemented by two 8-bit latches. Only 14 of the actual 16 bits are used to feed address to the microcode memory.

Interrupt Latch

A 14-bit interrupt latch is another source of microcode addresses in the address unit. This latch can be loaded by other processors in the Digital Parity Simulator, and it provides the ability for one processor to interrupt another processor. The mechanism for this interrupt is described in detail in sections 2.5 and 2.6. As in the dispatch latch, the interrupt latch is implemented by two 8-bit latches, of which only 14 of the actual 16 bits are used.

2.2.2 Condition-Code Unit

The condition-code unit consists of a 24-bit register, a 24-to-1 multiplexor, and some combinational logic. The register takes 24 bits of status data from the entire DPS processor as input. This register can be selectively loaded at the beginning of any microinstruction cycle. The register feeds into the 24-to-1 multiplexor, which is controlled by the current microinstruction to select the appropriate status bit. The output of the multiplexor passes through the combinational logic which can selectively invert this signal. The output is feed to the condition-input of the microsequencer. Thus, the microsequencer can do conditional operations which are a function of one of 24 status signals that have been saved in some previous microinstruction cycle. The condition can be
inverted so that both "jump if \textit{status-bit}_i\)," and "jump if not \textit{status-bit}_i\)," instructions can be implemented.

The 24-bit register is implemented with 8-bit registers and the 24-bit multiplexor is implemented as a combination of a 16-bit and a 8-bit multiplexor.

\textbf{2.2.3 Instruction Latch}

The instruction latch is a 96-bit latch which is used to store microinstructions into the microcode memory. Sixteen-bit data from the Multibus interface is stored into one of six sections of this latch. When all sections have been loaded, then the entire 96-bit latch can be used as input to load an instruction into the 96-bit microcode memory. The latch is implemented as 12 8-bit latches.

\textbf{2.2.4 Microcode Memory}

The microcode memory is 96 bits wide, and is initially composed of 12 8K by 8-bit static RAM units. Critical path analysis which is performed at the end of this chapter shows that an access time of about 100\textit{ns} is optimal. Since the current chips are only 8K bytes large, only 13 of the 14 supplied address bits are actually used in referencing memory. These chips contain a no-connect pin as well as an extra chip-enable pin, and are almost pin compatible with a 32K by 8-bit chip that uses these pins as extra address lines. Thus, if 8K turns out to be too small for the microcode memory in the future, 32K by 8-bit chips can be used with a very minor processor board modification. Note, however that in this case only 16K of the memory will be referenced since the microsequencer can supply only 14 bits of address.

The static RAMs contain an address input and a multiplexed I/O port. Therefore, external multiplexing must be provided to direct the outputs of the instruction latch to the I/O ports during microcode store operations. This is
accomplished by taking advantage of the tri-state buffers in the instruction latch and the RAM chips. The outputs of the instruction latch are connected directly to the I/O port on the RAMs. The I/O ports on the RAMs are also connected to the inputs to the instruction register. During normal operation, the outputs of the instruction latch are in a high-impedance mode, and data from the RAMs goes directly from memory to the inputs of the instruction register. When microcode is being loaded into the RAMs, the RAM I/O ports are put into a high-impedance mode, and the outputs of the instruction latch are enabled. Data from the instruction latch is then loaded into the RAMs.

2.2.5 Instruction Register

The 96-bit wide instruction register stores instructions which are addressed by the address unit and are retrieved from the microcode RAM. The register ensures that the control signals are steady during the entire microcycle, even though the address to the micromachine may be changing. It is implemented with 12 8-bit registers.

The 96-bit instruction word that is stored in the instruction register is divided into logical fields. Each field controls a different portion of the DPS processor. For the purpose of this discussion, the interpretation of most of the fields is not important. The interested reader is directed to the appendix for the size and meaning of each field.

The reason that the microcode word is so large is because very little decoding is done on the microcode word. Minimal decoding of the microcode allows most of each cycle to be spent doing actual data manipulation instead of waiting for the instruction to be decoded. The wide microcode also gives the ability to express many parallel operations within each microcode cycle. The floating-point unit can be controlled at the same time as the integer ALU, both can be controlled at
the same time as the main-memory system, and all three can be controlled during microsequencer operations, thus giving great opportunity for overlapped execution. The few cases where the microcode word is encoded are in the control of the drivers of various busses in the system. Many of the system busses have several tri-stated drivers which allow the opportunity for several different devices to drive the same bus at different times. Since only one device should be driving a bus at any given time, decoders interpret several fields in the microinstruction word and guarantee that there is only one device driving each bus during a particular microinstruction cycle.

One field that is quite important in the microinstruction word is the upper 16 bits of the instruction word. These 16 bits can be interpreted as a branch or subroutine address which is fed to one of the input ports of the microsequencer, or as a 16-bit constant which can be inserted into the data path. These dedicated bits allow any microinstruction, except one that is using a microcoded constant, to contain a potential control branch or subroutine call. When this field is used to supply a microcode constant, the 16-bit field is inserted into the lower half of one of the data paths described in section 2.3 by means of a 16-bit buffer called the constant buffer. As stated before, the data paths in the processor are 32 bits wide; Therefore, when a constant from the microinstruction word is used, the processing elements must make sure that they only look at the lower 16 bits of the constant. The upper 16 bits will not be driven, and therefore contain invalid and meaningless information. In order for 32-bit constants to be retrieved from the microinstruction stream, two 16-bit constants have to be combined through the ALU to form a 32 bit datum. This takes two microinstruction cycles to retrieve the two halves of the constant, plus as many cycles as are needed to assemble these two pieces into one datum.
2.3 ALU/Data-Path

Actual computation in a DPS processor is performed in what is called the ALU/data-path. The ALU/data-path can be thought of as containing two types of elements, high-speed registers and arithmetic/logic processing units (ALUs). Since the Digital Parity Simulator is required to process numbers in floating-point format, a floating-point ALU is needed. However, since memory is typically organized in a linear array referenced by integer addresses (fractional addresses are not allowed!), an integer ALU is needed to generate memory references. Therefore, the DPS processors contain both integer and floating-point ALUs, as described below. Fortunately, the integer ALU that is used in the DPS processors contains a register file, and thus a separate register unit is not needed.

It is the availability of two 32-bit VLSI ALUs that has dictated a 32-bit architecture for this first implementation of the Digital Parity Simulator. Although 64-bit floating-point numbers are preferred for simulation purposes due to their increased accuracy, parts which process 64-bit floating point numbers are expensive and scarce at this time. A major concern was that no commercially available 64-bit integrated floating-point adder/multiplier existed at this time. Separate 64-bit adders and 64-bit multipliers are available, but two separate units would further complicate the design at an increased expense. Of course, a separate floating-point adder and floating-point multiplier would increase performance by allowing parallel floating-point additions and multiplications. As described in Chapter One, the Cray-1 achieves a great deal of it’s vector processing performance by the use of separate adders and multipliers. Separate floating-point units may be the proper direction for the DPS processors to evolve to as the technology becomes less expensive.
2.3.1 Registered Integer ALU

The Texas Instruments SN74AS8832 32-bit Registered ALU is the heart of the data-path subsystem. This 208-pin chip provides 64 general registers with an integer ALU that supports one-cycle 32-bit integer addition and subtraction. Multi-cycle 32-bit multiplication and division is supported in the chip both architecturally and in the chip's instruction set. The large number of general-purpose registers offers tremendous flexibility in simulation software.

The registered ALU is organized as a three-bus architecture. The register file is dual-ported, allowing the simultaneous reference of any two of the 64 registers. The two outputs of the register file feed into the two inputs of the ALU by means of two 32-bit busses. The ALU combines these two inputs according to the instruction being executed, and the single output of the ALU can be feed back to store data into the register file by means of a third bus. The chip has three I/O ports which connect to the three busses just described. Two of the I/O ports allow the outputs of the registers to be sampled or for data outside of the chip to be input directly to the ALU. Additional feedback paths also allow data on these ports to be stored directly into a register in the register file. The third I/O port allows the output of the ALU to be sampled or for data outside the chip to be stored into a register. Although the three ports are actually bidirectional, typical operations will utilize the two ports that connect to the inputs of the ALU as input ports, and the port connected to the output of the ALU as an output port. The chip is rated to perform a register-to-ALU-to-register operation in 50ns.

The chip has parity generation and parity checking to verify the data that it
operates on\textsuperscript{10}. The data in the registers is actually stored as 32-bits with an additional 4 bits of parity. The I/O ports also have 4 bits of parity associated with each of them. Error flags are asserted when the chip-generated parity of the data on an I/O port does not match with the parity supplied to the device on that I/O port. In the DPS architecture, the parity functions of the SN74AS8832 are ignored since no other subsystem in the processor is equipped to handle the extra parity bits.

2.3.2 Floating Point Unit

The Advanced Micro Devices Am29325 Floating-Point Processor is a 144-pin chip that supports one-cycle addition, subtraction, and multiplication of 32-bit IEEE-754 floating-point numbers. Conversion between integer and floating-point numbers is also supported. Chip instruction support is provided for the iterative Newton-Raphson division algorithm.

