PROGRAMMING FINE-GRAINED RECONFIGURABLE ARCHITECTURES

APPROVED BY
DISSERTATION COMMITTEE:

Supervisor: ____

To my family

PROGRAMMING FINE-GRAINED RECONFIGURABLE ARCHITECTURES

by

STEVEN ANTHONY GUCCIONE, B.S.E.E., M.S.E.E.

DISSERTATION

Presented to the Faculty of the Graduate School of

The University of Texas at Austin

in Partial Fulfillment

of the Requirements

for the Degree of

DOCTOR OF PHILOSOPHY

THE UNIVERSITY OF TEXAS AT AUSTIN

May 1995

Acknowledgments

First and foremost, I would like to thank Dr. Gonzalez for his unflagging support throughout this research. This research would not have been possible without his guidance and good advice.

I would also like to thank the other committee members, Professors Swartzlander, Cragon, Szygenda, and Jenevein. All gave generously of their time and each contributed, at some time, in steering me toward more fruitful areas of research, or away from the rocks.

Also I would like to thank the other researchers in this area, particularly Eric Dellinger, Martin Poenie, Tom Kean and Jeff Arnold for the interesting and ongoing exchange of ideas.

Finally, I would like to thank Craig Fowler of Texas Instruments, who set me on the path toward this degree.

STEVEN ANTHONY GUCCIONE

The University of Texas at Austin

May 1995

PROGRAMMING FINE-GRAINED RECONFIGURABLE ARCHITECTURES

Publication No. ____________

Steven Anthony Guccione, Ph.D.
The University of Texas at Austin, 1995

Supervisor: Mario J. Gonzalez, Jr.

Recently, several systems based on reconfigurable logic have been designed and built. These systems permit arbitrary digital logic functions to be configured in hardware. This ability to dynamically configure such circuits promises to provide the flexibility of a software based system with the performance of custom hardware.

This dissertation proposes a high level language approach to programming reconfigurable logic based machines. This approach uses a data parallel variant of the *C* language. To support this high level language approach, a novel reconfigurable logic device is described. These devices are in turn interfaced to a memory system and used to perform computation.

To demonstrate the validity of this approach, several computationally intensive algorithms are implemented and simulated. Among these algorithms are cellular automata, image processing, neural networks, the Mandelbrot set and the Fourier transform. In addition, selected portions of the Livermore FORTRAN kernels are simulated. Estimates of performance and required system resources are reported for each algorithm.

Table of Contents

List of Tables

List of Figures

Chapter 1

Introduction

Traditionally, there have been two major methods for implementing algorithms. Most commonly, an algorithm is coded in software and implemented on a general purpose processor. In other cases, custom hardware is designed and built to implement the given algorithm. The custom hardware solution is usually characterized by its high performance, but relatively high cost and inflexibility. The software solution, on the other hand, has typically been characterized by its high degree of flexibility, but at a lower cost and lower performance. The choice of creating a custom hardware solution to a problem as opposed to a software-based solution has traditionally been based on cost and performance versus flexibility.

A relatively new technology, reconfigurable logic, provides a new way to implement algorithms that promises to change this traditional tradeoff between cost, performance and flexibility. Reconfigurable logic permits custom digital circuits to be dynamically created and modified via software. This ability to create and modify digital logic without physically altering the hardware provides a more flexible and lower cost solution to the implementation of custom hardware. This represents a significant divergence from the traditional hardware / software tradeoff.

Reconfigurable logic devices were first introduced commercially in the mid-1980s. These initial devices had a relatively low density and could be used to create custom circuits of several hundred equivalent logic gates. These early devices were used primarily by hardware designers to group random logic into a single package. This approach to logic design has two advantages. First, the combining of several integrated circuits into a single package reduces the size and cost of the circuit board. Perhaps more important, however, is the ability to modify the hardware design with a simple software change. Because the function of the reconfigurable logic device is defined by software, design errors can be corrected without having to fabricate new hardware. Existing system hardware may also be modified and upgraded without any physical modifications. Only a change to the software used by the reconfigurable logic device is required.

As the density of these devices has increased, it has became apparent that entire systems can be constructed using reconfigurable logic. This type of architecture can provide a flexible approach to producing customizable hardware. Several such systems based on reconfigurable logic have been designed and implemented. All have been used successfully to accelerate selected algorithms. Larger systems have reported performance surpassing that of conventional supercomputers while using modest amounts of hardware running at relatively low clock speeds.

While impressive levels of performance have been reported, none of these architectures are poised to challenge more traditional high-performance systems. Currently, the main limitation in the use of systems based on reconfigurable logic is software. The task of programming these systems is typically based on hardware design rather than traditional high level language programming.

It is the goal of the research described in this dissertation to describe a system which:

- makes large scale use of reconfigurable logic
- achieves supercomputer levels of performance
- is programmable using traditional high level programming languages

Chapter 2 begins with a short history of programmable logic. The uses of this technology, particularly reconfigurable logic, is outlined. Special emphasis is given to machines which use reconfigurable logic to perform general purpose computations. Other related work is also examined.

Chapter 3 explores some of the issues involved in the large scale use of reconfigurable logic. Of particular interest is the high overhead, both in terms of hardware and software, associated with the use of reconfigurable logic. Conclusions based on these architectural features are drawn.

Chapter 4 proposes a parallel programming methodology based on the computational characteristics of reconfigurable logic. Compilation and optimization techniques are presented. Finally, a novel construct for providing data sequencing is presented.

Chapter 5 examines existing reconfigurable logic devices and proposes a novel microarchitecture geared toward high-level computation. This microarchitecture directly supports the software methodology proposed in Chapter 4.

Chapter 6 discusses the system level architecture of a machine based on reconfigurable logic. This architecture is designed specifically to make use of the programming methodology described in Chapter 4 as well as the microarchitecture described in Chapter 5.

Chapter 7 describes the implementation of selected algorithms. These algorithms represent a set of popular computationally intensive algorithms typically executed on parallel or vector supercomputers. A discussion of which sorts of algorithms are not appropriate for this architecture is also presented.

Chapter 8 discusses the suitability of fine-grained reconfigurable architectures as a high-performance computing medium. Possible future directions for this technology are mentioned.

Chapter 2

Background and Related Work

The first part of this chapter introduces programmable logic. Programmable logic may be defined as a digital logic circuit whose behavior is defined by software. By writing the appropriate data patterns to a programmable logic device, logic circuits may be implemented or modified. The evolution of these devices, from early programmable logic devices to modern reconfigurable logic devices is briefly traced.

Programmable logic devices were initially used on a small scale, primarily to simplify the hardware design process. As denser programmable logic devices have become available, it has become feasible to build entire systems based on this technology. The second part of this chapter is concerned with systems constructed using a particular type of programmable logic known as reconfigurable logic. Several interesting approaches, achieving various levels of performance in various application areas, are discussed. Finally, other related work is presented.

2.1 Programmable Logic

The term *programmable logic* is the most general term used to describe any device which can be customized to produce a specific logic function. There are two major types of programmable logic. The first is *write-once* programmable logic. This type of logic can be programmed only once, usually before it is installed in the system. The particular logic device, once programmed, typically cannot be reprogrammed.

A second type of programmable logic, often referred to as *reconfigurable logic*, may be reprogrammed repeatedly. These devices are unique in that they may be modified while in-system. It is these devices which are primarily of interest in fine-grained reconfigurable architectures. The write-once style of devices are discussed briefly, primarily to place reconfigurable logic in a larger context.

2.1.1 Memory Devices

The earliest, and most general form of programmable logic device is memory. While these devices are generally associated with data storage, they may also be viewed as programmable circuits for implementing arbitrary logic functions.

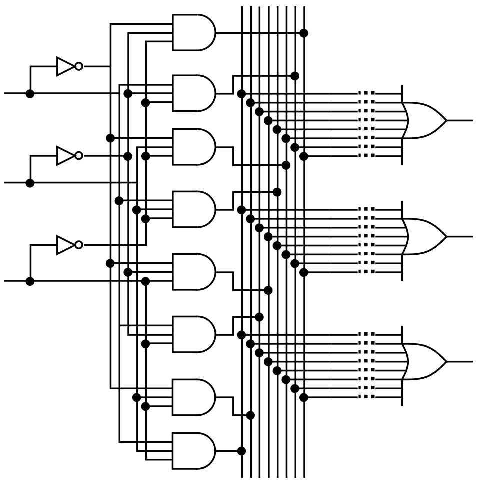


Figure 2.1: A ROM circuit.

Memory devices may be either programmable or reconfigurable. The one-time programmable devices are typically referred to as *Read Only Memory*, or *ROM*. These devices are usually implemented as two levels of logic, an AND array and an OR array. All possible 2^N input bit patterns are decoded by the AND array. These are then selectively ORed to produce the outputs, as shown in Fig 2.1. With this scheme, a standard hardware module can easily be programmed to produce any possible boolean function of the inputs.

While these devices can typically only be programmed once, the function of the hardware can still be modified by replacing the ROM device with another device containing a new function. This swapping of a single device is typically much simpler than other hardware modifications.

A device similar to the ROM is the *Random Access Memory*, or *RAM*. This device also permits any arbitrary boolean function of N inputs to be implemented in hardware. Rather than having to physically replace the device to alter the function of the hardware, RAM device may be dynamically reprogrammed. By writing the proper bit patterns to the device, any output function of the N input functions may be generated.

While the most flexible solution to customizable hardware, RAM and ROM devices have a hardware complexity proportional to 2^N. For even moderately sized functions, the amount of hardware, as well as the time necessary to write the necessary data to program the devices, becomes prohibitively large.

As an example, a boolean function of 32 inputs and 32 outputs would require 16×10^9 bytes of memory. Aside from being a physically large amount of memory, writing the bit pattern necessary to define the desired boolean function would take nearly three minutes at a rate of 100 megabytes per second. If a 64 bit function is desired, the situation is much worse. Approximately 10^{10} gigabytes would be necessary to define the boolean function. Even if such an enormous amount of RAM or ROM existed, it would take on the order of thousands of years to program, even at the rate of 100 megabytes per second.

2.1.2 Programmable Arrays

While RAM and ROM devices are capable of implementing any arbitrary boolean function, they are prohibitively expensive for implementing all but the smallest functions. A more restricted programmable logic device similar in structure to the ROM is the *Programmable Logic Array*, or *PLA*.

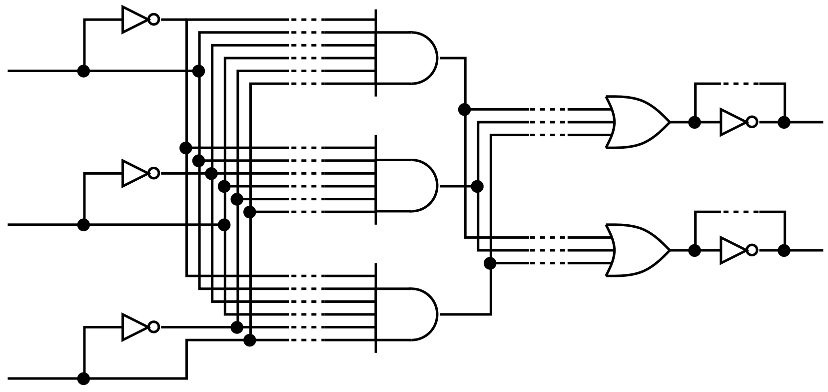


Figure 2.2: A PLA with 3 inputs, 3 product terms and 2 outputs.

These devices do not attempt to decode all possible 2^N input patterns. While structured as an AND-OR array like the ROM, these devices provide the ability to define the inputs to the AND array, as well as to the OR array. While the number of boolean functions which can be expressed by this structure is reduced, it has been found that for most practical applications, this simple structure is adequate.

While this type of programmable logic has only been used on a small scale, it has served as as the basis more much of the later work in programmable and reconfigurable logic. Early work at IBM [34, 80] as well as that of Patil and Welch [102] cover this area in more detail.

2.1.3 Cellular Arrays

Whereas ROMs and PLAs use two levels of fixed logic and a programmable interconnection structure to permit the definition of boolean functions, the reverse approach was taken by cellular arrays. Rather than fixing the logic and programming the interconnections, cellular arrays instead consist of an array of programmable logic cells joined by fixed interconnects. Cellular arrays were an active area of research in the late 1960s. Most prominent was the work done at the Stanford Research Institute [129]. Research was also performed by Kautz [62, 61] and by Minnick [92, 93]. In 1970, Shoup published his dissertation on the subject [114]. After this, very little is found in the literature relating to cellular arrays for several years.

In the mid-1980s, there was a resurgence in interest in cellular architectures. This lull of a decade and a half is somewhat inexplicable. Perhaps the best explanation for the resurgence during the mid-1980s is the increasing density of silicon devices. At this time, it became feasible to produce moderately sized cellular arrays.

The first commercial offering of a fixed interconnect cellular array was the Algotronix *Cellular Array Logic* or *CAL* device [63, 65, 64]. This device used an interconnection scheme described by Minnick nearly a quarter of a century earlier [92] and explored by Shoup [114] in his dissertation. The structure of

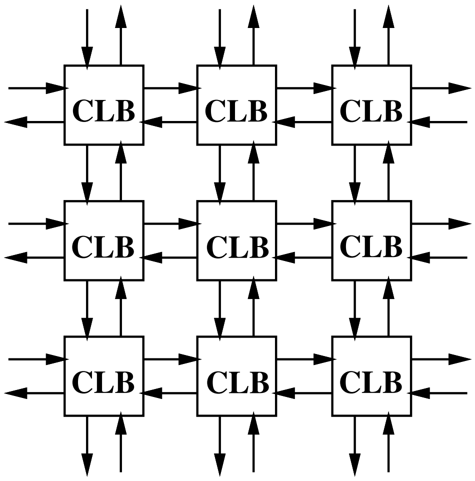


Figure 2.3: The Algotronix CAL architecture.

this two-dimensional array is shown in Figure 2.3.

Unlike ROM and PLA style programmable logic devices, this cellular approach permits arbitrary circuits using multiple levels of logic to be constructed. The cell of the device, called a *Configurable Logic Block*, or *CLB*, can perform any boolean operation of two inputs. Other inputs and outputs in the cell can be used to transfer signals to other CLBs. Since the cells are based on RAM technology, they may be dynamically reconfigured. A simple diagram of the CAL CLB is shown in Figure 2.4.

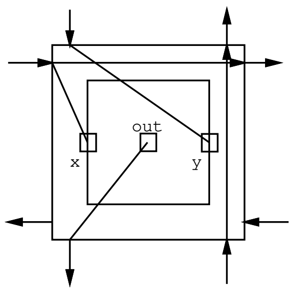


Figure 2.4: The Algotronix CAL logic cell.

2.1.4 Programmable Interconnections

Where early programmable logic devices use fixed function logic with programmable interconnect, and cellular arrays use programmable logic with fixed interconnect, a novel reconfigurable logic device which combines these two features was commercially introduced by Xilinx in 1985 [17, 133]. This device was known as the *Field Programmable Gate Array*, or *FPGA*. The flexibility of this device made it immediately popular with designers of custom hardware.

The architecture of the Xilinx device is shown in Figure 2.5. The interconnection network consists of groups of wires routed between the logic cells. The groups of wires intersect at junction points containing a programmable switch. This switch permits signals to be routed in arbitrary paths to distant logic blocks.