Like the registered ALU, the floating-point unit is also organized as a three-bus architecture. Two input ports feed into 32-bit input registers. The floating-point ALU combines the data from these two registers and produces an output that can be stored into an output register. Internal feedback paths allow the contents of the output register or the output of the ALU to be fed back as inputs to the ALU. Both the input registers and the output register can be disabled to put the the chip into a “flow-through” mode. In the flow-through mode, the device acts as a combinational circuit and the inputs go directly to the ALU and the output of the ALU goes directly to the output port. The chip is rated to do

\textsuperscript{10}Note that the parity described in this section refers to a function of the bits in a piece of data, not to the parity between the simulation and the architecture. In the SN74AS8832, the parity function is as follows: for each 8-bit byte in a 32-bit word, the number of binary ones in the byte is computed. If that sum is odd, then a one is generated for the parity of that byte. Four parity bits for each 32-bit datum allow the parity function to be computed for each of the four 8-bit bytes in the 32-bit datum.
an add, subtract, multiply, or conversion operation in 150ns.

### 2.3.3 ALU Interconnection

The "standard" way to connect three port devices is by matching up all the ports on each of the devices. In this configuration, the outputs of all the devices feed into one bus, and inputs to all the devices are connected to one of two other busses. Transfer registers are typically provided to allow data to move from the output-bus to one of the input busses so that data from the output of one device can move to the input port of another device.

*Figure 2-4: Integer ALU and Floating-Point Processor Interconnection*
The integer ALU and the floating-point processor are not connected in this standard way in the Digital Parity Simulator. Example microcode has shown that a more pipelined interconnection is better suited to vector operations such as the vector-scale-and-add operation. The actual interconnection is shown in Figure 2-4. Here, the outputs of the integer ALU feed into one of the input ports of the floating-point unit and to the address and data inputs in the memory system (see section 2.4). The data output of the memory system feeds into the other input of the floating-point unit as well as to one of the inputs to integer ALU. The output of the floating-point unit feeds into the other input port of the integer ALU. This type of interconnection allows a very efficient vector-type operation as will be shown in section 2.8. Two obvious improvements that this interconnection provides are that it allows data from the memory system to go directly into the floating-point unit while the ALU issues a memory request and that it allows for a result from the floating-point unit to be stored into a register in the registered ALU while the ALU issues a memory request.

2.4 Data Memory System

The data memory system is the main storage unit in the Digital Parity Simulator. This memory system interfaces to both the ALU/data-path and to the multiprocessor communication system as shown in Figure 2-5. These two interfaces both consist of a 16-bit address bus, a 32-bit input data bus and a 32-bit output bus. Control signals select between read and write operations as well as selecting which subsystem has control of the memory. The data memory subsystem provides low-latency memory references of 32-bit operands. Requests take one cycle to complete, and a memory request can be issued each processor cycle. This high speed memory is essential to keep the processing units busy.

The main structure of the memory system consists of a bank of Memory Address
Figure 2-5: Memory System

Registers (MARs) which feed into the address inputs of two banks of static RAM chips. Each memory bank consists of 4 8-bit wide chips which are of the same type as in the microcode memory. These chips can either be 8K by 8 bits or 32K by 8 bits. With two memory banks, either 16K or 64K words of data memory are possible. The I/O ports on the memory chips connect to Memory Data Registers (MDRs) and also act as data outputs to the adjoining subsystems.
Multiplexing allows data from the MDRs to be stored into the memory system at an address selected by a MAR, or for the data at a selected address to be read from the memory unit.

Several 16-bit MARs are provided in the memory system. First, two general purpose MARs are connected to the ALU/data-path. These two MARs (called "MAR1" and "MAR2") allow the saving of repeatedly used memory addresses. This is useful in overlapped read-modify-write operations. First the processor can request the data in location $x$ by storing that address in MAR1. At the end of the next cycle the data is available to be processed by one of the ALUs. During this processing, the memory system is unused, and parallelism can be achieved by retrieving the next piece of data to be operated on. Hence, the address of the next piece of data, $y$, can be stored in MAR2. As the ALUs operate on the data from location $x$, the data in location $y$ can be retrieved. On the next cycle, the processor can store the modified data back into location $x$ by selecting the unchanged MAR1 as the destination address. The processor does not have to send this address again since it has been saved. Without the extra MAR, either the processor would have to send the address $x$ both on the read and the write, or else the read of address $y$ could not happen during the processing of the data at address $x$.

Two other MARs are used to provide support for linked-lists. These are the CAR and CDR memory address registers\(^{11}\). These 16-bit MARs take as input the two halves of the 32-bit data that is retrieved from memory. Since memory words are 32 bits wide and memory addresses are 16 bits wide, two pointers can be stored in one 32-bit word. Thus, a Lisp pair can be implemented as a 32-bit word which contains a head pointer and a tail pointer. The CAR and CDR

---

\(^{11}\) CAR and CDR are traditional names for Lisp primitives that extract the head and tail of a pair.
MARs can be loaded with the two pointers from pairs that are extracted from memory. Therefore, indirecting to get from one pair to the next pair in a list can be performed without the ALU having to extract and manipulate the associated pointers. This allows the processor to perform useful floating-point or integer ALU operations during memory indirection.

The Data Memory subsystem also interfaces to the other processors through the multiprocessor connections. A synchronous global communication cycle with all the processor nodes allows data to be transferred at the clock speed of the individual processors. A global transfer of data is accomplished by having the sending processor load addresses into one of the two MARs and selecting data to be sent to the other processors. The selected address from the MAR and the corresponding data from the output of the memory banks are directed to the receiving processor next in the ring through the trunchevers. From the trunchevers, the addresses and data move to the pipeline registers on the next processor where they can be used to load the memory bank on that processor. The address and data can propagate around the ring network so that copies of data in the sending processor's memory can be stored on each processor. The intricacies of the sending of data from one processor to the next are further explained in the next section.

2.5 Multiprocessor Communication Interface

The multiprocessor communication interface connects the memory systems on different processors. It provides for synchronous multiprocessor ring communication at the speed of the processors. Synchronous communication is possible because all the processors in the DPS system run from a common clock supplied through the host interface (see section 2.6). The multiprocessor communication interface implements the ring structure of the DPS
multiprocessor as shown in Figure 2-6 with ribbon cables that connect to 60-pin connectors on each processor.

The multiprocessor communication system consists primarily of a set of address and data registers. On processor \( n \) in the ring, the 16-bit address registers accept addresses from processor \( n-1 \). These addresses can be sampled as well as directed to processor \( n+1 \). The same is true for the 32-bit data registers. The sampling of data in the ring is accomplished through the memory system tranceivers described in the previous section. Addresses and data from processor \( n \) can be injected into the ring by reversing the direction of the tranceivers and directing addresses and data from the memory system to the next processor in the ring.

A second subsystem in the multiprocessor communication network is the interrupt subsystem. The interrupt subsystem allows one processor to interrupt another processor in the ring by forcing it to start executing instructions at a selected microcode address. There are two parts to the implementation of this function. First, addresses that move in the ring pipeline can be considered as addresses of potential processors to be interrupted if a special tri-state interrupt line is driven in the ring. When this interrupt line is driven, then addresses in the ring pipeline are interpreted as follows. For each bit in a 16-bit address in the pipeline, a high bit indicates that the corresponding processor is about to be interrupted. Thus if bit \( n \) is high and the interrupt line is driven, then processor \( n \) is being interrupted. This allows up to 16 processors to be simultaneously interrupted. A processor can ignore the interrupt if it is not currently enabling interrupts. The interrupt enable is one of the fields in the microcode instruction word. If interrupts are enabled, however, then the interrupt latch in the micromachine is loaded with the value on the 14 lower bits of the data portion of the ring. One cycle later, the interrupt latch drives the address of the microcode memory, and the processor effectively jumps to that address. Because the microsequencer contains an interrupt return register, the processor can resume
Ring Bus Address Input

Ring Bus Data Input

Ring Token Input

Ring Mode Input

Need Input

Pass Input

Figure 2-6: Multiprocessor Communication System
what it was doing after the interrupt is processed.

The third portion of the multiprocessor communication system is the token subsystem. One-bit tokens are used in the DPS system for arbitration and for the validation of data. These tokens are implemented with one-bit registers which are connected in the same way as the address and data registers in the communication system. The ring structured address, data, and token registers can be thought of as a recirculating pipeline. A valid token is a high bit that moves in this pipeline. There is no priori limitation to how few or how many valid tokens can be in the ring at any given instant.

Three operations on tokens are available to each processor on the ring. These are:

- pass any incoming tokens to the next processor in the ring,
- absorb (do not pass) incoming tokens, and
- create a new outgoing token to be passed to the next processor in the ring.

These operations are achieved by combinational logic between the output of the token register and the input to the next processor. A token is valid for a particular node if that processor's token register contains a high bit. A token's validity is irrespective of whether or not the token is passed to the next processor.

There are several uses for the token in the Digital Parity Simulator. First, if upon system initialization every processor absorbs incoming tokens, after a time the ring will be free of valid tokens. At this point, one processor can count how many processors are in the ring by creating a token and counting how many cycles are needed before the token returns. Note that this depends on the fact that the entire system is synchronous. Second, a token can be used for
arbitration. Consider normal system operation in which there is only one token in the ring. If a processor needs exclusive access to a resource, such as access to the interrupt line, it can use the token to tell it when it can use the resource. If it has valid token, it absorbs it so that other processors won’t attempt to use the resource. It can then use the resource. When it is done with the resource, that processor should create a new token and pass it on to the next processor. Third, tokens can be used to verify the validity of data in the ring. For instance, in a memory transfer over the ring, the token is used to assert a write pulse synchronous with the data moving from a source to a destination processor.

In the Digital Parity Simulator, the token can be used for both arbitration and validation during normal operation by the use of additional control lines in the ring. These control lines can be asserted by any processor and allow different interpretations of the token depending on the current state of the control lines.

The fourth subsystem in the multiprocessor communication interface is the request logic. The request logic processes and generates requests for global communication cycles. As stated in Chapter One, the primary mode of communication in the multiprocessor will be the broadcast of common state variables. When a processor has finished computing the new values of the state variables for which it is responsible, it asserts a “communication-request” signal. This signal propagates asynchronously around the ring and is logically ANDed with the communication-request signal at each processor. One processor is deemed the system master, and it listens to this signal. When it is high, indicating that all processors are ready for a global communication cycle, the master signals that a global communication cycle is about to begin by dropping the communication-request signal for one cycle. Each processor waiting for a communication cycle goes into a “transparent mode” when it detects this signal from the master. In transparent mode, addresses and data from the ring go directly to the memory system through the tranceivers, and valid tokens generate
write pulses to store the ring data into the receiving processors. Therefore, the master can start sending its updated state variables around the ring by forcing its tranceivers to send appropriate addresses and data into the ring and by creating new tokens to validate this data. When the master is done updating its state variables, it asserts a “pass” signal and goes into the transparent mode. When a processor in transparent mode receives the pass signal, it then can go into a sending mode and start sending its new state variables around the ring. Finally, when the master receives a pass signal, it knows that all processors have sent copies of their state variables around the ring. Thus, the global system state has been updated in a broadcast fashion, and computation of the next state of the system can begin. The master signals this by dropping the need request signal, at which all the processors can start computing new state variables.

The use of extra signals associated with the tokens allows other communication modes to be established during a simulation. Although global state update is the primary communication mode, local communication between several processors is also possible when a global communication cycle is not in progress. An example of this other communication mode is as follows. In normal operation between global communication cycles, one token circulates in the ring for arbitration. A token is valid for arbitration when the token is valid and no other control lines are asserted. If a processor decides to perform a non-global send to another processor, it grabs the token and asserts the common interrupt line. It then creates a token and sends it simultaneously with the proper mask for the address of the destination processor. At any given processor, an interrupt is generated if the interrupts are enabled, the interrupt line is asserted, the token is valid, and the address mask in the ring matches its own address. If all these conditions are true, the interrupted processor absorbs the address and token and sends out a response message (without a token). The reason a token is not sent is to prevent other processors from interpreting the response as an address. The initiator of
the interrupt listens for a proper response (without a token).

Once the request and response are completed, the interrupt line can be dropped and a “local transfer” line can be asserted. Only the interrupted processor will have enabled the transparent mode in which tokens on the ring will generate write pulses for its memory. Thus the sending processor can now send data with valid tokens. As each token reaches the receiving processor, a write pulse will be generated and the data that is synchronous with the token will be written into the receivers memory. The other processors will not store this data since they will not be in transparent mode, and they will not interpret these tokens as arbitration tokens since the local transfer line will be asserted. Thus global communication cycles are completely compatible with point-to-point local communications in the ring.

Although there is at yet no clear use for the local communication in the ring, it was implemented since it added flexibility at an only small increase in system cost and design complexity. The token system was already needed for the global communication cycle to generate the write pulses as global data is broadcast throughout the pipeline. Thus, the only extra hardware needed to implement the extra local communication cycle is the hardware needed to support the interrupt and local transfer modes. This consists of a priority decoder chip to decode interrupt addresses, a pair of line drivers, and the two extra bits in the microcode instruction word needed to control these drivers.

### 2.6 Host Interface

The host interface system allows the processors in the Digital Parity Simulator to communicate to the outside world. On each processor, there are two basic pieces to the host interface system as shown in Figure 2-7.
The address interpretation unit decodes addresses from the host processor. Once address are interpreted, the unit supplies control signals to perform various operations on the DPS processor and generates the appropriate acknowledgment signal to the host. The buffer unit sends data between the host processor and various systems on the DPS processor and conforms to the electrical and timing specifications of the Multibus.

The interface system allows the connection of the DPS processors to a Multibus backplane. This allows communication with an Intel 310 microcomputer which also resides on the bus. The Multibus contains 16 address lines and 16 data lines. The addresses can be interpreted as memory addresses, or as I/O addresses depending on whether the bus request is for memory or I/O. The DPS processors reside in the I/O space of the Multibus system, and the I/O
communication over the Multibus is accomplished through a standard send-acknowledge protocol. During an I/O write from the host processor to an I/O device on the bus, the host processor puts the destination address on the address lines and the desired data on the data lines. It then activates the *IOWC line which indicates an I/O write. All the devices listen to the address lines, and the device whose address matches the address on the bus loads the data. It signals the completion of this loading by activating the *ACK acknowledge line. Upon receiving the acknowledgment, the host processor then deactivates the *IOWC line and stops driving the address and data lines. When the *IOWC line is deactivated, the I/O device stops driving the *ACK line. For I/O reads, the host processor puts the source address on the address lines and activates the *IORC line which indicates an I/O read. The I/O devices listen to the address lines, and the matching device sends the desired data by driving the data lines and activating the *ACK line. Once the host processor has finished sampling the returning data, it deactivates the *IORC line, after which the selected I/O device stops driving the data and deactivates the *ACK line.

The DPS processors reside in the I/O space, so they each listen to addresses on the Multibus. The upper 7 bits of an I/O address are used to select the DPS system for an I/O transfer. If the upper 7 bits match an address coded through DIP switches on the DPS processors themselves, then the DPS system is being addressed as the source or destination of an I/O transfer with the host processor. The next 4 lower bits of the I/O address select which processor the host is attempting to communicate with. If all 4 bits are high, then all processors are being selected. This only makes sense for an I/O write from the host to all the DPS processors in the system, as in the case of loading common microcode to all processors. When all processors are being selected and written to, the address interpretation unit makes sure that only one DPS processor generates the acknowledgment signal. DIP switches on each processor determines which
processor will activate the *ACK signal when all the processors are being written to. With 4 bits, 16 different addresses are possible, and one of these addresses, namely 15, is specially interpreted. Thus there is a maximum of 15 addressable DPS processors in this implementation. The next 4 bits in the I/O address contain the command to be processed in the I/O transfer. The lowest bit in an I/O address is unused since it is always zero when the Multibus is performing 16 bit transfers. The DPS interface to the Multibus is 16 bits so all transfers to and from the host processor are 16 bits wide.

With 4 bits in the command field of the I/O address, there are 16 possible commands. Only 12 different commands are actually implemented, and these are:

- load one of 6 portions of the microcode latch,
- load the input latch,
- load the dispatch latch,
- force the address in the dispatch latch as the address of the next instruction,
- force the address in the dispatch latch and load the microcode word in the dispatch latch into the microcode memory,
- force the address in the dispatch latch and cause the microsequencer to save an interrupt return address, and
- read the value from the output register.

It is important to note that only the last of the above commands is actually a read command. The rest of the commands can be implemented as write commands. The first 6 commands are used to assemble a 96-bit microinstruction word to be loaded. The microinstruction word is assembled by 6 16-bit portions that come from the Multibus. Another command allows the dispatch latch to be loaded. With these two abilities, another command forces the microinstruction
word to be loaded into the microcode memory. Other commands allow data to be transferred to the ALU/data-path through the input latch or for data to be sent from the ALU/data-path to the host processor through the output register. A useful command that was easy to implement was for the host processor to interrupt a DPS processor. This was facilitated by the existence of the interrupt return register in the microsequencer which pushes onto the stack the return address for when the interrupt finished being processed. Because the host processor runs asynchronously with respect to the DPS processors, the interrupt address and the interrupt request are buffered in order to be sent to the processor synchronous with the DPS system clock.

Since all Multibus transfers are initiated by a Multibus master, and since none of the DPS processors are implemented as bus masters, all I/O transfers to and from the DPS system are initiated by the host processor. The ability for a DPS processor to interrupt the host was deemed too expensive in terms of increased hardware and system complexity. Nevertheless, there is no great loss of flexibility in this decision. While simulations are running on the Digital Parity Simulator, the host processor can periodically sample each processor's output register in order to detect any sort of trap or error code that has been put there by a processor requesting special service from the host.