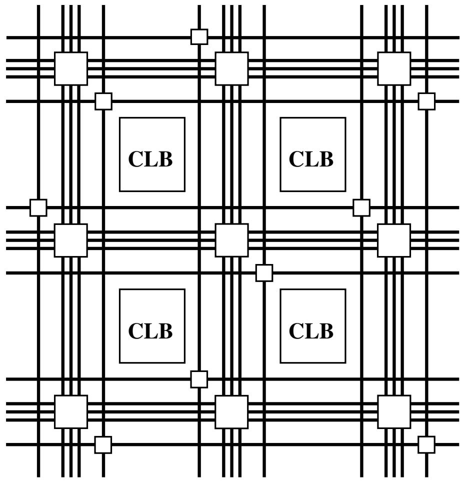


Figure 2.5: The Xilinx architecture.

In contrast to the cellular architectures, the Xilinx architecture features a more complex logic cell, as shown in Figure 2.6. A function of five inputs and two outputs can be configured. Other options such as internal feedback paths and registered outputs further increase the complexity of the cell. In more recent devices by Xilinx, this approach has been pursued further, with even larger configurable logic cells being employed.

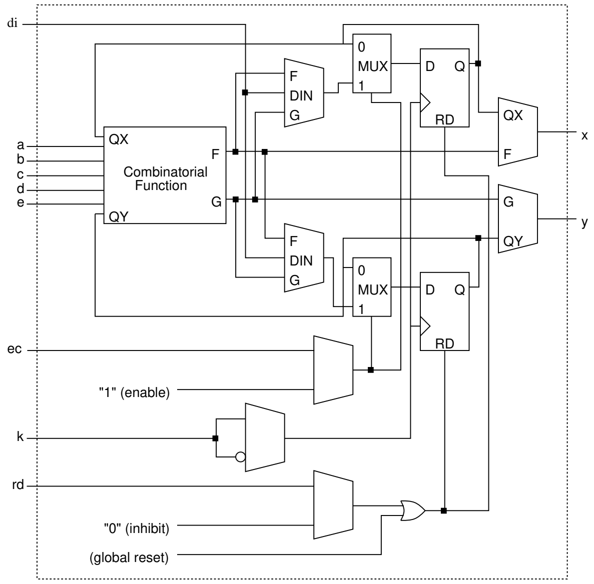


Figure 2.6: The Xilinx CLB.

While a close relative to cellular architectures, the grid of programmable interconnections give the Xilinx architecture a conceptual advantage. The separation of the logic cells and their interconnections more closely resembles the traditional hardware design situation. It is not necessarily clear, however, that the use of such programmable interconnect provides any material advantage over other cellular architectures with fixed interconnection networks.

2.2 Fine Grained Reconfigurable Architectures

Much of the original motivation for developing and using reconfigurable logic was to simplify the hardware design process. The ability to quickly modify a digital logic circuit without having to physically modify the hardware gave designers a new level of flexibility.

The extension of this concept leads to a system where all of the hardware is reconfigurable. Such hardware should be extremely flexible. Since the system can implement arbitrary digital circuits, virtually any custom logic function can be configured into the system. Such a system should provide the flexibility and programmability usually associated with an instruction set processor, but with the performance approaching that of custom hardware.

Figure 2.7 shows the basic components of a system based on reconfigurable logic. The reconfigurable logic in this system, referred to here as the *Reconfigurable Processing Unit* or *RPU*, contains one or more reconfigurable logic devices. The RPU is interfaced to a memory system which may be coupled to the RPU in a number of ways. Finally, a host processor is used for a variety of tasks, including reconfiguration of the RPU.

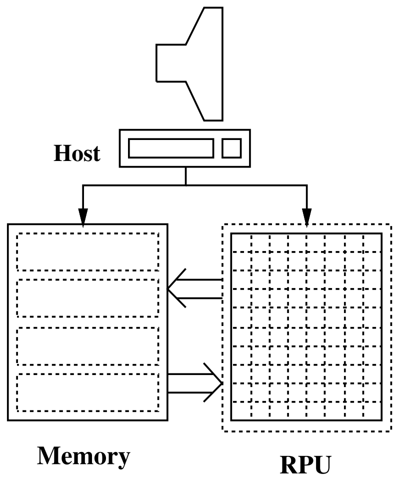


Figure 2.7: A reconfigurable system.

The term introduced here for such a system is a *fine grained reconfig-*

urable architecture. The appellation *fine grained* is used to distinguish these systems from other types of systems which may employ reconfiguration in other ways or for other purposes [76].

2.2.1 Early Work

The idea of systems using easily modifiable hardware was explored by several early researchers. One notable effort was the *Fixed-Plus-Variable* machine of Estrin [28, 29]. This machine attempted to make use of simple, standard hardware modules to accelerate software applications. While this machine did rely on manual reconfiguration, it embodied many of the concepts used by later software based reconfigurable machines. Several applications were implemented, including matrix computation [31]. Perhaps more importantly, this work also involved a substantial software effort. The ability to translate code into circuits was an early innovation of this system [30]. Much of this work influenced researchers in custom microcode, and would be cited as the predecessor of later reconfigurable logic based machines.

Shortly after this effort, research in cellular arrays began, but no substantial effort was made to build a programmable system using this technology. In the mid-1980s, interest in large scale use of reconfigurable logic to build reconfigurable machines increased. It is likely that this interest was fueled by the introduction of commercial reconfigurable logic devices from Xilinx [17, 133] and others. These devices, however, were initially quite small, providing the equivalent of only a few hundred logic gates. As the density of reconfigurable devices increased, it became apparent to several independent groups of researchers that complete systems based only on reconfigurable logic could be successfully constructed.

By 1990, a small number of research projects were constructing reconfigurable machines and reporting encouraging results. These early machines included the work at the Paris Research Laboratory of the Digital Equipment Corporation [9, 10, 113, 112], the Supercomputer Research Center [40, 41] and the work at Kean at the University of Edinburgh [63, 65, 64].

By the mid-1990s, the number of hardware platforms based on reconfigurable logic was growing rapidly. One count placed the number at approximately 40 by the middle of 1994 [45]. A workshop dedicated to reconfigurable logic based machines was started in 1993 and appears to be increasing in popularity [15, 16]. A European conference previously oriented toward programmable and reconfigurable devices has begun to attract a number of reports on reconfigurable computing [95, 44, 96, 51].

Although a large number of systems have been designed and implemented in a very short period, they tend to fall into four major categories. Two relatively independent architectural parameters can be used to classify the systems into these categories: RPU size and dedicated local memory.

The first parameter, the RPU size, is the amount of reconfigurable logic used to implement the RPU. This value can be measured more or less by the number of equivalent logic gates in the RPU. This will determine the complexity of the functions which can be implemented by the RPU.

The second parameter is dedicated local memory. This is the memory directly accessible to the RPU. The absence or presence of dedicated memory will affect the system at several levels. Architecturally, dedicated memory implies that the reconfigurable portion of the system may operate independently from the host. From a software perspective, a programming model which supports an independent processor and memory space is indicated. Finally, at the application level, dedicated memory will affect the types of algorithms that can benefit from the use of reconfigurable processing.

Based on these two parameters, reconfigurable machines can be divided into four major categories. These are *Application Specific Architectures (ASA)*, *Reconfigurable Logic Coprocessors (RLC)*, *Custom Instruction Set Architectures (CISA)* and *Reconfigurable Supercomputers (RS)*. Table 2.1 shows the four types of reconfigurable architectures and their RPU sizes and presence or absence of dedicated local memory.

	No Local Memory	Local Memory
Small RPU	CISA	RLC
Large RPU	ASA	RS

Table 2.1: An architectural classification of reconfigurable machines.

In this table, a small RPU is defined to be less than 10^5 equivalent gates and a large RPU is defined to be greater 10^6 equivalent gates. This boundary is somewhat arbitrary and leaves a “grey area” for machines between 10^5–10^6 equivalent gates. Machines which have RPUs whose gate count is somewhere in this region may have features of two classes of machines.

2.2.2 Application Specific Architectures

The first class of reconfigurable systems are *Application Specific Architectures (ASA)*. These machines were some of the earliest to exploit the advantages of reconfigurable logic. They have no dedicated memory and have relatively large RPUs. These machines are primarily characterized by a very narrow area of application.

One popular use of such application specific architectures is in the acceleration of logic simulation [126]. Here, reconfigurable logic is used to prototype custom hardware. This approach has resulted in dramatic speedups over more traditional software simulations. A good overview of this area can be found in [99].

Another example of an application specific machine is *GANGLION* [21]. This machine was used to implement a fixed size three-layer neural network. This system made use of reconfiguration to provide a dramatic speedup over established software techniques. However, this hardware was only useful to implement a single neural network configuration. Modifying the number of neurons in the system was not possible.

While perhaps the earliest large-scale use of reconfigurable logic, these machines function much like custom hardware. While they may take advantage of reconfiguration to accomplish their tasks, they are typically used for a single application. In this sense these machines are more closely related to traditional fixed custom hardware than more general purpose reconfigurable machines.

2.2.3 Reconfigurable Logic Coprocessors

The second class of machines based on reconfigurable logic are called *Reconfigurable Logic Coprocessors (RLC)*. These machines are relatively small, with only a few thousand equivalent gates in the RPU. They contain dedicated memory directly coupled to the RPU. Since the RPU is relatively small, the memory on these systems is similarly limited. Typically on the order of 1 megabyte or less is provided.

Figure 2.8 gives a high-level diagram of the RLC approach. Some examples of this approach to reconfigurable computing are the *Algotronix 2x4* [3, 65], the *AnyBoard* system [125, 23], the *TUT-CA* processor [127] and the *BORG* system [19].

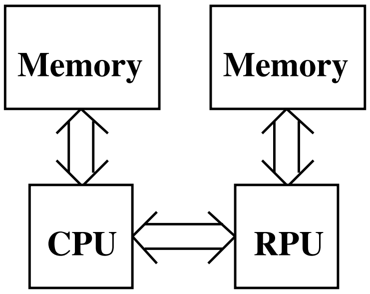


Figure 2.8: The coprocessor approach.

Because of the relatively small RPU, these systems are used primarily as small custom logic prototyping systems and are programmed using circuit design tools and methodologies. They may be used effectively to perform tasks of low computational complexity requiring high throughput. Digital signal processing is one fertile area of application for this class of machine.

2.2.4 Custom Instruction Set Architectures

The third type of reconfigurable system is the *Custom Instruction Set Architectures*, or *CISA*. These machines trace their roots to earlier custom microcode machines. They attempt to increase performance by providing customized instructions typically unavailable in traditional instruction set architectures.

Figure 2.9 gives a diagram of this approach to reconfigurable computing. These machines differ from reconfigurable logic coprocessors in that they are typically more tightly coupled to the host CPU and have no dedicated memory. Some examples of CISA machines are the *PRISM* systems [5, 7, 130], the *flexible processor* [132], *Spyder* [60], the *ArMen* machine [108], the *xputer* [49, 50] and the *CM-2X* [22].

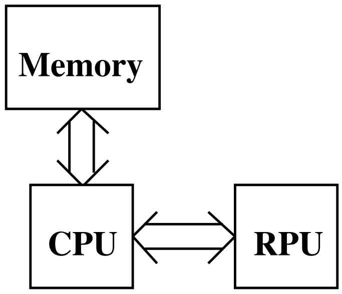


Figure 2.9: The CISA approach.

CISA machines typically offer a more traditional programming environment than other types of systems. This is primarily because the architecture is based on the instruction set model of computation. This shared programming model permits the host and RPU to cooperate closely. The function configured into the RPU is viewed by the host as another instruction available to the processor. This permits a simple interface for existing tools and languages. It is likely that these systems will continue to be used for research in high level language programming of reconfigurable machines.

While these machines tend to be easier to program, the use of the instruction set model of computing makes this approach more or less serial. While the RPU can effectively implement complex bit operations not found in traditional architectures, it is not possible to further exploit parallelism within the RPU via pipelining.

Additionally, scaling to larger RPUs permits more complex functions, but the increase in the amount of logic will tend to slow the speed of the RPU. Depending on the coupling to the host processor, this may require a decrease in the system clock speed.

Except for special cases, the CISA approach to reconfigurable computing offers relatively modest gains in performance. It is interesting to note that of the five machines cited above, three are multiprocessor systems. This multiprocessor approach should further boost performance by exploiting data parallelism, but at the cost of replicated hardware.

2.2.5 Reconfigurable Supercomputers

The final class of reconfigurable machines is *Reconfigurable Supercomputers (RS)*. These machines have large RPUs, on the order of one million equivalent gates. They typically have large amounts of dedicated memory and a high bandwidth link to a powerful host processor.

Architecturally, reconfigurable supercomputers resemble reconfigurable logic coprocessors. The difference is primarily one of scale. Reconfigurable supercomputers are several times larger (both in RPU and memory size) than reconfigurable logic computers. This permits these machines to implement larger and more complex algorithms.

Some examples of reconfigurable supercomputers are the *PAM* systems [9, 10], the *Splash* systems [40, 41] and the *Virtual Computer* [18]. All of these systems tend to be on the low end of the scale, with none having an RPU with one million equivalent logic gates. These systems are perhaps better referred to as reconfigurable mini-supercomputers. They are distinguished, however, by their large memory and I/O bandwidth, as well as their fairly powerful host machines.

Like the smaller reconfigurable logic coprocessors, these systems currently tend to be programmed using hardware design tools and methodologies. Because of their larger size, however, they can be used to implement larger and more complex algorithms, often involving more general arithmetic operations. Several applications have been implemented on these machines achieving speeds surpassing that of large vector supercomputers.

2.3 Other Related Work

The use of reconfigurable logic to perform computation represents a somewhat unique juncture in the development of computer systems. Work from several diverse areas of technology have combined to produce this technology. Rather than being a branch of any one particular area of research, reconfigurable systems represent a fusion of several established areas.

At the circuit level, the synthesis of logic functions is based on work in the areas of circuit design and computer aided design tools. Automatic circuit synthesis is particularly applicable. And because higher level arithmetic functions are often used by these systems to perform computations, research in techniques for constructing arithmetic circuits is a closely related area of research.

At the system level, many techniques used by vector and parallel supercomputers are used by or directly influence the architecture of these machines. In the approach studied here, software techniques for these high performance systems is a significant component.

While reconfigurable systems can trace direct lineage to several different areas of computing, some specific approaches will be employed in this work. Rather than attempting to list these details here, they will be referenced along the way, as various aspects of these systems are explored.

2.4 Overview of This Work

The work described in this dissertation proposes a technique for programming fine grained reconfigurable machines using high level languages. This technique is intended primarily for use with reconfigurable supercomputer class machines, but may apply to other systems as well. Based on this approach, architectural recommendations, both at the device and system level are made. From this architecture and programming model, several algorithms are implemented and simulated.

From these simulations performance estimates are made and compared to other high performance systems. These results are then used to make suggestions for further architectural refinements.

Chapter 3

Reconfiguration

With the commercial availability of relatively large reconfigurable logic devices and the evolution of computer aided design tools, it has become feasible to build fairly large and powerful systems based on reconfigurable logic. While it is clear that large gains in performance can be achieved with this approach, little has been reported on architectural issues concerning these systems. Unfortunately, these machines appear on the surface to be sufficiently different from existing approaches to computation that direct comparison to traditional architectures is difficult and often confusing.