In order to maintain reliability in data transfer from the host to the processors, two one-bit status registers are automatically updated during I/O operations. One status register is associated with the input latch. When the host processor writes data to the input latch, the input status register is set indicating that data is available. The output of this status register is directed to the condition-code unit in the micromachine so that the DPS processor can test for the validity of input data. The input status register is automatically cleared when the input latch is read by the ALU/data-path. The other one-bit register is associated with the output register. The bit in this register automatically gets set when the
ALU/data-path loads data into the output register. When the output register is read by the host processor, the status bit is cleared. This status bit is also directed to the condition-code unit so that the DPS processor can detect when the data that it is sending to the host processor has actually been read by the host. It is important to note that although this system prevents the DPS from reading invalid data or from overwriting data that has not yet been read by the host, the converse is not true. That is, nothing stops the host processor from overwriting data in the input latch before the DPS processor reads it or from reading data multiple times from the output register. This can be remedied by having the DPS processor use changing values in the output register to perform handshaking and by enforcing upper bound response times to host requests.

2.6.1 Intel 310 Host Processor

The host processor for the DPS system will initially be an Intel 310 microcomputer. It contains an 8086 16-bit microprocessor, a 8087 floating-point coprocessor, dynamic memory, and hard and floppy disk drives all connected by a Multibus backplane. It runs the iRMX\textsuperscript{TM} \textsuperscript{12} operating system, and a C language compiler and a 8086 assembler are available for program development. In the software for interface to the DPS system, a few basic I/O routines will be hand-coded in assembler, but most of the software will be written in C. The software that will be written for the DPS system includes a microassembler to generate microcode instruction words out of symbolic microcode, a downloading program to store microcode into the DPS processors, interface software to transfer data to and from the DPS processors, and software to monitor system status and simulation progress. As of this time, the microassembler is nearly complete, and the other software packages are in the design stages.

\textsuperscript{12}iRMX is a trademark of Intel Corporation.
Because the DPS processor boards do not conform to the Multibus form factor, a modified interface Multibus card cage is used to house the DPS processors. The card cage interfaces to the Multibus on the Intel 310 through a buffer card. This card ensures that the host processor can reliably drive all the inputs on the DPS processors. It also provides clocking circuitry for the DPS the system clock and distributes power supply signals needed to power the DPS processors.

2.6.2 Expansion Processors

Because the DPS processors connect to the Multibus, there is opportunity for other specialized processors to be incorporated into the DPS system. Currently, there are plans to include a graphics processor into the Digital Parity Simulator system. This processor could receive programs from the host processor through the Multibus and could receive data to display from the DPS processors through the multiprocessor ring connections. If this processor has a similar memory interface to the ring as the other DPS processors, then it could be programmed to display the changes in state variables as they are updated every global communication cycle. It would have a copy of the global state variables and could access these in between communication cycles. The graphics processor would receive all the benefits of the global broadcast communication in the ring, but it would never be a sender of data.

Other possibilities for specialized processors are dedicated data-acquisition and control units. These could be used to receive outside data and perform real-time control by analyzing system function simulation performed on the regular DPS processors.
2.7 Critical Path Analysis

All the systems in the Digital Parity Simulator have now been described in some
detail. In order to calculate the maximum clock frequency at which the system
can execute microcode instructions, it is necessary to trace a microinstruction
cycle during its execution. Figure 2-8 is useful in this discussion in order to
follow the electrical signals as they flow through the system.

Critical path analysis in a microcoded system is typically divided into two pieces.
One is the analysis of the critical path in the control system, and the other is the
analysis of the critical path in the data-path system. The best possible machine
cycle time is the maximum of these two critical timing constraints. In this
discussion, the following notation will be used.

\[ \tau_{\text{clkpd}} \]
This is the propagation delay through a device from the rising
edge of a the clock until the output of the device is valid.

\[ \tau_{\text{pd}} \]
This is the propagation delay through a device from the
change of an input signal until the output of the device is
valid.

\[ \tau_{\text{access}} \]
This is the propagation delay for a memory device from the
change of an input address until the output of the memory is
valid.

\[ \tau_{\text{setup}} \]
This is the minimum time that data must be valid before the
rising edge of the clock in order to properly store data into a
register.

Note that a device may have several different \( \tau_{\text{clkpd}} \)'s and \( \tau_{\text{pd}} \)'s since it may have
several different inputs and outputs.
Figure 2-8: Control and Dataflow in the Digital Parity Simulator
2.7.1 Micromachine Critical Path

For the purpose of this discussion, all microcycles will be said to begin and end at the rising edge of the system clock. In the micromachine, the two important events that happen on the rising edge of the clock are the loading of the microcode instruction register and the loading of the status register. From these events, a microcode address must be selected and fed into the microcode memory. The microcode memory must deliver a new microcode word to be loaded into the instruction register. In this analysis, the access time of the microcode memory will be a variable chosen to balance out the entire system. Hence, the maximum paths will be calculated with the understanding that the access time for microcode memory will have to be added.

There are four important paths from the clock edge to the generation of a microcode address. These are:

1. from the status register, through the condition-code multiplexor, to the condition-code input in the microsequencer, and to the selection and output of a microcode address,

2. from one of the extra counters in the microsequencer to the output of a microcode address,

3. from the branch field in the microcode word, through an I/O port in the microsequencer, and to the output of a microcode address, and

4. from the I/O port in the integer ALU, through an I/O port in the microsequencer, and to the output of a microcode address.

The first path can be calculated as

\[ \tau_{reg_{clkpd}} + \tau_{mux_{pd}} + \tau_{logic_{pd}} + \tau_{museq_{pd}} \]

This reduces to \( 9ns + 20ns + 10ns + 29ns = 68ns \). The second path can be calculated as \( \tau_{museq_{clkpd}} \). This reduces to 50ns. The third path can be calculated as

\[ \tau_{reg_{clkpd}} + \tau_{museq_{pd}} \]
This reduces to $9ns + 14ns = 25ns$. The fourth path can be calculated as

$$\tau - ALU_{clkpd} + \tau - \mu seq_{pd}$$

This reduces to $17ns + 16ns = 33ns$. The maximum path of these four is the first one, so at most $68ns$ are needed for the generation of a microcode address. After an address is generated, the corresponding word in the microcode memory must be accessed, and the word must be presented to the instruction register in order to meet the setup time of the register. Thus the total control system timing can be calculated as

$$\tau - clock = 68ns + \tau - \mu code_{access} + \tau - reg_{setup}$$

This reduces to

$$\tau - clock = 68ns + \tau - \mu code_{access} + 3ns$$

or

$$\tau - clock = 71ns + \tau - \mu code_{access}$$

(2-1)

2.7.2 Data Critical Path

In the data path section of the DPS processor, there are several paths to consider. Again, a cycle will be considered to begin and end on the rising edge of the clock. The following 5 paths are important:

1. from the Y output of the ALU to any input such as the register file, the floating-point unit, or the MAR, MDR or output-data registers,

2. from the input registers in the floating-point unit through the floating-point ALU to the output register in the floating-point unit,

3. from the output of the floating-point unit to a register in the registered ALU,

4. from the selection of a MAR to the output of data from the memory system to either a register in the ALU, or a register in the floating-point unit, or the CAR and CDR registers, and

5. from the selection of a MAR to the output of data from the memory system through the data tranceiver to the pipeline register on the
next processor.

The first path can be calculated as
\[ \tau - ALU_{clkpd} + \max (\tau - \text{reg-file}_{setup}, \tau - \text{float}_{setup}, \tau - \text{reg}_{setup}) \]
This reduces to \( 38ns + \max (12ns, 10ns, 3ns) = 50ns \). The second path can be calculated as
\[ \tau - \text{float}_{clkpd} + \tau - \text{float}_{pd} + \tau - \text{float}_{setup} \]
This reduces to \( 10ns + 150ns + 10ns = 170ns \). The third path can be calculated as
\[ \tau - \text{float}_{clkpd} + \tau - \text{reg-file}_{setup} \]
This reduces to \( 10ns + 12ns = 22ns \). The fourth path can be calculated as
\[ \tau - \text{reg}_{clkpd} + \tau - \text{data}_{access} + \max (\tau - \text{reg-file}_{setup}, \tau - \text{float}_{setup}, \tau - \text{reg}_{setup}) \]
This reduces to
\[ 9ns + \tau - \text{data}_{access} + \max (12ns, 10ns, 3ns) = 21ns + \tau - \text{data}_{access} \]
The fifth path can be calculated as
\[ \tau - \text{reg}_{clkpd} + \tau - \text{data}_{access} + \tau - \text{receiver}_{pd} + \tau - \text{reg}_{setup} \]
This reduces to
\[ 9ns + \tau - \text{data}_{access} + 10ns + 3ns = 22ns + \tau - \text{data}_{access} \]

From this analysis, the maximum time in the data-path section of the DPS processor is 170ns, if the \( \tau - \text{data}_{access} \) is minimized. The maximum term which contains the data memory access time is the fifth equation, so it will be the case that
\[ \tau - \text{clock} = 22ns + \tau - \text{data}_{access} \] (2-2)

### 2.7.3 Critical Path Conclusions

From the previous analysis, it is obvious that the bottleneck is in the data-path section, namely the floating-point unit. This forces a 170ns cycle time, if memory timings in equations 2-1 and 2-2 can be satisfied. This will be the case if \( \tau - \mu code_{access} = 170ns - 71ns = 99ns \) and \( \tau - \text{data}_{access} = 170ns - 22ns = 148ns \). These timings are reasonable for static RAMs so a 170ns system clock period should be expected.
2.8 Microcode Examples

Now that the DPS system has been overviewed and the expected clock period has been calculated, the expected performance of a DPS processor can be examined. For this exercise, two examples will be looked at in detail. Both of these examples will be implementations of the vector scale-and-add operation which is a crucial part of Gaussian Elimination. These routines are expected to be much-used primitives in the simulation system. The two separate microcode implementations reflect two of the many possible numerical vector representations that this architecture can support.

2.8.1 Vector Scale-and-Add with Storage Vectors

![Diagram of storage vectors](image)

**Figure 2-9:** Representation for Storage Vectors

The vector scale-and-add operation, $\vec{z} = \vec{a} + k \cdot \vec{b}$ is to be performed. In this example, storage vectors will be used to represent the numerical vectors. The bases of the three vectors are stored in registers in the registered ALU. Registers
1 and 2 contain the addresses of the bases of vectors $\vec{a}$ and $\vec{b}$, respectively. Register 3 contains the address of a free block of memory large enough to hold the vector $\vec{c}$ that will be created. The scalar constant $k$ is in register 0, and other registers will be used to hold temporary values. Vectors are represented as contiguous 32-bit words in memory. The first word is an integer header that indicates the number of elements in the vector, and all the other words contain 32-bit floating-point numbers that are the elements of the vector, as shown in Figure 2-9.

The code in these examples is symbolic microcode that could be assembled using the DPS microassembler. The symbolic microcode contains labels, branches, and processor directives. Microcode comments will explain the syntax or code that is non-intuitive. Comments are lines that begin with ";".
When we start execution at #SAA_1, register 1 (REG1) will contain the base of \( \mathcal{A} \) and register 2 (REG2) will contain the base of \( \mathcal{B} \). Register 3 (REG3) will contain the address of a free block of memory that can be used to allocate the new vector \( \mathcal{C} \). Register 0 (REG0) will contain the scaling constant, \( k \).

Start of scale-and-add (SAA) code. First, load the scaling constant \( k \) into a register in the floating-point unit.

\#SAA_1: FLOATS <- REG0.

Next, request length of \( \mathcal{A} \).

MAR1 <- REG1.

Next, request length of \( \mathcal{B} \). During this time, receive the previous request by selecting the first Memory Address Register (MAR1). Store the length of \( \mathcal{A} \) in register 4.

SEL MAR1, REG4 <- MEM, MAR2 <- REG2.

Now, receive the length of \( \mathcal{B} \) and store it in register 5.

Also, prepare to write the length of the result vector \( \mathcal{C} \) by loading the length of \( \mathcal{A} \) into the Memory Data Register (MDR).

SEL MAR2, REG5 <- MEM, MDR <- REG4.

Now, compare the two lengths to make sure that the vectors are the same length. The compare operation is performed via an integer subtract operation and by loading the condition code register with the flags generated by the ALU.

SUB REG4 REG5, LATCHCC.

If the two vectors are not the same size, then jump to an error routine. During this time, store the address of the header of \( \mathcal{C} \) in case the vectors are the same length and we will actually write out the vector \( \mathcal{C} \).

JMP NOT ZERO #L_ERR, MAR1 <- REG3.

If the vectors are the same length, then prepare for the scale-and-add inner loop. Store the length of the result vector, and prepare to request the first element of \( \mathcal{B} \). We increment the address in register 2 in order to get the address of the first element of \( \mathcal{B} \). We store this incremented value back in register 2 so that we can use this to increment again. This store code also doubles as the store instruction that is executed for each element of \( \mathcal{C} \) as it is calculated in this loop of code. (A period "\( .\)" separates multiple destinations of the same source datum.)

\#Loop: SEL MAR1, MEMWRITE, MAR2.REG2 <- REG2 +1.
; #SAA_1 continued

; Now we receive the element of \( b \) and store it in an
; input register in the floating-point unit. We issue the request for the
; element of \( \bar{a} \). Note that we are also incrementing and
; storing the base address of \( \bar{a} \).

    SEL MAR2, FLOATR <- MEM, MAR1.REG1 <- REG1 +1,

; At this point, we commence the scaling operation by doing a
; floating-point multiply operation. The result will be stored into
; the output register of the floating-point unit, FLOATF. We can at this
; time get ready to do the add operation by loading FLOATR with the
; element of \( \bar{a} \) that is coming from memory.

    SEL MAR1, FLOATR <- MEM, FLOATF <- FLOAT* FLOATR FLOATS,

; Now we do the add operation through an internal feedthrough path in
; the floating-point-processor. Also, we prepare to write out the
; result in \( \bar{c} \) by storing the address of the next element
; of \( \bar{c} \) in MAR1.

    FLOATF <- FLOAT+ FLOATR FLOATF, MAR1.REG3 <- REG3 +1,

; The result of the scale and add is stored into a temporary register.
; We can use this otherwise wasted cycle to decrement the number of
; elements left to do.

    REG6 <- FLOATF, REG4 <- REG4 -1, LATCHCC,

; Now we loop back if we still have more elements to process. We load
; the MDR with the result of the scale and add, and the actual write
; into memory will be performed at the #Loop instruction. If there
; are no more elements to process, then we fall through to the end of
; the routine.

    JMP NOT ZERO #Loop, MDR <- REG6,

; End of #SAA_1 routine. Perform the final write operation.

#Done:  SEL MAR1, MEMWRITE,

The inner loop for this code contains 6 instructions. Running at a 170ns clock cycle, a Digital Parity Simulator Processor will achieve the scale-and-add performance of 1.02\( \mu \)s per element.
2.8.2 Vector Scale-and-Add with Linked-Lists

An alternative way to represent numerical vectors is to use linked-lists. There are many ways to represent numerical vectors with linked-lists. One major distinction between various representations is whether or not support for sparsity is provided. If a numerical vector contains only a few non-zero elements, then it may be advantageous to store only the non-zero elements. In the case of the DPS architecture, however, software support for sparsity provides few advantages for most cases. First, since most simulations will contain a modest number of nodes, the amount of storage that is saved in using a sparse representation is not too significant. Second, since the DPS processors can perform a floating-point operation each cycle, redundant floating-point operations will not hinder performance significantly unless the sparsity is very great.

![Diagram of linked-list representation of vector elements](image)

**Figure 2-10:** Representation of List Based Vectors

The vector representation used in this example is a simple linked-list that does not provide support for sparsity. A vector will be represented as a linked-list. The head of each pair will point to an element in the vector, and the tail of each pair will point to the next pair. The final pair in the list will have a tail that
contains a null pointer, the number 0. This representation is depicted in Figure 2-10.

The following code is somewhat similar in structure to the code for the storage-vector based code. It is complicated by the fact that extra indirections are needed to get to the elements of the vectors, and by a deferred register update that helps to reduce the number of instructions in the routine.
When we start execution at #SAA _2, register 1 (REG1) will contain a pointer to \( a \) and register 2 (REG2) will contain a pointer to \( b \).
; Register 3 (REG3) will contain the address of the last written word in a free block of memory that can be used to allocate the new vector \( c \).
; This is being used in a pre-increment fashion. We will also cache a special pair template in register 4 (REG4). This special pair template will have a high word that points to the second free word after the last written address in REG3. The low word will point to the word just after that, which will be the location of the next pair. The pair template will be used to write out the newly allocated pairs of \( c \). Register 5 (REG5) will contain the incrementing constant for this special pair template. The constant increments by two addresses both in the high and low pointers.
; Register 0 (REG0) will contain the scaling constant, \( k \). Register 8 ;(REG8) contains a constant to mask out the upper word pointer of a pair.