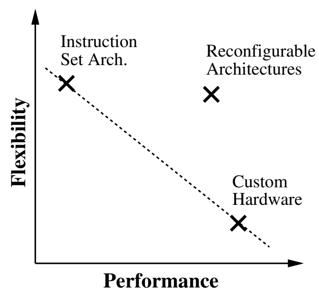


Figure 3.1: The traditional tradeoff of flexibility and performance.

Figure 3.1 gives a general diagram of traditional approaches to computation and their relation to reconfigurable architectures. Until the recent large scale use of reconfigurable logic to perform calculation, there was a generally accepted tradeoff between flexibility and performance in computing systems. In general, the more flexible a machine was, the simpler the programming, but the lower the performance. At one extreme, instruction set architectures can easily implement a wide variety of algorithms, but at only moderate levels of performance. At the other extreme is custom logic, which typically performs a single task very efficiently, but other tasks poorly or not at all. Between are various domain specific architectures which tend to trade performance for flexibility.

One parameter which is not explicit in this graph is system cost. Typically, cost is proportional to performance and inversely proportional to flexibility. With reconfigurable architectures, it is expected that costs will tend to be similar to that of more general purpose hardware. The use of commodity reconfigurable logic devices with modest requirements in density and clock speed should permit these relatively low system costs.

The use of reconfigurable logic appears to offer the flexibility of instruction set architectures with the potentially high performance of fully custom hardware. This unique combination has led to difficulties in analyzing reconfigurable machines. Performance comparisons to both custom hardware and to instruction set machines can be found in the literature. While these comparisons are useful for benchmarking, they provide little insight into how performance gains are achieved and what levels of performance can be expected for other algorithms.

3.1 A Reconfigurable Model of Computing

On selected algorithms, the performance of reconfigurable machines approaches that of custom logic. This level of performance is typically two to three orders of magnitude greater than that of implementations on instruction set architectures. For this reason, it is tempting to make performance comparisons to a custom hardware reference.

However, it should be a foregone conclusion that any custom hardware implementation of an algorithm can also be similarly implemented on a suitably large reconfigurable machine. The custom hardware implementation may be used to determine the maximum achievable performance for a given algorithm, but this is only useful in the cases where a comparable custom hardware solution exists.

Viewing reconfigurable systems as a form of custom logic does nothing to aid in predicting performance for algorithms for which no custom hardware reference platform exists. Neither is it clear that comparing a highly programmable machine to a fixed one is appropriate. Despite the similarities in performance, it appears that comparing reconfigurable machines to fixed custom logic implementations of algorithms can only be useful in providing a rough expectation of performance levels.

This ability to dynamically reconfigure hardware is sometimes viewed as the unique feature of machines based on reconfigurable logic. However, for any hardware to be used for more than a single purpose, some level of reconfigurability is necessary. A traditional instruction set processor may be viewed as a reconfigurable processor. At the heart of the system, the arithmetic and logic unit, or *ALU* can be viewed as a *reconfigurable processing unit*, or *RPU*. A dedicated path is provided to the ALU for rapid reconfiguration. Depending on the data sent to this port, the ALU performs various different logical operations on the inputs. At a higher level, this reconfiguration data is viewed as the operation codes which partially define the behavior of the system. Figure 3.2 shows and ALU with instruction operation codes being used to reconfigure the ALU.

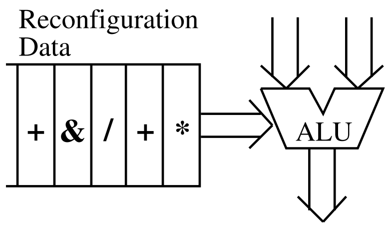


Figure 3.2: A reconfigurable model of computing.

From this perspective, the traditional ALU is actually a specific class of RPU. The ALU is characterized by:

- A dedicated port for reconfiguration data
- Few possible configurations
- Rapid but frequent reconfiguration

The number of bits used to configure a typical ALU is less than 10, with 8 being a representative number. This permits at most 2^8 unique configurations. These configurations define the operations available to the machine.

ALU reconfiguration typically takes place on the order of once per clock cycle. While the number of possible functions is limited, sequential combinations of these operations permit a large number of useful functions to be performed. While not particularly efficient for any single task, this approach provides a fairly constant level of performance for a wide variety of algorithms. The primary drawback to this scheme is the bandwidth consumed by constant reconfiguration. Since this bandwidth is limited, operations must be performed in a more or less serial manner. In spite of these limitations, the inherent flexibility of this approach has made it an extremely successful approach to computation.

By contrast, RPUs based on reconfigurable logic devices make large scale use of reconfiguration. Instead of the roughly eight bits used to reconfigure a traditional ALU, thousands of bits are used to reconfigure a typical RPU. This very large number of bits permits a very large number of possible functions. While providing increased functionality, the large number of bits which must be written to the control port of the RPU will consume system bandwidth. How this use of system bandwidth is managed will impact the efficiency of the system.

3.2 Instruction Replacement

Based on the reconfigurable model of computation, all reconfigurable machines can be viewed as coprocessors which implement custom instructions. A portion of the increase in performance comes from replacing a sequence of instructions normally executed on the host with a single complex instruction implemented in the RPU.

Equation 3.1 shows a sequence of n functions $f_1, f_2, f_3, \ldots$ performed by instructions in the host processor, producing a result y.

$$y = f_1(f_2(x_1, f_3(x_2, x_3)), ...) \tag{3.1}$$

This sequence of functions may be replaced by a single complex function F as shown in Equation 3.2. Neglecting all sources of overhead, this provides an n-fold increase in performance over the host calculation of y.

$$y = F(x_1, x_2, ...x_k) \tag{3.2}$$

CISA reconfigurable machines use this principle to provide performance increases of at most a factor of n. This assumes repeated use of the function F programmed into the RPU, with no other sources of overhead.

While the tightly coupled CISA approach can provide this increase in performance, the more loosely coupled RLC approach can not only replace these instructions in the same manner, but the function F may operate on data in dedicated memory. If there are no dependencies, the function F may be pipelined, increasing the throughput. In general, the more complex the function F, the more the function may be pipelined.

Finally, reconfigurable supercomputers have the same advantages as RLC machines, but their larger RPU permits even more complex functions. Rather than a small number of simple instructions, reconfigurable supercomputer class machines can provide this high level of throughput for larger sequences of instructions, including those using high level operations such as multiplication. It is this ability to execute large numbers of high level operations that give these machines performance levels of existing supercomputers.

3.3 Reconfiguration Overhead

Unfortunately, there are some barriers to achieving these increases in performance. The most significant barrier is the overhead of RPU reconfiguration. Thousands of bits of reconfiguration data must be written to the RPU for each new function to be implemented. How this use of RPU bandwidth is managed will determine the effectiveness of the system.

A simple analysis may be performed to decide whether or not it is profitable to implement a function F in the RPU of a system or whether it should be left to the host CPU.

Consider the case where a sequence of N instructions is to be executed M times, performing some processing function. In the instruction set architecture of the CPU, assuming one instruction per cycle and a CPU cycle time of t_{cpu}, the total time of execution, T_{cpu} is given by Equation 3.3.

$$T_{cpu} = (N * M * t_{cpu}) \tag{3.3}$$

To perform the same calculation on the RPU, some time, T_r must initially be spent on reconfiguration. This configures the function F in the RPU, replacing the sequence of N instructions with a single operation. If the RPU is able to execute the function at a clock speed of t_{rpu}, the total time of execution, T_{rpu} is given by Equation 3.4.

$$T_{rpu} = T_r + (M * t_{rpu}) \tag{3.4}$$

In order to profitably perform the function F on the RPU, the total time of execution on the RPU, T_{rpu}, must be less than the total time on execution on the host CPU, T_{cpu}. The break-even point, when both are equal is given by Equation 3.5.

$$M = \frac{T_r}{(N * t_{cpu}) - t_{rpu}} \tag{3.5}$$

If a simplifying assumption is made that both the host CPU and the RPU run at the same clock speed, T_{cpu} is equal to T_{rpu}. The equation reduces to Equation 3.6. Time has been removed from this version of the equation by normalizing with respect to clock cycles. The total time of reconfiguration, T_r is replaced by R, the number of reconfiguration cycles.

$$M = \frac{R}{N - 1} \tag{3.6}$$

What Equation 3.6 states is that the number of times that the function F must be executed on the RPU to break even is M. The greater the number of reconfiguration cycles, R, the larger M must be to break even. Similarly, the more instructions that are replaced, N, the fewer the number of times M that the RPU function must be executed to break even.

An interesting point concerning this equation is that R and N are not independent variables. Both in some form represent the complexity of the function being implemented. The number of bits necessary to specify the function should bear some relation to the complexity of the function as expressed in number of instructions replaced. The exact value of this relation is dependent on the both the structure of the underlying reconfigurable logic devices as well as the sort of instructions which are replaced.

One method for reducing this break even point is to decrease the total time of reconfiguration, T_r. One method of accomplishing this is to provide a higher bandwidth port for reconfiguration. Unfortunately, existing reconfigurable logic devices are not optimized for rapid reconfiguration. Reconfiguration data ports are often very narrow. Bit serial implementations are common.

But even if the full bandwidth of a device were dedicated to reconfiguration, the overhead would still be significant. If a reconfigurable logic device is assumed to be implemented as a square circuit n units on a side, the I/O bandwidth of the device may be assumed to be proportional to the perimeter of the device, or $O(4n)$. This assumes that all I/O occurs at the perimeter of the device. It may also be assumed that the amount of data necessary to configure the device is roughly proportional to the area of the device, or $O(n^2)$. This gives an increasing gap in reconfiguration overhead as the size of the device increases.

This reconfiguration overhead is one of the defining features of existing reconfigurable systems. Since a large number of cycles must necessarily be spent to configure the RPU, the function implemented in the RPU must be used a larger number of times to amortize this overhead. Existing systems are so slow to reconfigure that they are typically configured only once at the beginning of execution and seldom, if ever, reconfigured. This has served to limit the types of algorithms which can be successfully implemented on these machines.

While the high cost of reconfiguration cannot be completely overcome, it may be hidden. Reconfiguration may take place while the host executes a portion of the algorithm not suitable for the RPU. This assumes that the host is participating in the calculation and that reconfiguration will take little, if any, of the host resources.

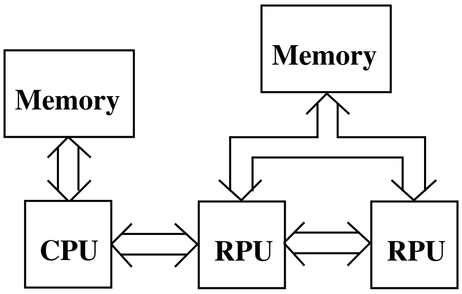


Figure 3.3: An RPU with two banks.

Figure 3.3 presents a second approach. A "banked" RPU may be used. Here, one bank may be reconfigured while the second is performing processing. When the second bank begins processing, the first bank may be reconfigured. If reconfiguration is not too frequent, this scheme can hide all of the overhead of reconfiguration, at a cost of doubling the RPU hardware.

3.4 Amdahl's Law

Another, perhaps more fundamental, barrier to performance is the inherent limitation on speedup. Often referred to as Amdahl's Law, this statement defines the limit of the performance of a given algorithm on a given machine. While usually stated in terms of multiprocessor machines, it is also applicable to the use of any subsystem which is used to increase overall system performance. This includes custom hardware and reconfigurable logic based machines.

Equation 3.7 gives the generalized version of the definition of *speedup* as it applies to reconfigurable logic based systems. This equation states that the speedup is equal to the unaccelerated execution time of a program, T_u, divided by the accelerated execution time of the program, T_a.

$$Speedup = T_u/T_a \tag{3.7}$$

In Equation 3.8, the accelerated version of the algorithm is broken down into three major components, the time used for reconfiguration, T_r, the time used executing code on the host machine, T_h, and the time of execution on the reconfigurable processor, T_{rpu}. Equation 3.8 gives this expanded version of the equation.

$$Speedup = T_u/(T_h + T_{rpu} + T_r) \tag{3.8}$$

The quantities in the denominator give the three limitations on speedup. Since it is possible to overlap both reconfiguration and the processing of data in the RPU with execution of host code, T_r and T_{rpu} can conceivably be zero. This leaves T_h as the only portion of the accelerated execution time which cannot be reduced.

This portion of the code is analogous to the "serial" portion of the algorithm in multiprocessor implementations of algorithms. Given the percentage of time T_h spent in this portion of the algorithm, the limit on acceleration provided by the reconfigurable hardware is $1/T_h$. If only ten percent of the time is spent in execution on the host, a maximum speedup of 10 can be achieved using the reconfigurable processing unit.

In order to make effective use of reconfigurable logic to perform processing, the algorithms must make significant use of the RPU. Algorithms which process data using the RPU only infrequently can expect diminished gains in performance.

3.5 Algorithms

Currently, the types of algorithms implemented on reconfigurable systems are quite limited. Due to the very slow reconfiguration of existing devices, it is difficult to achieve high levels of performance in algorithms which require frequent reconfiguration. The majority of algorithms currently implemented configure the RPU once at the beginning of execution and never reconfigure it. This has led to the implementation of algorithms which have traditionally been well suited to custom hardware implementations.

These implementations necessarily perform a single fixed function repeatedly. This has led to algorithms that can be categorized summarily as signal and image processing. This is taken to include cellular automata, which can be considered an image processing technique.

The second major limitation on existing machines is their size. The small number of configurable logic gates in the RPU eliminates all but the simplest bit-level algorithms from consideration. Once larger RPUs become available, larger and more complex algorithms are likely to be implemented.

As with existing systems, larger reconfigurable machines will also have to perform a large number of repeated operations to warrant their use. The difference will be that it will be possible to perform high level arithmetic and logical operations, not just simple bit level operations. Algorithms which perform large numbers of repeated high level operations can be currently found executing on parallel and vector supercomputers. It is expected that these algorithms, particularly those which operate on large amounts of data, can profit from reconfigurable computing.

Chapter 4

The Software Architecture

Ideally, we would like to take existing programs written in popular programming languages and execute them directly on a reconfigurable machine. Some interesting work in this area has been performed for the *PRISM* system [6, 5] and for *SPLASH* [42]. *PRISM* was able to compile small functions implemented using a subset of the *C* programming language into reconfigurable logic. This approach, while simplifying the software, showed only modest performance gains. The *SPLASH* work involves a description of a data parallel approach using reconfigurable SIMD hardware. No actual results on implementations of this technique have been reported. Other more theoretical work has been performed by the Hardware Compilation Group at Oxford University [83, 85, 101, 84, 82]. Direct execution of existing languages such as *C* has been shown to be difficult. Much of the problem appears to be the mapping of constructs from an instruction set machine to a reconfigurable architecture.

4.1 An Arithmetic Calculation

Before attempting to provide full programming language support, we will perform a simple arithmetic calculation. From this base other supporting constructs will be added. Consider the function in Equation 4.1.

$$X = \frac{(A + B) * (C + D)}{2} \tag{4.1}$$

For the purposes of this example the variables can be assumed to be integers. In reality, any data type or data representation may be used. A simple method of executing this code fragment on a reconfigurable architecture would be to reconfigure the RPU for each operation, and send the appropriate values from memory to the RPU. The results can then be read at the output port of the RPU. This leads to the following execution sequence:

1. Configure the RPU as an adder
2. Send A and B to the RPU and store the output in a temporary variable
3. Send C and D to the RPU and store the output in another temporary variable (no reconfiguration necessary)
4. Configure the RPU as a multiplier
5. Send the two temporary variables to the RPU and store the output in another temporary variable
6. Configure the RPU as a divide-by-two unit
7. Send the previous product to the RPU and read the output

In this mode of operation, we see that the RPU must be reconfigured frequently. In fact, unless two identical operations happen to be performed in sequence, the RPU must be reconfigured once per operation. This is undesirable because of the high RPU reconfiguration overhead. Since reconfiguration is a costly operation, this approach will be inefficient and will probably offer no advantage over traditional instruction set architectures. What is desirable is some methodology which reduces the reconfiguration overhead.

4.2 Vector Processing

One method of reducing the reconfiguration overhead is to execute algorithms in which operations are repeated frequently. In the example above, no reconfiguration was necessary between the two addition operations. Algorithms with many such operations in a row would have an advantage on reconfigurable hardware.

Fortunately, such repeated calculations do occur frequently. Looping constructs in programs are often used to repeat an operation many times. Special purpose coprocessors, usually called vector processors, have been designed and built to accelerate the operations of these repeated operations. High-level programming language support for these vector processors exists in varying degrees.

In a programming language like *C*, vectors are defined as arrays of values, and the operations are performed within loops. Consider the *C* code in Figure 4.1. This code fragment is used to perform the operation in Equation 4.1 on vectors of data. It is a simple matter to configure the RPU as an adder and to process vectors `a` and `b` and store the result in a temporary variable, and continue in a manner similar to that outlined for the scalar calculation.

Unfortunately, the code contains more than just arithmetic operators. The looping and indexing constructs are features used by the instruction set ar-

```
int a[SIZE];
int b[SIZE];
int c[SIZE];
int d[SIZE];
int x[SIZE];

for (i=0; i<SIZE; i++)
   x[i] = ((a[i] + b[i]) * (c[i] + d[i])) / 2;
```

Figure 4.1: C code to process vectors.

chitecture for addressing and sequencing control. Translating these constructs directly to produce vector code for the reconfigurable processor may be difficult or impossible. One approach is to perform the looping and indexing on the host processor and the arithmetic operations on the reconfigurable machine. This, however, would seem to offer little advantage to using the host processor to perform all of the calculations.

Another approach is to remove the explicit sequencing and indexing code. If we are performing a vector operation, these details are redundant. The length of the vector is already known. The pairwise operation may also be assumed.

The code fragment in Figure 4.2 illustrates the same vector addition using this vector or data parallel style of programming. And all the information necessary to process the vectors is readily available. Interestingly, this code is simpler than the original C code. This approach to expressing parallelism has recently gained popularity among users of large multiprocessors [12, 52, 54, 55, 110].

Based on this code fragment, the RPU may be configured once as an

```
int a[SIZE];
int b[SIZE];
int c[SIZE];
int d[SIZE];
int x[SIZE];

x = ((a + b) * (c + d)) / 2;
```

Figure 4.2: Vector C code to process vectors.

integer adder and each of the `SIZE` elements in the vectors may be added in sequence. Afterwards, the vector multiply and division operations may be performed in a similar manner. It is interesting to note that when processing vectors in this way, one arithmetic operation is performed per clock cycle. All of the other overhead associated with looping and address calculation is eliminated. It is this reduction in overhead that makes this approach, as well as that of more traditional vector processors, attractive.

4.3 Exploiting Temporal Parallelism

The previous example demonstrates that a reconfigurable machine can be used to process vectors efficiently. Using this mode of operation, the system would only require an RPU large enough to perform the most complicated operation offered by the language. Simpler operations will result in idle hardware in the RPU. One method of increasing both RPU utilization and overall performance is to execute operations in parallel.

In the example calculation, note that the vector multiplication of the two vector sums are immediately followed by the vector division by two. It should be possible to perform these two operations in a single pass, exploiting the temporal parallelism in this calculation.

Figure 4.3 shows a diagram of the functional units for the multiplication and the division. In this situation, the RPU in the reconfigurable machine would be configured as a multiplier feeding a divider.

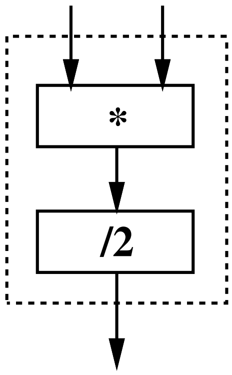


Figure 4.3: Chaining of functional units.

Performing multiple operations in this manner will increase the delay through the circuit. One method of reducing this delay is to pipeline the circuit. If latches are placed at the outputs of the two functional units, results can still be produced at a rate of one per clock. Pipelining will, however, increase the latency. In this case, it is not until the second clock cycle that a result is produced. If the vectors being processed are longer than the number of pipeline stages, the increased clock speed will compensate for the increased latency.

In addition to performing two operations in a single pass, there is an added benefit to pipelining. When the operations are executed separately, a temporary variable is required to store the intermediate result. When the results are fed directly from the multiplier to the divide unit, this temporary variable is no longer necessary. This technique can be viewed as a generalization of *chaining* as found in Cray vector processors.

Several operations may be pipelined in this fashion. The depth of the pipeline will be limited only by the algorithm being implemented. It may also be useful to pipeline the individual functional units internally. For instance, the multiplier unit may itself be pipelined. This results in a *superpipeline* which produces a result once per cycle, with a very low cycle time. While the latency will also be increased, it will be significant only if shorter vectors are processed.

4.4 Exploiting Spatial Parallelism

In addition to exploiting temporal parallelism through cascading of functional units and pipelining, it may also be possible to exploit spatial parallelism by performing independent calculations in parallel.

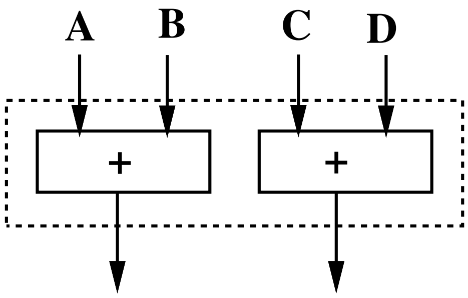


Figure 4.4: Parallel functional units.

In the previous example, both addition operations may be performed independently. Figure 4.4 demonstrates how the RPU could be configured to produce these results in parallel.

4.5 A Systolic Dataflow Framework

If these two techniques are combined, both temporal and spatial parallelism in the algorithm may be exploited. From the example discussed above, a circuit using both the cascaded pipelining of the two lower functions and the superscalar style parallelism of the addition operations may be constructed. Figure 4.5 shows the resulting circuit.

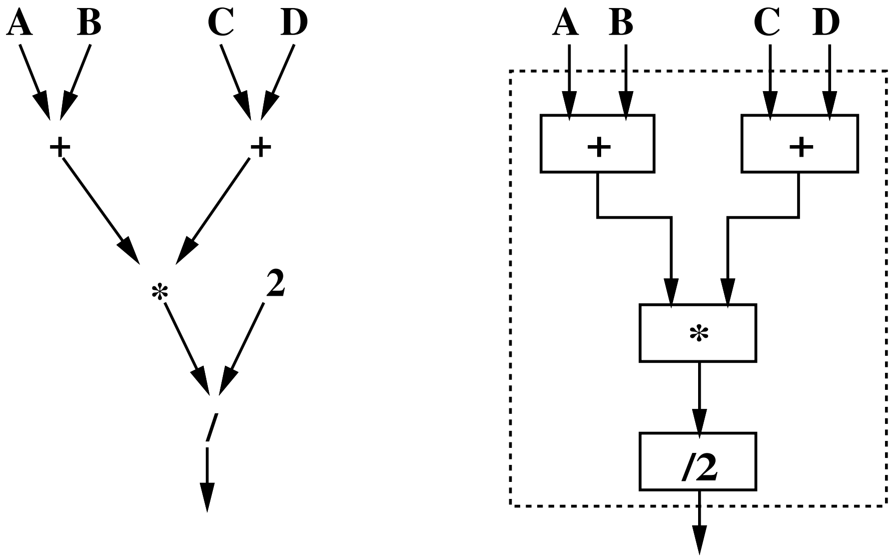


Figure 4.5: A systolic dataflow circuit.

Since we are using data dependencies in the algorithm to structure the interconnections between the functional units, we see that construction of a data flow graph for the algorithm results in a description that is easily mapped onto the hardware.

Since this circuit is also pipelined, all functional units have registered outputs and operate from a common clock. This technique takes advantage of all of the low level parallelism available in the algorithm.

4.6 The Scan Operator

Processing vectors is a useful operation and has been supported in several high performance processors and coprocessors. Now we look at performing non-vector operations on the data in vectors that have typically been left to slower non-vector processing units.

Vectors may also be processed using a class of powerful operations called *parallel prefix* or *scan* operations [72, 11, 75]. These operations were originally introduced in the *APL* programming language, but are more widely known through recent work in programming parallel architectures [54, 55].

The parallel prefix or scan operators are used as a bridge between scalar and vector operations. This permits operations to be performed on the elements within a vector. Consider the *add-scan* operation in Figure 4.6. Successive elements of the vector A are added and accumulated to produce the result vector.

A:	[1, 4, 7, 2, 6, 0, 3]
add-scan(A):	[1, 5, 12, 14, 20, 20, 23]

Figure 4.6: An example of the add-scan operation.

Other scan operations are also possible. An *or-scan()* function is used to perform a bitwise OR of elements in a vector. This may be used, for instance, to check if any error flags were set in some vector calculation. In general, any standard boolean or arithmetic operation can be used in this manner.

To take advantage of this programming construct, a logic circuit implementing the scan operation must be designed. If the add-scan parallel prefix operation is considered, a circuit for this operation can be constructed.

The add-scan function may be viewed as an accumulation. An adder circuit with the output fed back into one of the inputs may be used to implement such an accumulator. Figure 4.7 shows a circuit which implements the add-scan operation.

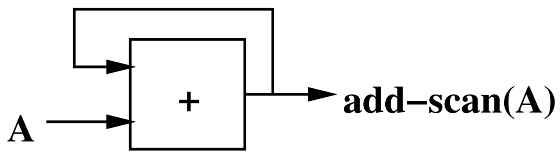


Figure 4.7: A circuit implementing add-scan().

It is a simple matter to extend this implementation of the add-scan operation to other parallel prefix operations. The *or-scan* operation would look identical to the *add-scan* operation in Figure 4.7 except that the functional unit would perform a bitwise OR operation rather than addition. The feedback connection would be the same.

4.7 An Example: Calculating e^x

An example which illustrates the systolic dataflow framework and the use of *scan* operators is series calculation. Consider the formula for the calculation of the Taylor Series for the exponential e^x in Equation 4.2.

$$e^x = 1 + x + \frac{x^2}{2!} + \frac{x^3}{3!} + \frac{x^4}{4!} + \cdots \tag{4.2}$$

On a conventional machine, this series would be calculated using looping constructs and temporary variables. It is possible, however, to consider this a vector calculation.

```
float denom[MAX];              /* Denominator */
float num[MAX];                /* Numerator */
float series[MAX];             /* Series terms */
float sum[MAX];                /* Series sum */
float result[MAX];             /* Final result */

denom = 1;                     /* [1, 1, 1, 1, ...] */
denom = add-scan(denom);       /* [1, 2, 3, 4, ...] */
denom = mult-scan(denom);      /* [1, 2, 6, 24, ...] */
                               /* [1!, 2!, 3!, ...] */

num = x;                       /* [x, x, x, x, ...] */
num = mult-scan(num);          /* [x, x^2, x^3, ...] */

series = num / denom;
sum = add-scan(series);        /* sum[MAX] = e^x - 1 */
result = sum + 1.0;            /* Add 1.0 to get e^x */
```

Figure 4.8: A data parallel C code fragment for calculating e^x.

A fragment of C code using scan operators to calculate the series for e^x is shown in Figure 4.8. The scan functions are used to first initialize vectors, then to perform calculations.

Figure 4.9 shows the dataflow graph and the resulting circuit for this algorithm. Note that the functional units for the scan operations are treated in the same manner as the other arithmetic and logical functional units.

Since there are six functional units operating in parallel, the level of

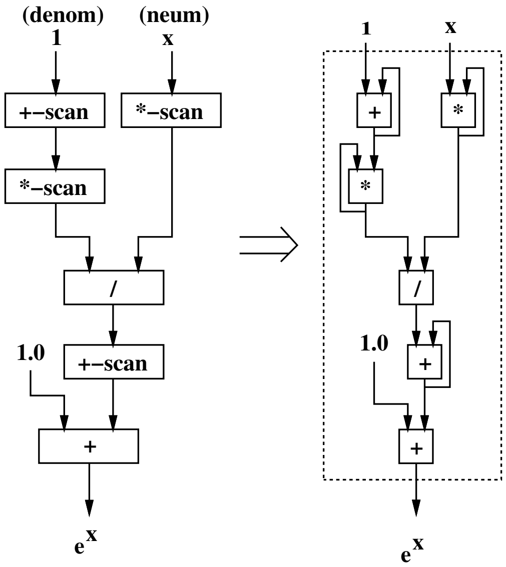


Figure 4.9: A circuit for calculating e^x.

parallelism at the operator level is, in this case, five. This level of parallelism is not, however, a function of the architecture. It is strictly a function of the algorithm. If the algorithm requires a more complex series of calculations, the level of parallelism will be higher. No matter how complex the calculation, given a large enough RPU, one result is produced per clock cycle, once the pipeline is filled.

4.8 Delay Balancing

When translating the dataflow graph into a pipelined circuit, some care must be taken to guarantee that results arrive at their proper place in the circuit at the proper time. Since the circuit is strictly feed forward, complicated pipeline scheduling problems seen in other pipelined machines can be avoided [68, 58].

While this technique avoids the problems of recurrent pipelines, there

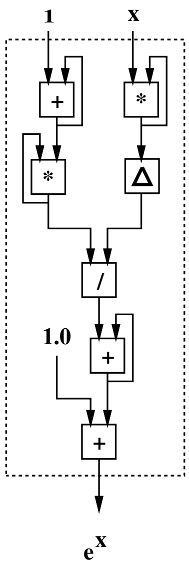


Figure 4.10: Pipeline balancing via delay insertion.

is still the problem of delay balancing. Careful examination of the circuit in Figure 4.9 shows that the results obtained will be incorrect. Assuming that each functional unit represents a pipeline stage with latched outputs, it is clear that the values arriving at the divide unit are not properly synchronized.

The vector elements on the left branch of the graph have passed through two functional units, placing the first vector element at the input of the divide unit at time $(t + 2)$. By contrast, the vector elements being processed by the right branch of the graph have only passed through one functional unit before arriving at the divide unit. This places the first element of the input vector at the divide unit at time $(t + 1)$. The proper values will not be divided.

One method of synchronizing this circuit is to insert a delay into the right branch of the circuit. This delay can be inserted at any point before the divide unit. In the diagram in Figure 4.10, the delay is inserted between the multiply functional unit and the divide unit. This will delay the numerator calculation by one clock cycle. This places the first element of both vectors at the divide unit at time $(t + 2)$.