;Start of scale-and-add (SAA) code. First, load the scaling constant \( k \) into a register in the floating-point unit.
#SAA_2: FLOATS <- REG0,

;Request first pair of \( b \).
MAR2 <- REG2,

;Now we receive the pair of \( b \) and load the CAR and CDR registers and update the current pointer to \( b \). Even though register 2 will now contain a pair with two 16-bit pointers, when it is used as a memory address, only the low word will be loaded into a MAR. Also, prepare for the read of the pair of \( a \).
SEL MAR2, REG2.CAR.CDR <- MEM, MAR1 <- REG1,

;Receive the element of \( b \) and store it in the floating-point unit.
#Loop: SEL CAR, FLOATR <- MEM,

;We start the scaling operation with a floating-point multiply operation. The result will be stored in the output register of the floating-point unit, FLOATF. Also, receive the pair of \( a \) and update the pointer.
SEL MAR1, REG1.CAR <- MEM, FLOATF <- FLOAT* FLOATR FLOATS,

;The element of \( a \) goes directly from memory to one input of the floating point unit. We load MAR1 with the address at which we will store the next pair of \( c \) and update the free pointer.
SEL CAR, FLOATR <- MEM, REG3.MAR1 <- REG3 +1

;Load the data register with the next pair template for the pair of \( c \) that we are allocating. Update the pair template. Now we do the add operation through an internal feedthrough path in the floating-point-processor.
REG4.MDR <- REG4 + REG5, FLOATF <- FLOAT+ FLOATR FLOATF

;#SAA _2 continued on next page
We now store the next pair of $\bar{\alpha}$. This new pair has a head that
points to the next free word of memory. We store the computed element
in the next cycles. The tail of the pair points to the next word after that,
which will contain the next pair of $\bar{\alpha}$ which will be stored in the next
iteration. If we are at the end of the list, then we will have to store a
null pointer for the tail. This is checked at the end of the loop, and the
fix up is performed at the exit point of the code. We also store the result
of the scale-and-add operation in a temporary register. We prepare for the
write of this result into $\bar{\alpha}$ by storing the next free address
into MAR1. We defer the update of the free pointer (REG3).

\begin{verbatim}
SEL MAR1, MEMWRITE, REG6 <- FLOATS, MAR1 <- REG3 +1
\end{verbatim}

The result is prepared to be written in the next free word of memory.

\begin{verbatim}
MDR <- REG6,
\end{verbatim}

Write the result and perform the deferred update of REG3.

\begin{verbatim}
SEL MAR1, MEMWRITE, REG3 <- REG3 +1
\end{verbatim}

We compare the current pair of $\bar{a}$ to check if we are at
the end of the list vector $\bar{a}$.

\begin{verbatim}
AND REG1 REG8, LATCHCC,
\end{verbatim}

Since the CDR register still contains the address of the next pair
of $\bar{a}$, we use this to get the next pair and load the CAR
and CDR registers and update the pointer to $\bar{b}$. We issue
the address of the next pair of $\bar{a}$ and store it in MAR1.
This puts us in a similar state as was at the beginning of the #Loop
instruction, so we jump there if we are not at the end of the list $\bar{a}$.

\begin{verbatim}
JMP NOT ZERO #LOOP, SEL CDR, REG2.CAR.CDR <- MEM,
MAR1 <- REG2,
\end{verbatim}

Exit code, which cleans up the last written pair of $\bar{a}$.
First issue the address of this last pair.

\begin{verbatim}
#CLEAN: MAR1 <- REG3 -1,
\end{verbatim}

This pair should have a head that points to the last written
element of $\bar{c}$, and a tail that is null. This is exactly register
$\bar{a}$, the pair pointer, with the tail bits masked to be zero.

\begin{verbatim}
MDR <- REG4 AND NOT REG8,
\end{verbatim}

Perform the update of the pair, and we are done.

\begin{verbatim}
SEL MAR1, MEMWRITE,
\end{verbatim}

End of #SAA-1 routine.

#Done:
This code has been very carefully hand coded, and it is somewhat confusing to follow. Nevertheless, it contains only 9 instructions in its inner loop. This is only 33% more instructions than the code for the storage vectors, and at a clock period of 170\textmu s, vector scale-and-add can be performed on these data structures at one element per 1.53\textmu s.

2.8.3 Microcode Conclusions

These two examples have attempted to demonstrate the flexibility of data representation that is possible in this Digital Parity Simulator. Performance is not terribly sacrificed when going from storage-vector to list representations because of the inexpensive memory enhancements which allow list tracing to be performed more efficiently than on conventional processors. These enhancements are somewhat tricky to handle in the coding of very compact machine code for the Digital Parity Simulator, but they allow very efficient system primitives to be implemented.

2.9 DPS Architecture Summary

In this chapter, the specific implementation of the Digital Parity Simulator has been examined. The motivations for its design were the need for high-performance floating-point operations with the ability to exploit simulation network parallelism through multiprocessing. A 32-bit design evolved from the available VLSI building blocks that could be used to implement the DPS processors. The processor design was described in detail, and each of the processor subsystems were examined. Of special interest were the architectural modifications in the memory system which provided support for linked-lists and the token-ring interprocessor connection network. Both these systems, as well as the somewhat non-traditional interconnection of the ALU’s in the processor, showed how the Digital Parity Simulator was more specialized than a general
purpose computer. Expected performance of the processors in the DPS was examined and microcode examples showed some of the benefits of the architecture for implementing various versions of the vector scale-and-add operation.

In the next chapter, the physical implementation of the Digital Parity Simulator will be examined. Conclusions will be drawn as how successful this architecture will be for the problems that it is intended to solve.
Chapter Three

Digital Parity Simulator Results

This chapter reports on the Digital Parity Simulator in terms of its physical implementation. These results are analyzed, and conclusions on the DPS system are drawn and explained. Finally, the current stage of this project is summarized and directions for further research are presented.

3.1 Digital Parity Simulator Construction

As stated in Chapter 2, the DPS system is implemented primarily with Advanced Schottky and VLSI components. Transients caused by the 3 ns or less switching edges in the Advanced Schottky parts can cause noise problems throughout the circuit. Also, VLSI components with large numbers of simultaneously changing outputs, such as the registered ALU and the floating-point processor, can create drastic power surges in the circuit. Quality printed-circuit boards can alleviate these problems. Multilayer printed-circuit boards with buried supply layers can reduce noise and facilitate the supply of power and ground to electronic components.

The DPS processors are built on 4 layer printed-circuit boards which are 12 by 17 inches in size. The cost and design time incurred by the use of printed circuit boards is justified by the fact that several processor boards are needed for the multiprocessor system. The next best connection technology would be wire-wrapping, but since each processor board contains 114 electronic components and several connectors and would take an incredible amount of time to wire-wrap by hand, wire-wrapping was rejected for this project. The printed circuit boards
were designed with the Circuit Board Design System (CBDS), which is described below.

3.1.1 CBDS

CBDS is a schematic-entry and printed-circuit board layout system developed by Bell-Northern Research. It runs on a 5080 graphics workstation connected to a 3720 IBM mainframe. The system allows the development of printed-circuit boards through four stages. First, circuit components that are not in the existing component database are defined in terms of schematic, logical, and physical features. Second, components are specified schematically and the necessary connections are designated. Third, the schematic is used as a base for layout. Automatic gate and pin assignment attempt to minimize interconnection length or crossing. Automatic and interactive component placement and automatic and interactive track routing are followed by automatic optimizations for reliability and manufacturability. Finally, the layout is used to drive pen-plotters to generate check plots. The layout also is used to generate Gerber photoplotter mask files and several types of drill-tape files. These files can be transferred to magnetic tape for use by outside manufacturers.

The system at first appears to be extremely powerful, especially for tutorial examples. The component database is fairly complete with standard TTL devices, and additions to the database are straightforwardly made, although the process is quite tedious for components with large numbers of pins. The schematic-entry package is excellent, except for the fact that it lacks the ability to express designs hierarchically. Designs are always specified with explicit components, and it is not possible to abstract multi-byte components out of single-byte components. The layout-package is the most unreliable package in the system, with several features that appear to corrupt designs, but many features are powerful. Pre-stored track patterns can be successfully used in
regular layout areas such as memory arrays and provide excellent component density. Five routing algorithms, ranging from very simple and fast to very complex and slow, are available. Typically, more and more complicated algorithms are used through the progression of the layout as the track congestion increases. Manual track routing is eventually necessary when the automatic routing algorithms can no longer operate due to the track congestion on the printed-circuit board. The manufacturing data generation package is easy to use, but it is extremely slow in generating layer mask files, and it appears to generate files that are not completely compatible with the Gerber photoplottter standards.

In reality, the system is far less than optimal for large designs such as the DPS processor boards. Although the layout package could route 95% of a 20 component tutorial example in about 2 hours, it could not do a very good job with the DPS processor board. The failures of the system are due to three reasons:

1. the system will not consider removing a track once it has been placed, and thus one bad route will cut off many possible good routes,

2. the DPS processors have a large component count, and

3. several of the components and connectors in the system contain an enormous number of pins.

Because of these problems, only about 40% of the board could be automatically routed. The remaining 60% was performed manually. The manual routing took at least 450 man-hours of effort and was only possible by routing about 100 signals in the buried layer reserved for the power plane. This compromises the effectiveness of the power plane role in decoupling signals from power and ground surges.
3.2 System Results

As of the time of this writing, the mask flies have been successfully used to generate the layer mask transparencies which will be used to manufacture the DPS processor boards. Once the boards are manufactured and inspected, the various subsystems in a DPS processor will be tested and exercised. Subject to component availability, it is anticipated that a two-processor system will be operational in a few months. At that time, distributed algorithms to solve circuit simulations will be tested on the system, and reports on that work will be forthcoming.

3.3 Conclusions

There are three types of conclusions that can be drawn from the work reported in this thesis. These are reported below.

3.3.1 DPS System Conclusions

Even before the DPS system has its hardware fully tested, there are several important conclusions that can be drawn. The first of these is the significance of this architecture for solving circuit problems. Although very little DPS software has been written, it is obvious from the microcoded examples in section 2.8 that software for the DPS processors will be very difficult and tedious to write. This may be relieved if the software can rely on carefully coded, highly efficient system primitives. If this is the case, then simulation software can consist of relatively inefficient and easy to code routines that "glue together" very efficient primitive routines. In any case, it remains to be seen whether or not a user friendly environment can be implemented on this Digital Parity Simulator system.