The delay stage can be thought of as the equivalent of a "no-op" in an instruction set computer. Rather than slowing down the overall time of calculation as the no-op would, the delay stage simply uses hardware resources in the RPU. There is no performance penalty associated with insertion of delay stages. Results are still produced at a rate of one per clock cycle.

The example in Figure 4.10 assumes that all functional units have unit delay. It is possible that the functional units may be pipelined internally. This does not, however, change the basic delay insertion technique. As long as the delay in each branch of the dataflow graph is made to match, the algorithm will be implemented correctly.

Well-known techniques exist for the delay balancing of such graphs. Delay balancing algorithms specific to pipelined arithmetic units are discussed by Hwang and Xu [59] and Gao [39]. While delay balancing is critical to the correct operation of these circuits, subsequent circuit diagrams will not include explicit delay balancing elements. This is done to simplify the presentation of these diagrams.

4.9 Optimizations

The translation of the dataflow graph into a circuit is an effective method of programming a reconfigurable logic based machine. While this will produce a correct circuit from the description, it may be possible to perform optimizations that can reduce the complexity of the circuit, the delay through the datapath, or both.

The optimizations discussed here are based on popular existing optimizations for instruction based architectures [100, 2]. The difference is that optimizations performed on code sequences for instruction based architectures attempt to reduce the number of cycles required to execute a program fragment. The optimizations discussed for a reconfigurable architecture, however, reduce the amount of hardware necessary to implement the particular algorithm.

The first optimization is *strength reduction*. Strength reduction refers to the substitution of a complex operator with a simpler one. In Figure 4.11, the complex operation of multiplication is replaced by the simpler addition operation. Since a multiplier has a circuit area complexity of $O(N^2)$ and an adder has a circuit area complexity of $O(N)$, where N is the number of bits being multiplied, this particular optimization can result in a savings of a factor of N in hardware.

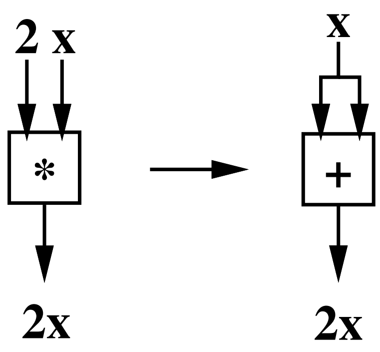


Figure 4.11: A strength reduction optimization.

Another optimization used by instruction set architectures that can be applied to a reconfigurable architecture is *common subexpression elimination*. This optimization eliminates redundant calculations by restructuring the equation. Consider the conversion performed in Equation 4.3.

$$2x^2 + 6x \Rightarrow (2x)(x + 3) \tag{4.3}$$

By rearranging the order of calculation, the number of operations is reduced. Figure 4.12 shows the circuits generated by the original and the optimized expression.

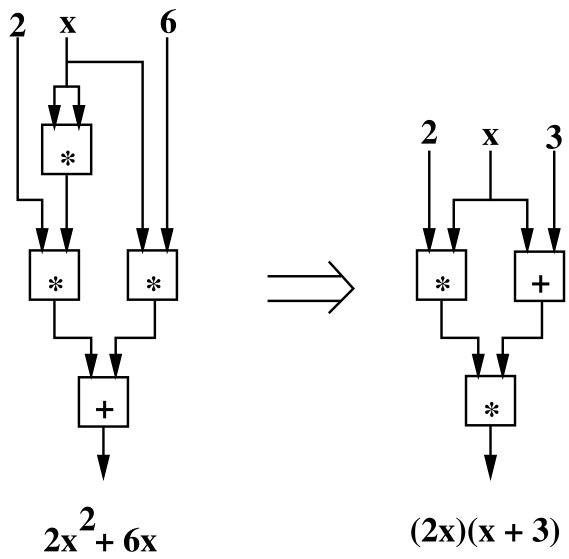


Figure 4.12: A common subexpression elimination optimization.

In this case the optimization has changed a circuit containing three multipliers and one adder into a circuit containing two multipliers and one adder. One multiplier has been eliminated. In addition to reducing the hardware complexity, the pipeline depth has also been reduced from two multipliers and one adder in depth to two multipliers. This will reduce the latency of the pipeline.

The final optimization discussed here is *constant optimization.* In this optimization, knowledge of a constant term in an expression is used to reduce the complexity of the calculation.

In the example in Figure 4.13, a multiplication by the constant 3 is replaced by a left shift operation (<<) and an addition. This reduces the hardware complexity of the operation from one multiplier unit to one adder unit and a shift operation. It should be noted that the shift operation may not necessarily be a functional unit consuming hardware resources. It may simply be the way in which the wires are routed to the adder unit. As such, the shift operation may not add any further complexity to the circuit.

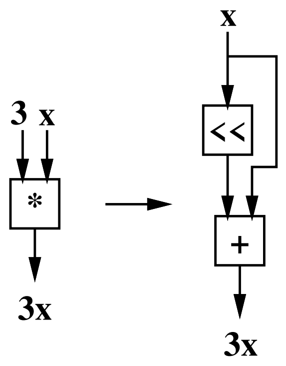


Figure 4.13: Optimization using constants.

Other optimizations used by instruction set architectures may be applied to reconfigurable architectures. It should be noted that optimizations based on control flow, such as loop unrolling, have no analog in reconfigurable machines. Only optimizations which reduce the complexity of a calculation are applicable.

Other lower level optimizations may also be applied to these circuits. These types of optimizations are based on logic optimizations and are performed at the gate level. These optimizations are not considered for two reasons. First, the elimination of single gates is insignificant compared to the reductions in complexity of the previous optimizations. Second, the effectiveness of these low level optimizations will depend on the underlying reconfigurable logic device architecture. Elimination of gates may not necessarily improve a design which is mapped to devices with large grained CLBs, for instance.

4.10 The Programming Model

It has been demonstrated that sequences of arithmetic and logical operations may be implemented on a reconfigurable architecture by translating the data flow graph of the vector operations directly to a pipelined circuit. Providing full programming language support from this point requires some further extension. For the remainder of this work the syntax of the *C* language [66] will be used. The only extension of the *C* syntax is the ability to perform vector operations on arrays of data. Rather than indicating a pointer to memory, the array data type will represent a vector of a given length. For those familiar with the *C++* language [27], this may be viewed as overloading arithmetic and logical operators for vector data types.

This approach has three major advantages. First, it does not require users to learn a new language or new language constructs. Second, it leverages the large body of existing software, including compilers, debuggers and other tools as well as applications.

Lastly, and perhaps most significantly, the explicit definition of vectors greatly simplifies the compilation process as well as the interaction between the host and the RPU. All code fragments which operate on vectors will be executed on the RPU. All other code will be executed on the host. This simple arrangement not only provides an effective method of partitioning the scalar and the vector code, but also gives the user control over how the RPU will be utilized.

This technique for compiling vector based data parallel code for reconfigurable hardware also simplifies other portions of the system. Most significantly, the pipelining of the circuits permits the RPU to run at a single fixed clock speed. In the case where circuits are synthesized in a more ad-hoc manner, the delay through the RPU may vary substantially. No simple technique exists for determining the maximum clock speed of the configured circuit. This approach avoids this problem while providing a relatively high clock rate.

Finally, this approach is based on the use of high level macrocells for arithmetic and logical operators. It is expected that these macrocells will be created by hand and optimized for the particular underlying reconfigurable logic devices which make up the RPU.

While traditional programming languages have data types which are associated with some fixed data width, this is not necessary with reconfigurable hardware. Where increments of eight bits in width are usually specified for most values, reconfigurable logic permits arbitrary datapath widths. Some languages, such as *Ada* permit specification of the width of the data types. While not a necessity, the ability to specify the width of the data types and their associated data paths would further increase the flexibility of the system.

In addition to permitting user specified datapath widths, the high level language compiler may also be used to specify the widths of the internal data paths to preserve accuracy.

Figure 4.14 shows an instance where two eight bit values are added, producing a nine bit value. In this situation, the increase in the width of the data path eliminates the possibility of an overflow. In other instances, such as when performing a division, increasing the width of the data path may be used to increase the accuracy of the result. The ability to vary the width of the data path in this manner permits more accurate calculations without adversely affecting the external bandwidth of the system.

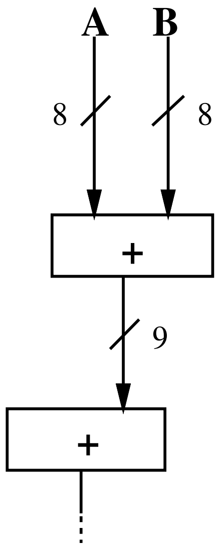


Figure 4.14: Preservation of accuracy.

4.11 Mixed Valued Striding

So far, a method for performing pairwise operations on vectors has been described. While a common operation, there are many cases where elements within a vector are accessed in other ways. In traditional programming languages, vector or array elements are accessed by performing some calculation to produce an index value. This index value is then used as an offset into the data. What is desirable is some method of accessing data within vectors without resorting to this type of indexing technique.

One method which provides some deviation from the standard linear incremental vector access is *striding*. Stone defines *stride* as the constant difference between successive addresses in a stream of data generated by a vector access [118]. Using strides in this manner, it is possible to access every Nth element in a vector. Additionally, special purpose hardware can be used to accelerate vector striding.

But by defining stride as a difference between addresses, Stone unnecessarily restricts stride values to that of integers. While existing machines have used this approach exclusively, it is proposed here that an extension to the traditional implementation of striding using mixed rather than integer values be used. This allows useful access patterns to the vector data to be produced. In many cases, this technique can be used to produce the types of data streams currently only achievable with indexing and scalar hardware.

The common software technique for creating address streams to access arrays or vectors of data is to use a looping construct. The parameters of this looping construct typically specify a starting index, an ending index and a step value. These values are usually represented as integers and used to directly index arrays or vectors of data.

In most programming languages, the values used by these looping constructs may be of any data type. But when these looping constructs are used as array indices, they are most often integral. It is possible, however, to use non-integral values in these looping constructs, so long as the final value used as an index is an integer. If mixed or fixed point values are used in the looping constructs, the final index can be converted into an integer by simply ignoring the fractional portion.

From this software model, one additional parameter will be added. This is the *modulus* value. This is an integral value which is used to limit the magnitude of the index. Before an index is added to the starting address, its modulus is taken. For a modulus of N, this remaps the index into the range 0 to N. The modulus is ostensibly used to eliminate illegal accesses, but it also has other more powerful uses.

Rather than represent vector striding in the context of a control loop, a function containing all of the necessary parameters will be used. Not only is this a more concise representation, but it removes the notion of a loop, which may or may not exist in the actual system being programmed. It also serves to separate what is purely a memory access operation from other calculations involving program data. The function takes as its input parameter a vector and produces as its output a vector. The syntax is shown in Figure 4.15.

$$V_{out} = \text{stride}(V_{in}, start, strideval, length, modulus)$$

Figure 4.15: The syntax for the mixed valued stride operation.

Because of the wrap-around nature of the modulus, the end parameter has been replaced by *length*. This specifies the number of elements in the resultant vector, V_{out}. This is a simpler and more reliable termination condition, and may be derived easily from the *start*, *end* and *step* looping paramenters in the software looping construct.

For example, consider the vector $A = \{1, 4, 7\}$ with a stride of 1/3 and a length of N^2, or 9. The resulting vector A' contains the values $\{1, 1, 1, 4, 4, 4, 7, 7, 7\}$. This data access pattern is commonly used in nested loops.

One of the most common uses for the traditional integral stride is to access an array by columns when it has been stored in row-major form. For an NxN array, this takes N vector operations of lengths N, while access by rows requires only a single vector operation of stride 1 and length N^2. By using a stride of $(N + 1/N)$ and a modulus of $(N * N)$, the array may be referenced as columns in a single vector access. Figure 4.16 shows a simple example of this access pattern.

$$
\begin{aligned}
\text{A} = [1, \quad 2, \quad 3,& \\
4, \quad 5, \quad 6,& \\
7, \quad 8, \quad 9]&
\end{aligned}
$$

$$A' = \text{stride } (A, 1.0, (N + 1/N), (N * N), (N * N))$$

$$
\begin{aligned}
A' = [1, \quad 4, \quad 7,& \\
2, \quad 5, \quad 8,& \\
3, \quad 6, \quad 9]&
\end{aligned}
$$

Figure 4.16: Column access of row major data ($N = 3$).

The perfect shuffle is another data access pattern that would require two vector accesses with integral striding. Using a mixed valued stride of $((N+1)/2)$ and a modulus of N, a perfect shuffle access pattern on a vector of length N

can be performed in a single vector operation. In addition, the vector can be restored to its original state with a stride access of $(2 + 2/N)$

Reversal of a vector can be achieved by using a start value of N and a stride of -1. Because of the operation of the modulus, it is also possible to reverse a vector using a starting value of N and a stride of $(N - 1)$. If other such cases are examined, it is clear that all access patterns can be reduced to strides greater than or equal to 0 and less than N. This limitation in the range of the stride is significant, because it can be used to limit the complexity of special purpose striding hardware.

Table 4.1 gives some examples of striding values and the result produced. While these data access patterns are some of the more popular, the flexibility of this approach permits may other interesting patterns.

Start	Stride	Length	Modulus	Description
N	-1	N	N	Reversal
N	$N-1$	N	N	Reversal
$(N/2)$	1	N	N	Butterfly
0	$(N/2)+(1/2)$	N	N	Perfect Shuffle
0	$2+(N/2)$	N	N	Reverse Shuffle
0	$N+(1/N)$	$(N*N)$	$(N*N)$	Matrix Transpose
0	1	$(N*N)$	$(N*N)$	N^2 by row
0	$(1/N)$	$(N*N)$	$(N*N)$	N^2 by column

Table 4.1: Some useful striding parameters.

So far, the stride software construct outlined is not much more than a recasting of other data accessing techniques. While it does provide unique access patterns, these could easily be mimicked using existing software structures. The benefit to this approach is that the memory access patterns are separated from the computation. This permits special purpose hardware to be used to generate the data stream. Additionally, the vector nature of the problem is explicitly specified, aiding compilers in generating efficient vector code. Figure 4.17 shows one possible implementation of the mixed valued striding hardware.

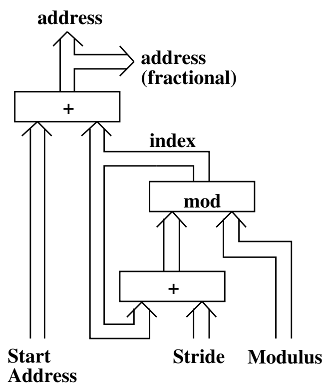


Figure 4.17: The striding circuit.

In the stride circuit, the adders are simple arithmetic units. The modulus operator is, in the general case, more complex. In this situation, however, a general purpose modulus operator is not necessary. Since this function is used to re-map values back into the range of the vector, a simple comparator and a subtracter can be used to provide the necessary functionality. When the mixed valued address is incremented past the *modulus* value, the *modulus* value is subtracted from the mixed valued address. This provides the necessary functionality for a stride between 0 and N. Figure 4.18 shows one implementation of the simplified modulus circuit.

Another concern when dealing with mixed value numbers is exact representation. In the case of a reciprocal, only values of N which are an even power of two may be exactly represented. Padding of vectors to these lengths

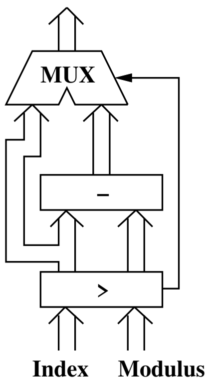


Figure 4.18: The modulo circuit.

is one alternative. Another is simply to have enough bits in the fractional portion of the stride to guarantee the necessary accuracy. This will be application dependent, but it is expected that a fixed point representation with a fractional portion equal to roughly half the integral portion of the address will be sufficient.

Mixed valued striding is a simple extension of the classic vector striding technique. The use of mixed value, rather than integral values allows the concise specification of complex reference patterns which produce longer uninterrupted data streams.

The hardware implementation of this technique is only incrementally more complex than traditional integral striding hardware. As a further simplification, this hardware has a direct mapping to the software construct.

It should be again noted that the use of a *stride()* function only defines an access pattern to a vector. It is not necessary to physically move data, as long as there is direct hardware support for the striding. This approach should help to alleviate some of the flexibility lost in going from a scalar looping style control structure to a vector approach. Most of the algorithms implemented in later chapters will make use of this construct.

Chapter 5

The Microarchitecture

Historically, programmable and reconfigurable logic devices have been used to simplify the hardware design process. While many high level functions are available commercially in standard packages, some portion of the design must usually be customized. This customization often involves small amounts of interface circuitry sometimes referred to as "glue" logic. These circuits are usually fairly unstructured and require the ability to arbitrarily interconnect logic elements.

Because of the unstructured nature of these circuits, popular reconfigurable logic devices have usually been designed to efficiently implement these types of circuits. While these commercially offered devices have been used for experiments in reconfigurable computing, they are not necessarily well suited for the types of circuits produced by a high level language approach of the type introduced here.

The implementation of reconfigurable processors may be considered a specific application area for reconfigurable logic. Because of the more specific nature of the circuits used by a reconfigurable machine, it may be possible to provide a more nearly optimal programmable logic device for this application domain. The three major architectural parameters of a programmable logic device, the cell granularity, the interconnect structure and the input/output structure, will be defined with the goal of producing an architecture directed toward pipelined arithmetic and logical circuits.

5.1 Existing Programmable Logic Architectures

Some previous efforts have proposed devices optimized for arithmetic circuits. These include Labyrinth [36], the CAL [65], and TRIPTYCH [26]. While each takes a slightly different approach, all have some common features. First, all three favor local interconnections. TRIPTYCH adds some global channel routing, but uses a predominately nearest neighbor interconnection scheme. Curiously, this nearest neighbor style of interconnect was used exclusively in nearly all of the early research in programmable logic [93, 114].

A second common feature is a smaller cell size. Where devices geared toward random logic such as Xilinx [133] contain cells with as many as 9 inputs and 3 outputs, the devices oriented toward arithmetic circuits opt for 2-3 inputs and 1-3 outputs. Internally, however, the logical functions computed by the cells typically use some subset of the inputs. All three calculate only a single logic output. Other outputs are pass-throughs of input signals, or a fanning out of the single logic output.

This common direction away from global programmable interconnection networks toward local interconnect can be easily justified. Existing random logic oriented programmable logic devices use as much as 75 per cent of their silicon area to implement the programmable interconnection network. While this is necessary for unstructured circuits, the more regularly structured arithmetic and logic circuits of the RALU do not require this large overhead. By moving to a nearest neighbor interconnection scheme, several times the number of cells can be made available.

Some additional benefits also accrue from the use of fixed interconnects. First, the fixed interconnects are faster. Programmable interconnects must pass through one or more programmable routing switches. Each pass through a routing switch reduces the maximum speed of the interconnection.

Problems with bus contention are also avoided. With programmable interconnections, it may be possible for two outputs to drive the same line, possibly damaging the device. Finally, fan-out is no longer an issue. With programmable logic, several inputs can be connected to a single output. At best this slows the speed of the circuit, at worst the circuit ceases to function correctly.

5.2 The 3I/3O/1F Cell

Since the problem domain of the programmable logic architecture being defined is calculation, the programmable logic cell should be well suited to produce circuits for common arithmetic and logical operations. The architecture must also support the interconnection of these operators. A three input, three output cell with one bit of feedback (3I/3O/1F) has been selected as the basis for the architecture. This granularity was chosen for practical reasons. The three inputs and three outputs can be used to implement a full adder, which is a basic building block for adders and multipliers. Using full adders, these circuits can be constructed using only nearest neighbor interconnections. Second, data can flow to the left, right and downward. A smaller cell would restrict the flow of data unnecessarily. Finally, the six connections indicates a hexagonal structure. This permits dense packing of cells in the array.

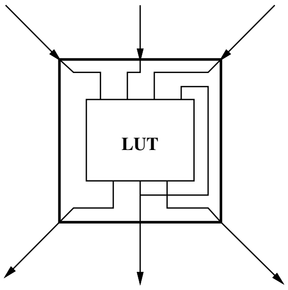


Figure 5.1: A 3I/3O/1F cell.

In addition to the external inputs and outputs, a single feedback input is provided. This data path is important for state machine circuits as well as parallel prefix or "scan" circuits. Finally, all outputs of the cell are registered and latched by a common clock signal.

This cell is currently described as a Look Up Table (LUT) containing four inputs and three outputs. The exact circuit implementation of this cell is not important, but for now it may be considered a 48 bit static RAM. Other more efficient implementations may be possible.

5.3 The Hexagonal Array

The three inputs and three outputs of the cell described in the previous section immediately indicate a hexagonal interconnection array. Figure 5.2 shows this interconnection structure. The array is strictly feed-forward, with inputs at the top of the array and outputs at the bottom. The center input of each cell at the top of the array is used as a device input. Similarly, the center output of each cell in the bottom row of the array is used as a device output. The other cell inputs and outputs around the array perimeter are not used. These perimeter inputs are assumed to have a logic value of zero.

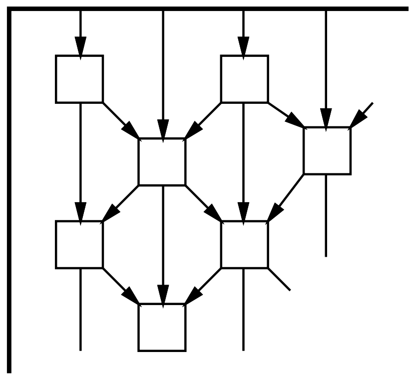


Figure 5.2: The cellular array.

It is also possible to provide I/O connections for signals on the left and right edges of the array. This will permit multi-device expansion of the array in width, as well as in depth. Current indications are that such expansion of the width of the array may prove to be unnecessary. Typical algorithms appear to favor narrow, but fairly deep arrays. Additionally, it is expected that the large number of cells in the array will preclude providing I/O access to all cells along the perimeter.

The hexagonal structure of the array also introduces some timing issues. Data traveling downward in the array proceeds at twice the rate of data traveling diagonally. To remedy this situation, a two-phased clocking scheme is used. Alternate levels within rows of cells in the array are clocked on alternate phases of a global clock. This causes data to flow at the same rate both downward and diagonally. This scheme permits the pipelining of circuits no matter what the path data takes through the array.

5.4 Some Example Circuits

A hexagonal array of 3I/3O/1F cells is proposed as a medium for the construction of arithmetic and logic circuits. In this section, some commonly used high-level arithmetic and logic circuits are implemented and discussed. Since these circuits are intended to be used as macrocells to construct high-level algorithms, some restrictions are placed on their design.

First, all macrocells are rectangular. All inputs must enter the macrocell at the top and all outputs must exit at the bottom. While this may potentially increase the number of cells as compared to arbitrarily shaped macrocells, a rectangular shape will simplify the placement and interconnection of the macrocells.

The second restriction is that data must enter and exit the macrocells as *buses*, not individual bits. Again, since these macrocells are to be cascaded and interconnected to form more complex circuits, it may be expected that data will arrive grouped in buses. This will tend to increase the number of cells used to transfer data, but will give a more realistic estimate of the size of the circuits produced.

Another restriction on the design of the macrocell circuits is the location of the inputs and outputs. All circuit inputs will use the center input signal on the cells in the upper row of the macrocell. All outputs will be similarly produced by the center outputs of the bottom row of cells. This is in a fashion similar to the restrictions for physical input and output of data in the device. Again, this may serve to increase the number of cells necessary to implement a given function, but the standardization will permit simpler interconnection of the macrocells.

The final requirement is that the macrocell constructed must be easily extended to N bits of accuracy. While this requirement is somewhat vague, in spirit it is intended to eliminate special circuit optimizations which may be only narrowly applicable. This will allow simpler construction of new macrocells, whether manually or automatically.

5.4.1 Boolean Calculations

A simple, but important operation is a bitwise boolean operation. Figure 5.3 shows one implementation of a 4-bit AND operation. Note that the majority of the cells are used to interleave the data from the two 4-bit buses. Only four cells are actually used to provide the ANDing functionality.

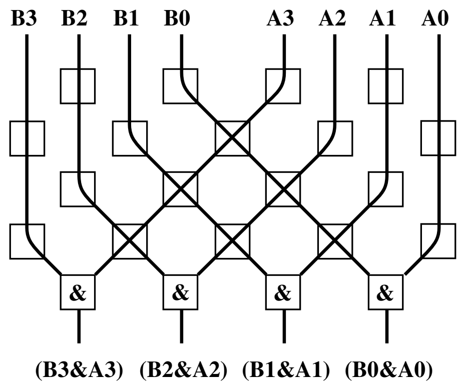


Figure 5.3: A 4 bit AND implementation.

While the majority of the cells in this circuit are used to route data to their proper location, this macrocell serves as an example of how this architecture makes explicit the cost of data movement. In architectures containing programmable interconnections, the use of these resources is usually not explicit. While cell utilization is usually measured, routing is typically "free".

Unfortunately, there is always cost in silicon area on the device when routing signals. This architecture quantifies and makes explicit the cost of data routing.

5.4.2 An Adder Circuit

In a fashion similar to the boolean circuit in Figure 5.3, a 4-bit adder macrocell is constructed in Figure 5.4. Unlike the boolean implementation, the data dependency of the *carry* bits produce the diagonal shape of the adder logic. Again, most of the cells are used to transport, rather than process, the data.

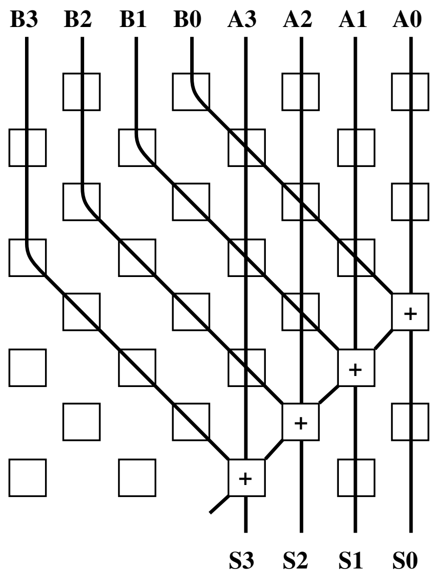


Figure 5.4: A 4 bit adder implementation.

This macrocell represents the implementation of a classic ripple-carry adder circuit. More sophisticated implementations with various tradeoffs of size and latency may also be constructed. Experimentation with other adder circuits, however, indicates that the ripple-carry approach may be superior for this architecture. Other adders depend on extra logic and interconnect to increase the speed at which the carry signal is propagated. Because of the structure of the array, the movement of data in all directions is quantized. N steps are always necessary to propagate data across N cells. There is no simple method for communicating the carry information across N cells in less than N steps.

5.4.3 A Multiplier Circuit

The 2-bit multiplier circuit in Figure 5.5 shows a more complex macrocell. Its construction is based on 2x2 *supercells*. A more compact representation is not possible, primarily because of the need to propagate both data and carries downward and to the left.

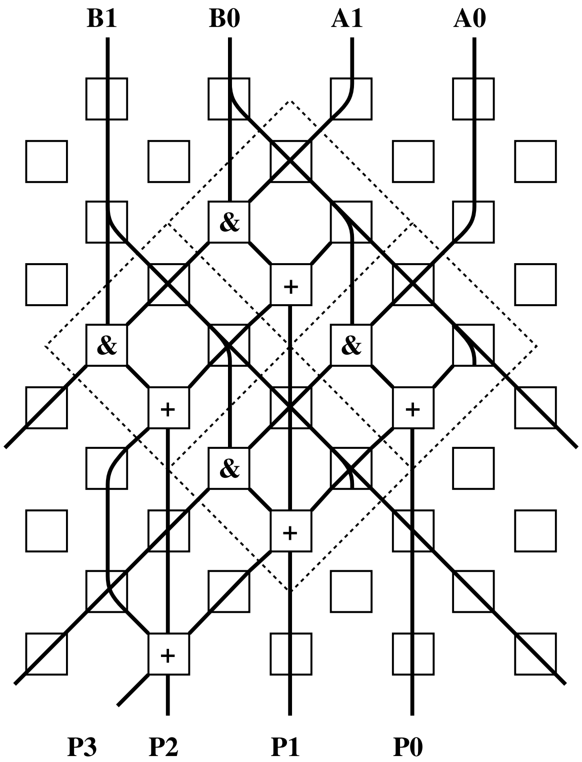


Figure 5.5: A 2 bit multiplier implementation.

This implementation is based on the summation of all possible product terms A_iB_j. These are produced by the *and* cells (&), while the full adders (+) compute the sums. Sums are propagated downward. This implementation is similar to standard array multipliers [86].

One feature of note is that the inputs and outputs of the macrocell are spaced twice as far apart as the previous macrocells. This may result in extra cells being used to connect this macrocell to narrower macrocells such as adders. Connection to other double-wide macrocells such as other multipliers, however, incurs no penalty.

A final notable feature of this macrocell is the natural buses that form the inputs. An interleaving of buses as seen in previous macrocells is not necessary.

5.4.4 Scan Circuits

The feedback connection in the cells is used to implement scan operators. Figure 5.6 gives an implementation of an *add-scan* circuit. Here, input values are accumulated by the adder. The structure of the macrocell is similar to the adder in Fig. 5.4, except that there is only a single input and a single output. The use of the feedback connection is indicated in the figure.

It is somewhat surprising that this relatively complex operation can be implemented with such economy. The fact that only a single input is used reduces the data communication cost and dramatically reduces the number of cells necessary to implement the function.

Similarly, a boolean scan operation, *and-scan* is shown in Fig. 5.7. Here, the boolean function is implemented in a very efficient manner. Since there is

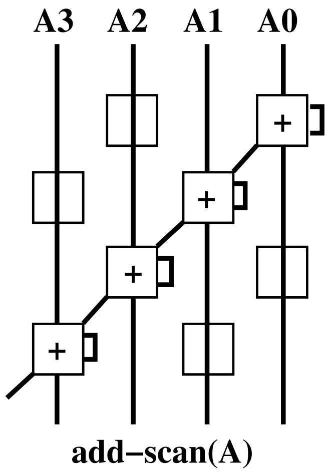


Figure 5.6: A 4 bit add-scan implementation.

only a single input and output and each bit is calculated independently, no extra cells are used to route data.