The second conclusion that can be drawn is that the problem of performing
circuit simulations quickly motivates the exploitation of parallelism at two levels. First, the DPS processors exploit fined-grained parallelism in an simplified array processor fashion. The non-standard processing-unit interconnection and the wide microcode word exist to allow the simultaneous operations of several functional units. This allows the efficient implementation of functions such as the vector-scale-and-add operation. Second, the DPS system exploits coarse-grained parallelism in a multiprocessor fashion. The DPS processors and the high-speed ring interconnection network exist to allow the distributed simulation of several relatively loosely coupled networks. Another way to look at this is that the fined-grained parallelism attempts to optimize numerical aspects of the circuit simulation, while the coarse-grained parallelism attempts to optimize the parity aspects of the circuit simulation.

3.3.2 Directions for Further Work
The most important work that must be performed once the DPS hardware is fully implemented is the creation of system software. Crucial in this task is the creation of highly efficient microcode primitives.

As various primitives are tried and tested, it may become feasible to reduce the cost of the system and increase performance by storing primitives in read-only-memory (ROM).

It has already been stated that the system could perform much more accurately if it could operate on 64-bit floating-point numbers. As 64-bit parts become more available, the system should be modified to take advantage of these.
3.3.3 Project Conclusions

There are several things that should have been done differently in the course of this project. Chief of these is the selection of electronic components. Although it was assured that the parts would be available much sooner, the Texas Instruments SN74AS8832 Registered ALU’s are still not in production at the time of this writing. Current plans call for the use of defective beta-test components to help expedite system testing. Regardless of how soon these parts do become available, one thing is absolutely clear: designing with non-existent components is very risky. This designer will never again take this type of risk!

The fact that the ultimate design of the DPS processors is so large indicates that there may be some basic flaw in this approach to the problems of circuit simulation. The processors are very complicated and allow for a great deal of local parallelism, but processor boards of this complexity are difficult to design, lay out, and manufacture. It may be possible that smaller boards which contain more decoding and less parallelism could be more effective if used in greater numbers. It is unclear how much smaller the processors could be, however, and it has already been proven in the case of Sterling’s research that microprocessors are not good as building-blocks. Perhaps the solution lies in the use of custom VLSI.

Another part of the project that should be questioned is the feasibility of designing a full-fledged simulation facility from scratch. This has certainly turned out to be a much more enormous task than previously anticipated, and the project is not near completion in any sense. The Digital Parity Simulator architecture appears to be unique in its attempt to exploit both local and global parallelism expressly for circuit simulation. Nevertheless, attempts at enhancing existing multiprocessors to support faster floating-point calculations or attempts at enhancing existing array processors to exploit multiprocessing may be more profitable in the long run in terms of cost and performance.
In conclusion, this attempt at an architectural description of a digital parity simulator is only a small step in the quest for increased circuit simulation performance. As technology advances, more challenging architectures for circuit simulation will be proposed and tested. It is the author's hope that some of the ideas presented in this architecture will motivate the design of future machines with unimaginable performance.
References

*Structure and Interpretation of Computer Programs.*  

*The Design of a Multiprocessor Development System.*  
Technical Report TR-279, M.I.T. Laboratory for Computer Science,  
Cambridge, Massachusetts, September, 1982.

A Digital Orrey.  

[4] Arvind and Culler, D.  
*Dataflow Architectures.*  
Technical Report TM-294, M.I.T. Laboratory for Computer Science,  
Cambridge, Massachusetts, February, 1986.

[5] Arvind and Iannucci, R.  
*Two Fundamental Issues in Multiprocessing.*  
Laboratory for Computer Science, Cambridge, Massachusetts, July,  
1986.

Technical Report 5284, Bolt, Beranek, and Newman Incorporated,  
Cambridge, Massachusetts, April, 1983.

[7] Russel, R. M.  
The CRAY-1 Computer System.  
*Communications of the Association of Computer Machinery 21*(1),  

[8] Crowther, W., Goodhue, J., Starr, R., Milliken, W., Blackadar, T.  
Performance Measurements on a 128-Node Butterfly™ Parallel Processor.  
1986.  
Internal Bolt, Beranek, and Newman Laboratories Paper.
[9] Deutsch, J. and Newton, R.
A Multiprocessor Implementation of Relaxation-Based Electrical Circuit Simulation.

An Overview of the VAX 8600 System.

The *Connection Machine (Computer Architecture for the New Wave).*
Technical Report AI Memo 646, M.I.T. Artificial Intelligence Laboratory,
Cambridge, Massachusetts, September, 1981.

*The Multibus Design Guidebook.*

Parity simulation of static power conversion systems.
1977.


CRAY-1 Computer Technology.
*IEE Transactions on Components, Hybrids, and Manufacturing Technology* CHMT 4(2), June, 1981.

*Chrysalis™ Programmers Manual, Version 2.2.*

Parallel Computer Processing for Power Electronic Network Simulation.
_ALS/AS Logic Data Book_.  

[19] Traub, K.  
A Compiler for the MIT Tagged-Token Dataflow Architecture.  
Master's thesis, Department of Electrical Engineering and Computer  
Science, Massachusetts Institute of Technology, August, 1986.

[20] Troiani, M., et. al..  
The VAX 8600 I Box, A Pipelined Implementation of the VAX  
Architecture.  

Scheme86—An Architecture for Microcoding a Scheme Interpreter.  
May, 1986.  
Undergraduate thesis, Department of Electrical Engineering and Computer  
Science, Massachusetts Institute of Technology.
Appendix A

DPS Instruction Set

The instruction word is 96 bits wide and is divided as in the following figure.

**Figure 1:** Microinstruction Word

The interpretation of the fields in the microinstruction word is according the following table. Additional tables show the interpretation of the multi-bit fields that are not direct inputs to the VLSI components. See the documentation on the SN74AS890 MicroSequencer, SN74AS8832 Registered ALU, and the Am29325 Floating-Point Processor for the interpretation of their control fields.
<table>
<thead>
<tr>
<th>Field</th>
<th>Width</th>
<th>Interpretation</th>
</tr>
</thead>
<tbody>
<tr>
<td>Branch/Constant</td>
<td>16</td>
<td>Field for Branch Addresses or Data Path Constants.</td>
</tr>
<tr>
<td>uMux</td>
<td>3</td>
<td>Controls from what source the Microsequencer delivers an address.</td>
</tr>
<tr>
<td>Stack</td>
<td>3</td>
<td>Controls the operation of the Microsequencer stack.</td>
</tr>
<tr>
<td>RC</td>
<td>3</td>
<td>Controls the loading and decrementing of the Microsequencer registers.</td>
</tr>
<tr>
<td>/Interrupt!</td>
<td>1</td>
<td>Asserts the tri-state interrupt line.</td>
</tr>
<tr>
<td>/IntEn</td>
<td>1</td>
<td>Enables local interrupts.</td>
</tr>
<tr>
<td>CCSel</td>
<td>5</td>
<td>Selects the condition code that is fed into the Microsequencer.</td>
</tr>
<tr>
<td>CCIInv</td>
<td>1</td>
<td>Inverts the condition code.</td>
</tr>
<tr>
<td>OSel</td>
<td>1</td>
<td>Selects the output of the DRA bus on the Microsequencer.</td>
</tr>
<tr>
<td>ASel</td>
<td>2</td>
<td>Selects which device drives the system ABus.</td>
</tr>
<tr>
<td>BSel</td>
<td>2</td>
<td>Selects which device drives the system BBus.</td>
</tr>
<tr>
<td>AddrSel</td>
<td>2</td>
<td>Selects which MAR drives the Data Memory Address input.</td>
</tr>
<tr>
<td>/LD</td>
<td>6</td>
<td>Selectively loads the MAR1, MAR2, CAR, CDR, MDR and Output registers.</td>
</tr>
<tr>
<td>RFS</td>
<td>2</td>
<td>Selects the source of Register File stores in the Registered ALU.</td>
</tr>
<tr>
<td>ALUI</td>
<td>8</td>
<td>Selects the ALU instruction.</td>
</tr>
<tr>
<td>Field</td>
<td>Width</td>
<td>Interpretation</td>
</tr>
<tr>
<td>----------</td>
<td>-------</td>
<td>----------------------------------------------------</td>
</tr>
<tr>
<td>ALUASel</td>
<td>6</td>
<td>Selects the A source register.</td>
</tr>
<tr>
<td>ALUBSel</td>
<td>6</td>
<td>Selects the B source register.</td>
</tr>
<tr>
<td>ALUCSel</td>
<td>6</td>
<td>Selects the C destination register for the ALU.</td>
</tr>
<tr>
<td>/ALUWE</td>
<td>1</td>
<td>Selects a register write.</td>
</tr>
<tr>
<td>ALUEA</td>
<td>1</td>
<td>Selects the source for the A input of the ALU.</td>
</tr>
<tr>
<td>ALUEB</td>
<td>2</td>
<td>Selects the source for the B input of the ALU.</td>
</tr>
<tr>
<td>CIn</td>
<td>1</td>
<td>Forces a carry input into the ALU.</td>
</tr>
<tr>
<td>SelMQ</td>
<td>1</td>
<td>Selects the source for the output port of the Registered ALU.</td>
</tr>
<tr>
<td>/ENSRF</td>
<td>3</td>
<td>Selects the loading of the Floating point registers.</td>
</tr>
<tr>
<td>FloatI</td>
<td>5</td>
<td>Selects the floating point instruction.</td>
</tr>
<tr>
<td>/Local!</td>
<td>1</td>
<td>Asserts the tri-state local transfer line.</td>
</tr>
<tr>
<td>Need</td>
<td>1</td>
<td>Asserts the need for multiprocessor communication.</td>
</tr>
<tr>
<td>Pass</td>
<td>1</td>
<td>Asserts the finish of a multiprocessor send cycle.</td>
</tr>
<tr>
<td>Trans</td>
<td>1</td>
<td>Puts memory into transparent mode.</td>
</tr>
<tr>
<td>Token</td>
<td>2</td>
<td>Creates and absorbs tokens.</td>
</tr>
<tr>
<td>MemW</td>
<td>1</td>
<td>Asserts a memory write.</td>
</tr>
<tr>
<td>LatchCC</td>
<td>1</td>
<td>Latches the current condition codes.</td>
</tr>
<tr>
<td>Bits</td>
<td>Selected Condition Code</td>
<td></td>
</tr>
<tr>
<td>---------</td>
<td>-------------------------------------------------</td>
<td></td>
</tr>
<tr>
<td>0 0 0 0 0</td>
<td>Always 0</td>
<td></td>
</tr>
<tr>
<td>0 0 0 0 1</td>
<td>Always 1</td>
<td></td>
</tr>
<tr>
<td>0 0 0 1 0</td>
<td>Zero Flag from ALU</td>
<td></td>
</tr>
<tr>
<td>0 0 0 1 1</td>
<td>Neg Flag from ALU</td>
<td></td>
</tr>
<tr>
<td>0 0 1 0 0</td>
<td>Carry Flag from ALU</td>
<td></td>
</tr>
<tr>
<td>0 0 1 0 1</td>
<td>Over Flag from ALU</td>
<td></td>
</tr>
<tr>
<td>0 0 1 1 0</td>
<td>Zero Flag from Microsequencer</td>
<td></td>
</tr>
<tr>
<td>0 0 1 1 1</td>
<td>Stack Warning Flag from Microsequencer</td>
<td></td>
</tr>
<tr>
<td>0 1 0 0 0</td>
<td>Zero Flag from Floating-Point Unit</td>
<td></td>
</tr>
<tr>
<td>0 1 0 0 1</td>
<td>Sign Bit from Floating-Point Output (High bit from ABus)</td>
<td></td>
</tr>
<tr>
<td>0 1 0 1 0</td>
<td>Underflow Flag from Floating-Point Unit</td>
<td></td>
</tr>
<tr>
<td>0 1 0 1 1</td>
<td>Overflow Flag from Floating-Point Unit</td>
<td></td>
</tr>
<tr>
<td>0 1 1 0 0</td>
<td>Inexact Flag from Floating-Point Unit</td>
<td></td>
</tr>
<tr>
<td>0 1 1 0 1</td>
<td>Invalid Flag from Floating-Point Unit</td>
<td></td>
</tr>
<tr>
<td>0 1 1 1 0</td>
<td>Not a Number Flag from Floating-Point Unit</td>
<td></td>
</tr>
<tr>
<td>0 1 1 1 1</td>
<td>Not Used</td>
<td></td>
</tr>
</tbody>
</table>
**Table 2: CCSel Table, continued**

<table>
<thead>
<tr>
<th>Bits</th>
<th>Selected Condition Code</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 0 0 0 0</td>
<td>Pass signal from previous processor</td>
</tr>
<tr>
<td>1 0 0 0 1</td>
<td>Need signal from previous processor</td>
</tr>
<tr>
<td>1 0 0 1 0</td>
<td>Token Valid Flag</td>
</tr>
<tr>
<td>1 0 0 1 1</td>
<td>/Local Communication Flag Status</td>
</tr>
<tr>
<td>1 0 1 0 0</td>
<td>Input Valid Flag</td>
</tr>
<tr>
<td>1 0 1 0 1</td>
<td>Output Valid Flag</td>
</tr>
<tr>
<td>1 0 1 1 0</td>
<td>/Flag indicating Host is still reading Output data</td>
</tr>
<tr>
<td>1 0 1 1 1</td>
<td>/Global Interrupt Status</td>
</tr>
</tbody>
</table>
### Table 3: ASel Table

<table>
<thead>
<tr>
<th>Bits</th>
<th>Device that Drives the ABus</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 0</td>
<td>Microsequencer DRA port</td>
</tr>
<tr>
<td>0 1</td>
<td>Registered ALU DA port</td>
</tr>
<tr>
<td>1 0</td>
<td>Input Register</td>
</tr>
<tr>
<td>1 1</td>
<td>Floating-Point Y Output port</td>
</tr>
</tbody>
</table>

### Table 4: BSel Table

<table>
<thead>
<tr>
<th>Bits</th>
<th>Device that Drives the ABus</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 0</td>
<td>Registered ALU DB port</td>
</tr>
<tr>
<td>0 1</td>
<td>MDR</td>
</tr>
<tr>
<td>1 0</td>
<td>Data Memory</td>
</tr>
<tr>
<td>1 1</td>
<td>Constant Buffer</td>
</tr>
</tbody>
</table>
### Table 5: AddrSel Table

<table>
<thead>
<tr>
<th>Bits</th>
<th>Device that Drives the Data Memory Input [Only valid if Memory is not in transparent mode]</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 0</td>
<td>MAR1</td>
</tr>
<tr>
<td>0 1</td>
<td>MAR2</td>
</tr>
<tr>
<td>1 0</td>
<td>CAR</td>
</tr>
<tr>
<td>1 1</td>
<td>CDR</td>
</tr>
</tbody>
</table>

### Table 6: /LD Table

<table>
<thead>
<tr>
<th>Bits</th>
<th>Register that Loads</th>
</tr>
</thead>
<tbody>
<tr>
<td>x x x x x 0</td>
<td>Output Register</td>
</tr>
<tr>
<td>x x x 0 x x</td>
<td>MDR</td>
</tr>
<tr>
<td>x x x 0 x x</td>
<td>CDR</td>
</tr>
<tr>
<td>x x 0 x x x</td>
<td>CAR</td>
</tr>
<tr>
<td>x 0 x x x x</td>
<td>MAR2</td>
</tr>
<tr>
<td>0 x x x x x</td>
<td>MAR1</td>
</tr>
</tbody>
</table>
### Table 7: ENSRF Table

<table>
<thead>
<tr>
<th>Bits</th>
<th>Floating-Point Register that Loads</th>
</tr>
</thead>
<tbody>
<tr>
<td>x x 0</td>
<td>F Output Register</td>
</tr>
<tr>
<td>x 0 x</td>
<td>R Input Register</td>
</tr>
<tr>
<td>0 x x</td>
<td>S Input Register</td>
</tr>
</tbody>
</table>

### Table 8: Token Table

<table>
<thead>
<tr>
<th>Bits</th>
<th>Token Command</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 0</td>
<td>Pass Tokens</td>
</tr>
<tr>
<td>0 1</td>
<td>Absorb Tokens</td>
</tr>
<tr>
<td>1 0</td>
<td>Create Tokens</td>
</tr>
<tr>
<td>1 1</td>
<td>Undefined</td>
</tr>
</tbody>
</table>
Appendix B

Host Processor Commands

The DPS system responds to 16-bit I/O address from the host processor. Bits 1-4 asserts a command, whose interpretation is as in the following table.
<table>
<thead>
<tr>
<th>A4 A3 A2 A1</th>
<th>Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 0 0 0</td>
<td>Load word 0 of instruction latch.</td>
</tr>
<tr>
<td>0 0 0 1</td>
<td>Load word 1 of instruction latch.</td>
</tr>
<tr>
<td>0 0 1 0</td>
<td>Load word 2 of instruction latch.</td>
</tr>
<tr>
<td>0 0 1 1</td>
<td>Load word 3 of instruction latch.</td>
</tr>
<tr>
<td>0 1 0 0</td>
<td>Load word 4 of instruction latch.</td>
</tr>
<tr>
<td>0 1 0 1</td>
<td>Load word 5 of instruction latch.</td>
</tr>
<tr>
<td>0 1 1 0</td>
<td>Load Input Register.</td>
</tr>
<tr>
<td>0 1 1 1</td>
<td>Load Dispatch Latch.</td>
</tr>
<tr>
<td>1 0 0 0</td>
<td>Force Dispatch Latch as Microcode Address.</td>
</tr>
<tr>
<td>1 0 0 1</td>
<td>Force Dispatch Latch and Load Instruction into Memory.</td>
</tr>
<tr>
<td>1 0 1 x</td>
<td>Interrupt DPS processor.</td>
</tr>
<tr>
<td>1 1 x x</td>
<td>Read Output Register.</td>
</tr>
</tbody>
</table>