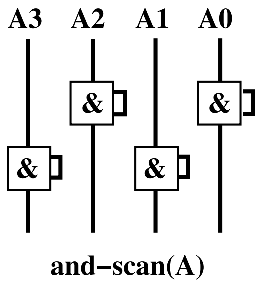


Figure 5.7: A 4 bit and-scan implementation.

5.5 Circuit Complexity

Table 5.1 gives the circuit complexity as the number of cells and latency in full clock cycles. Note that all circuits except the boolean *scan* require $O(N^2)$ cells and are $O(N)$ latency.

Using these measures of complexity, it is possible to estimate the number

Circuit	Cells	Latency
Boolean	$N^2 + 1.5N$	$N/2 + 1$
Adder	$2N^2$	N
Add-scan	$N^2/2$	$N/2$
Multiply	$12N^2 - 3N$	$3N$
Boolean scan	N	1

Table 5.1: Complexity of the arithmetic circuits.

of cells necessary to implement a given high-level algorithm. Additional cells may be necessary to act as buses to move data to the desired functional unit. The exact count of the number of cells necessary will not be known until the compilation process for a given algorithm is complete.

This use of macrocells to implement algorithms greatly reduces the complexity of the placement and routing problem. First, since there is a pre-defined flow of data from the top of the array to the bottom, placement is no longer an issue. Macrocells are placed from the top of the array downward in the order in which the operations are performed. Routing in simply interconnection of the outputs of one macrocell to the input of another. This is often accomplished by simple abutment.

This scheme permits a higher level approach to algorithm synthesis. By relying on a library of high-level arithmetic and logical operators, many of the issues of concern to more general logic synthesis are avoided. Placement of macrocells and routing of interconnections is greatly simplified. Additionally, the synchronous methodology avoids timing issues usually associated with general purpose circuit synthesis.

Experience with algorithm compilation indicates that the array should be relatively narrow, but deep. The width of the array should be at least $2N$, where N is the width of the input data operands. This permits two N-bit vectors as input, with either two N-bit vectors, or a single double accuracy $2N$-bit vector as the result.

The depth of the array depends on the complexity of the algorithm. Simple algorithms have fewer operations, and a shorter cascade of macrocells. More complex algorithms tend to use deeper arrays. Since the processing unit may consist of several devices chained with one output feeding another's input, expansion is fairly straight forward. Of course, both wider and deeper arrays may be emulated on a smaller reconfigurable processing units by performing the calculation in phases. Partial result vectors are calculated; the RALU is reconfigured; and further calculation performed on the partial results. This process proceeds until the final desired result is achieved.

Chapter 6

The System Architecture

The architectural model of the reconfigurable machine used in this study consists of three major components, the host machine, the *Reconfigurable Processing Unit*, or *RPU* and the memory system. Figure 6.1 shows these system components and their relation.

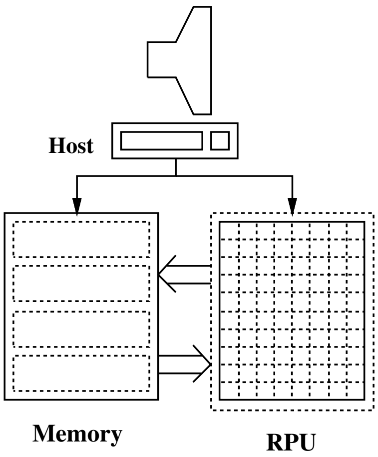


Figure 6.1: The system architecture.

6.1 The Host Machine

The host machine is responsible for all input and output to the reconfigurable hardware. This includes the configuration of the RPU as well as the loading of data and the unloading of results from the memory system. It is expected that the host machine will also run the development tools used to program the reconfigurable hardware.

In addition to these functions, the host machine will also provide the necessary control for the reconfigurable hardware. It will coordinate reconfiguration and processing. Some algorithms may also contain computations that are ill suited to the reconfigurable hardware. Where appropriate, the host machine will perform selected portions of the algorithms in conjunction with the reconfigurable hardware.

Since it is expected that the reconfigurable hardware will operate at supercomputer levels of performance, it is necessary that the host portion of the system be sufficiently powerful. It should provide the necessary bandwidth to the reconfigurable processor for both I/O and reconfiguration. Since it is possible that the host will participate in the execution of the algorithms, its performance should also be high enough to provide a suitably balanced system.

6.2 The Reconfigurable Processing Unit

The RPU is a collection of reconfigurable logic devices. While any reconfigurable logic device capable of implementing arbitrary digital circuits may be used for the RPU, it is assumed that the 3I/3O/1F device architecture discussed previously will be used to construct the RPU. This architecture is able to easily implement high level arithmetic and logical functions like those commonly found in high level languages. This capability will simplify the translation process as well as simplifying the analysis of the configured circuits.

It is also assumed that the RPU is large enough to completely implement each of the algorithms being studied. Rather than fix an arbitrary size and geometry to the RPU, the results taken from the simulation of real algorithms will be used as a guide to determine these design parameters.

Finally, it is assumed that a sufficient number of ports to and from the memory system will permit data to be available as it is needed. As with the size and geometry of the RPU, simulation results from selected algorithms will be used to estimate suitable values for RPU bandwidth.

6.3 The Memory System

The memory dedicated to the RPU is assumed to be large enough to hold all of the data used by the algorithm. It should also be able to deliver all of the input data and store all of the results as required by the algorithm. A reasonable, but modest, clock speed of 50 MHz has been assumed for the system. Since the algorithms will generally be vectorizable, the hardware may take advantage of this fact.

In traditional processors, cache memories are one technique used to provide a faster primary store for data. However, caches are designed based on locality of reference principles. That is, data used once are likely to be used again and data near recently used data are also likely to come up for use.

These principles were derived from code traces on very long executions of older codes written specially for instruction set machines. These data access patterns are substantially different for vector architectures. The absence of caches in modern vector supercomputers indicate that they are not an effective means for increasing the performance of vector applications [118].

The dedicated memory unit is also expected to directly provide input data and store output results from the RPU on each cycle. This roughly corresponds to memory to memory style operations in instruction set machines. While these types of operations were supported by early vector supercomputers such as the Control Data and Burroughs machines, as well as smaller non-vector instruction set machines, they appear to have have fallen out of favor. Both vector and scalar instruction set architectures today opt for operations involving processor registers. All memory accesses are made explicit with *load* and *store* operations.

Caches as well as processor registers can be used advantageously only when data are reused frequently. This applies to both scalar and vector registers. In instruction set architectures, where operations are performed serially, the intermediate results are stored in registers so that they can be used in later stages of the calculation.

In a reconfigurable logic based machine, however, these stages of execution are performed in parallel in a single pass. It is unlikely and undesirable that the same input data be sent to the RPU more than once. This approach may be viewed as a generalization of functional unit *chaining* in Cray vector units. Chaining permits the results of one vector computation to be forwarded directly to the input of another functional unit. This permits multiple functional units to operate in parallel and eliminates the overhead of storing intermediate results in either registers or memory.

Static architectures such as the Cray provide some small, fixed number of functional units which may be utilized in this manner. A reconfigurable machine may configure an arbitrary number of functional units in this manner. The number of operations performed in parallel is limited only by the parallelism available in the algorithm and by the physical size of the RPU. It is this internal forwarding of partial results that should permit the gains in performance that will allow reconfigurable logic based machines to compete with supercomputers.

While the memory system will not employ multi-level memory techniques such as caching and registers, other techniques may be used to improve memory performance. First, since the memory system is expected to be used to provide vector access to data, techniques such as memory interleaving are likely to provide improved memory system performance. Also, since the system is expected to make use of the mixed valued striding construct, hardware to implement this function should be part of the memory system.

Finally, the memory system should be able to read N data values and pass them to the RPU inputs as well as store M output results from the RPU per cycle. Each of these $M \times N$ data ports will contain hardware to perform mixed valued striding. The number and width of these data ports is one of the primary design parameters in the system. In many cases, the performance of the system will depend directly on the ability to provide the necessary memory bandwidth to the RPU. As with the RPU, these design parameters will not be specified at this time. Analysis of selected algorithms will indicate the general range of suitable values for these design parameters.

Chapter 7

Applications

Several algorithms have been selected for implementation and simulation for this reconfigurable machine. These algorithms have been selected as typical of those execution of existing parallel and vector supercomputers.

The first five algorithms selected are:

- Cellular Automata
- String Matching
- The Mandelbrot Set
- A Neural Network
- The Fourier Transform

In addition to these high level algorithms, selected portions of the Livermore FORTRAN Kernel (LFK) benchmark are also implemented and simulated. These kernels provide the means to study algorithmic structures, particularly those which cause difficulties on existing supercomputers.

Each of these algorithms is simulated directly from their high level language implementation. These simulations are used primarily to verify accuracy of the algorithms as they are implemented. From these verified algorithms, circuits based on the dataflow graph of the vector portion of the algorithms are extracted. Based on these circuits, performance estimates are made.

7.1 Cellular Automata

The cellular automata model of computation was originally developed by John von Neuman and Stanislaw Ulam. This model specifies a number of *cells* which exist in some typically small number of states. Cells transition between states based on the state of their neighbors. This highly parallel approach to computation has found many uses, ranging from random number generation to physical modeling and image processing [48, 121].

While an elegant computational model, cellular automata implementations can perform poorly on traditional processors. The regular structure and inherent parallelism of cellular automata, however, make it a prime candidate for data parallel implementation, and execution on a reconfigurable machine.

This section begins with a description and implementation of a simple linear cellular automata system. This is expanded to a two-dimensional array. Popular image processing applications based on this model are presented. Finally, a performance assessment is done.

7.1.1 Linear Cellular Automata

The linear cellular automata system consists of a single linear array of cells, which interact with neighboring cells. In this example, the cells can be in one of two states, either zero or one. Each cell interacts directly with only its two nearest neighbors. In this example, the interaction is the Exclusive-OR'ing of the state of the cell with the states of its two neighbors. Because the operation is repeated for each element in the array, a simple data parallel solution can be formed.

Figure 7.1 gives the data parallel implementation for the algorithm. A single vector A is used to produce the resultant vector *Out*. The neighboring values are generated by using the *stride()* function to produce two vectors with the elements shifted by one and two, respectively.

```
A1 = stride(A, 1, 1, N, N);
A2 = stride(A1, 1, 1, N, N);

Out = A ^ A1 ^ A2;
```

Figure 7.1: The data parallel code for the linear cellular automata.

Figure 7.2 shows the results of several iterations through this procedure. Each line in the image represents one pass through the algorithm. The output vector *Out* is then assigned to the input vector A and re-processed. The characteristic inverted triangle pattern of this algorithm is clearly visible.

The circuit extracted from this code is given in Figure 7.3. While other multi-state automata are possible, this example uses a simple two state cell. This makes all data in the circuit implementable using single bit values.

What is significant in this example is the structure of the circuit. A single vector is input, a single vector is output, and nearest neighbor operations are performed. Extensions to larger number of neighbors is accomplished by adding more delay stages. This can increase the complexity of the calculation

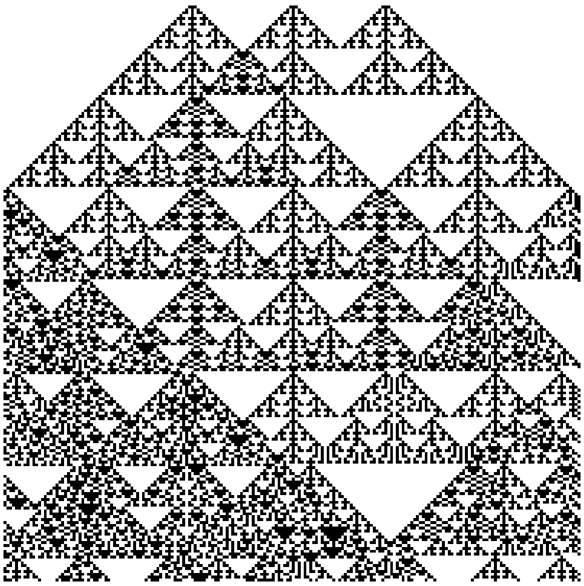


Figure 7.2: Output of the linear cellular automata implementation.

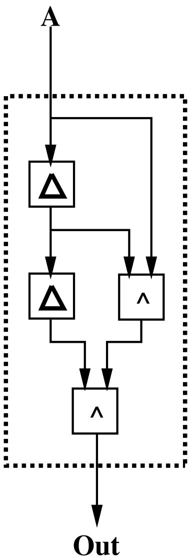


Figure 7.3: The linear cellular automata circuit.

without affecting the throughput.

Note that the *stride()* functions in the code have been used to produce the *delta* delay elements, rather than external memory references. This optimization reproduces the shifted version of the vector indicated by the *stride()* function. The addition of the *delta* functional units greatly reduces bandwidth while only marginally increasing circuit complexity. This operation is so common that a special function of the form *delta(X, N)* may be used in place of the *stride()*. This provides the same functionality while simplifying the code. It also may be used to provide a "hint" to the software that a delta element is the desired implementation.

7.1.2 Conway's Life

The one dimensional linear cellular automata system can be extended easily to two dimensions. In this two dimensional version, a cell is surrounded by eight neighbors. While operations involving neighbors not directly adjacent to the cell are possible, they will not be discussed here.

Perhaps the most popular cellular automata implementation for a two dimensional grid is Conway's game of *Life*. This early cellular automata simulation is based loosely on a model of biological cell growth. Like the linear cellular automata example, cells can be set to one of two states, "living" (1) or "dead" (0). A cell is "born" if there are exactly three living neighbors. If there are two living neighbors, the cell retains its present state. For all other values the cell dies, presumably from either starvation or overcrowding.

This simple set of rules can produce very complex patterns. These patterns have been shown to be capable of, among other things, performing general purpose computation. Poundstone provides a popular, but thorough, examination of Conway's *life*, as well as other related topics [105].

```
/* B starts in 2nd row */
B = stride(A, width, 1.0, (width*height), (width*height));

/* C starts in 3rd row */
C = stride(A, (2*width), 1.0, (width*height), (width*height));

A1 = delta(A, 1); /* 3 cells across top row */
A2 = delta(A1, 1);
B1 = delta(B, 1); /* 3 cells across 2nd row */
B2 = delta(B1, 1);
C1 = delta(C, 1); /* 3 cells across 3rd row */
C2 = delta(C1, 1);

/* Add all but center cell */
Sum = (A + A1 + A2) + (B + B2) + (C + C1 + C2);

/* Apply Conway rules */
Out = (Sum == 3) || ((Sum == 2) && (B1 == 1));
```

Figure 7.4: The data parallel code for *life*.

The data parallel code for this algorithm is shown in Figure 7.4. This code is an extension of the linear cellular automata implementation. Note that the simpler *delta()* notation is used in place of the *stride()* function in several places. A single vector A of length $(n \times m)$ is used to represent the state of the cells. While this vector is a linear array of values, it can be viewed as m concatenated vectors of length n, thus representing a two dimensional array.

A second vector B is generated directly from A. This vector is offset by *width* and represents data in the second row of the $3x3$ neighborhood. It should be emphasized that although B is defined as a vector variable, it only defines a method for accessing the data in vector A. It is not necessary to allocate storage for B, nor is it necessary to copy values from A into B. In a similar fashion, a third vector C is defined. This vector contains the data in A offset by *(2∗width)* and provides the third row of the neighborhood.

Now that the three rows in the neighborhood are defined by A, B and C, the individual elements of the neighborhood must be specified. As with linear cellular automata, two *delta()* functions are used by each row to produce the neighboring cells. These *delta()* functions produce the vectors *A1*, *A2*, *B1*, *B2*, *C1* and *C2*. These vectors, along with A, B and C represent the nine values in the three by three neighborhood.

Once this effort has been made to define the vectors for the calculation, the rest is a straight forward implementation of Conway's rules. In the data parallel code, *B1* represents the center cell in the neighborhood. The other vectors represent the neighboring cells. To calculate the new value of the *B1* cell, all of the other vectors are summed. If the sum is exactly three, the cell is set to living. Or if there are two living neighbors and the cell is currently alive, the cell remains living.

The result is written to the vector *Out*. The data parallel nature of the algorithm permits the operation to be defined using vector values just as if it were being defined for a single scalar calculation. This data parallel approach produces code which is in many ways simpler than comparable serial implementations.

The circuit extracted from this data parallel code is shown in Figure 7.5.

Note the similarities to the circuit for the one dimensional case in Figure 7.3. The circuit resembles three of the linear cellular automata circuits operating in parallel, performing additions rather than Exclusive-OR operations. The results of these sums are further processed to produce the single output.

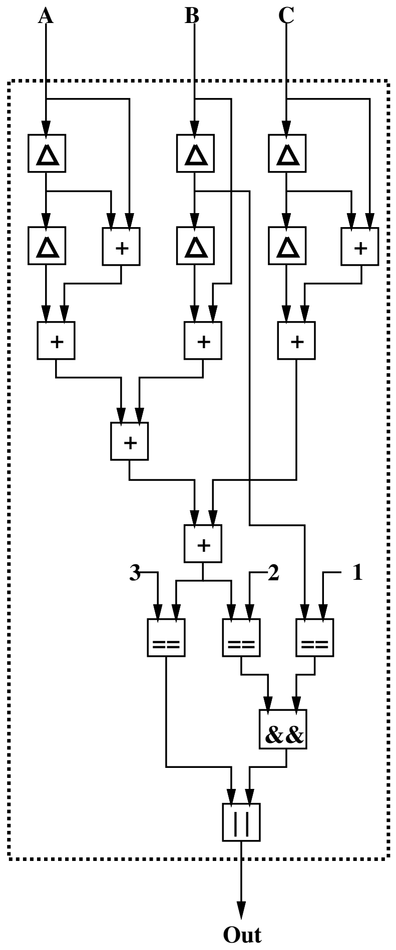


Figure 7.5: The extracted circuit for *life*.

Also note that despite all input and output values being boolean, multi-bit arithmetic operations are used to sum the elements in the neighborhood. The equality operators (==) are used to compare the multi-bit sum value to constants to produce a boolean values. These outputs are further processed by other boolean operators to produce the result. In most cases the output *Out* will serve as the input vector A for subsequent processing.

Figure 7.6 shows the evolution of a simple five cell block known informally as the *R-pentomino*. This structure is known to produce a large variety of patterns in Conway's *life*. It is also known to continue to produce unique patterns for several hundred iterations before converging on a fixed pattern.

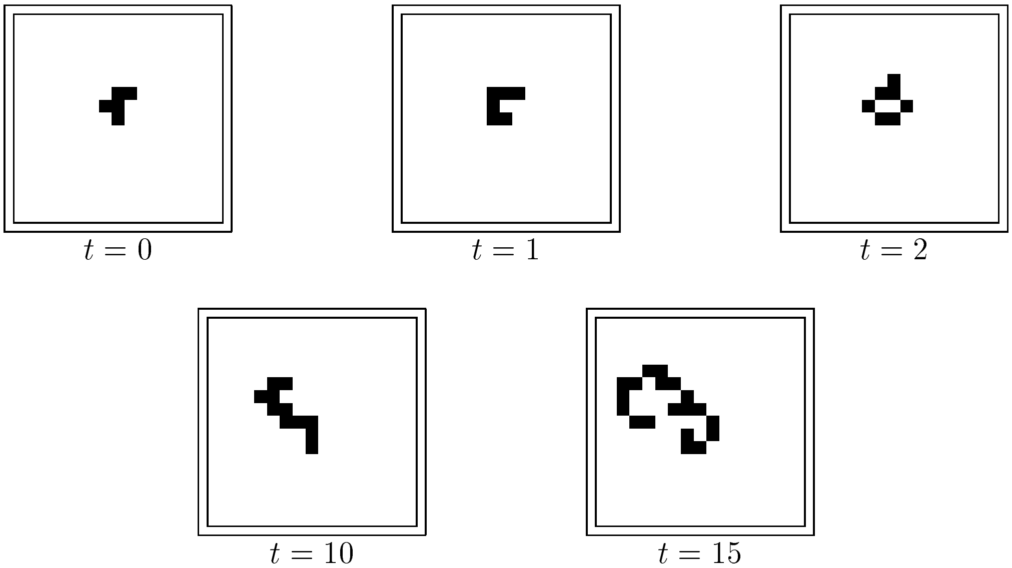


Figure 7.6: The evolution of a simple pattern in *life*.

At $t = 0$, the original R-pentomino is shown. After one iteration, at time $t = 1$, the r-pentomino takes on a hooked shape. At time $t = 3$, this changes to a closed oval object. The shape continues to grow, with the results at times $t = 10$ and $t = 15$ displayed in the figure. The object eventually splits into several pieces, finally producing more than 20 stable objects.

7.1.3 Image Processing

A simple extension of Conway's *life* is the basis for several popular digital image processing techniques. By performing operations on the pixels in a digitized image in a manner similar to *life*, useful and interesting functions may be performed.

As with Conway's *life*, these image processing techniques use the values of the neighboring cells or in this case, pixels, to perform the processing. Like the other cellular automata, these operations may be performed independently in parallel.

In these examples, all of the images are $256x256$ pixels, with each pixel containing an integer value from zero to 255. These values are displayed as a standard greyscale. This leads to a cellular automata grid that is $256x256$ with each cell assuming one of a possible 256 states. Unlike the previous examples, the operations performed on these cells will require at least eight bits of accuracy. Again, all neighborhoods are $3x3$ pixels. Larger neighborhoods are possible, but they are less common and are not discussed here.

All of the image processing operations are performed by multiplying each value in the $3x3$ cell by some fixed value, summing the results and dividing by some fixed quantity. The nine values which are used to scale the pixels are often referred to collectively as a *spatial mask*. The quantities are usually displayed as a $3x3$ array representing the relative pixels on which they will operate. Depending on the values in the spatial mask, various image processing functions are performed.

Figure 7.7 gives four popular masks which will be used in subsequent examples. Each performs a useful image processing function. The first mask *(a)* contains all ones and is used for image smoothing or blurring. The final result from this kernel should be divided by nine. An alternative representation of this kernel is one with all cells containing the fraction 1/9. The eliminates the need for the final scaling, but requires the representation of fractional values. Rather than use a fractional representation, all mask values are specified as integers. If necessary, a final scaling is performed. This scaling is typically accomplished by dividing the output by the constant sum of the nine mask values.

The second and third masks are known as *Sobel edge detectors*. These provide a gradient or first derivative operation which enhances changes in pixel values. Collectively, this tends to bring out "edges" in an image. The mask in *(b)* preferentially selects horizontal edges, while the mask in *(c)* selects vertical edges.

Where the *(b)* and *(c)* masks provide something of a first derivative, detecting edges, the last mask *(d)* performs a second derivative. This mask is often referred to as a *Laplacian*. Gonzalez and Wintz [43] contains other masks, as well as a discussion of this and other image processing techniques.

The data parallel code for making use of these masks is shown in Figure 7.8. This code is very similar to the two dimensional cellular automata implementation of Conway's *life*. In fact, the generation of the nine cell neighborhood vectors is identical. For this reason it is not be repeated here. The code listing begins after the A, B and C vectors have been defined.

In this code fragment, the values in the mask are represented by the

1	1	1
1	1	1
1	1	1

(a)

-1	-2	-1
0	0	0
1	2	1

(b)

-1	0	1
-2	0	2
-1	0	1

(c)

0	1	0
1	-4	1
0	1	0

(d)

Figure 7.7: Some 3 x 3 image processing masks.

```
Out =  ((A * m1) + (A1 * m2) + (A2 * m3) +
        (B * m4) + (B1 * m5) + (B2 * m6) +
        (C * m7) + (C1 * m8) + (C2 * m9)) / scale;

/* Clip to 0 - 255 */
Out =  max(Out, 0);
Out =  min(Out, 255);
```

Figure 7.8: The data parallel code for mask based image processing.

constants *m1* through *m9*. These are multiplied by the vectors representing the $3x3$ neighborhood and finally divided by some constant scale factor, *scale*.

While this produces the desired image, the last two operations, *max()* and *min()* make sure that the output values are between the desired values zero and 255. All values less than zero are mapped to zero and all values greater than 255 are mapped to 255.

Figure 7.9 gives the circuit extracted from the data parallel code in Figure 7.8. From an external point of view, the circuit is identical to the previous two dimensional cellular automata circuit. Three vector inputs, A, B and C produce a single output vector *Out*.

Internally, the circuit resembles the two dimensional cellular automata circuit. The major difference is that nine multiplier units are used to scale the masked values. The constant values *m1* through *m9* are input as the scale factors to this circuit.

Note that since these scale factors are typically small constant values, it is possible to replace the multipliers with circuits optimized for these constant operations. This approach is especially desirable when a single specific mask is used. If several masks are to be used, standard multipliers may be employed. The functionality can then be changed by simply modifying the values input to the multiplier units.

The images in Figure 7.10 show the results of the the smoothing kernel. Here the constant mask values *m1* through *m9* are all one. This essentially eliminates the effect of the multipliers, providing a simple sum of the $3x3$ neighborhood. This is identical to the function of the upper portion of the two

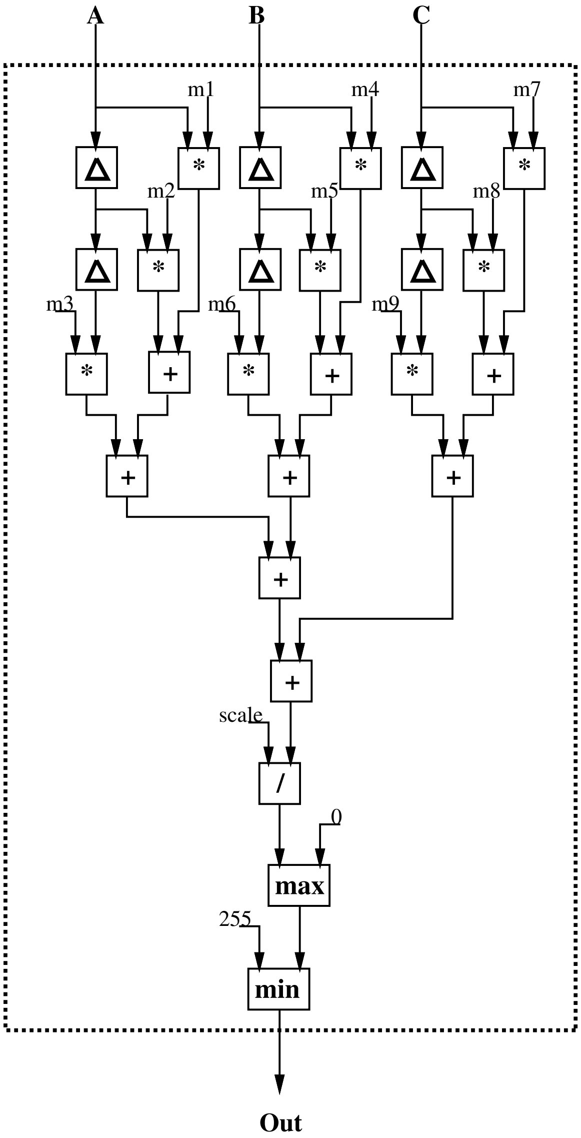


Figure 7.9: The extracted image processing circuit.

dimensional cellular automata circuit.

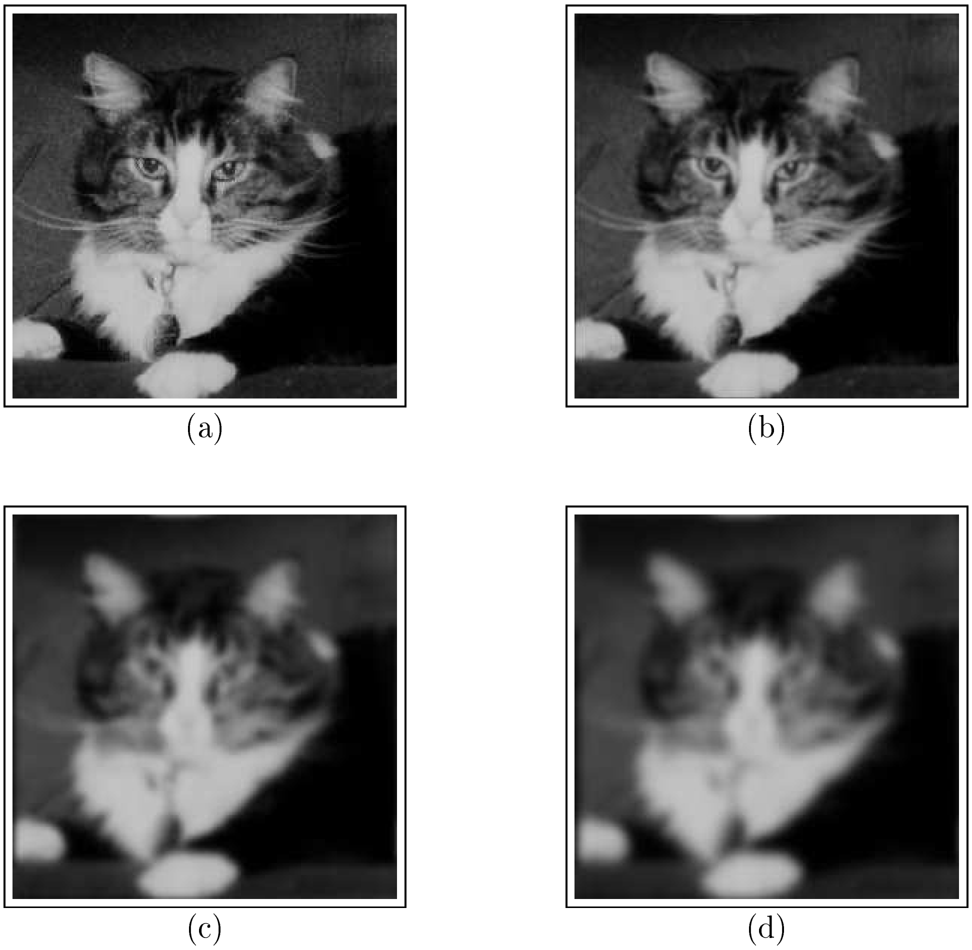


Figure 7.10: The effect of the smoothing operation.

After summation, the sum is divided by nine which should bring it back into the desired range of zero to 255. The *max* and *min* operators are nevertheless employed to ensure a valid pixel range. While not necessary for the smoothing operation, other operations may require this remapping into a valid range.

The images in Figure 7.10 show the effect of the smoothing operation.

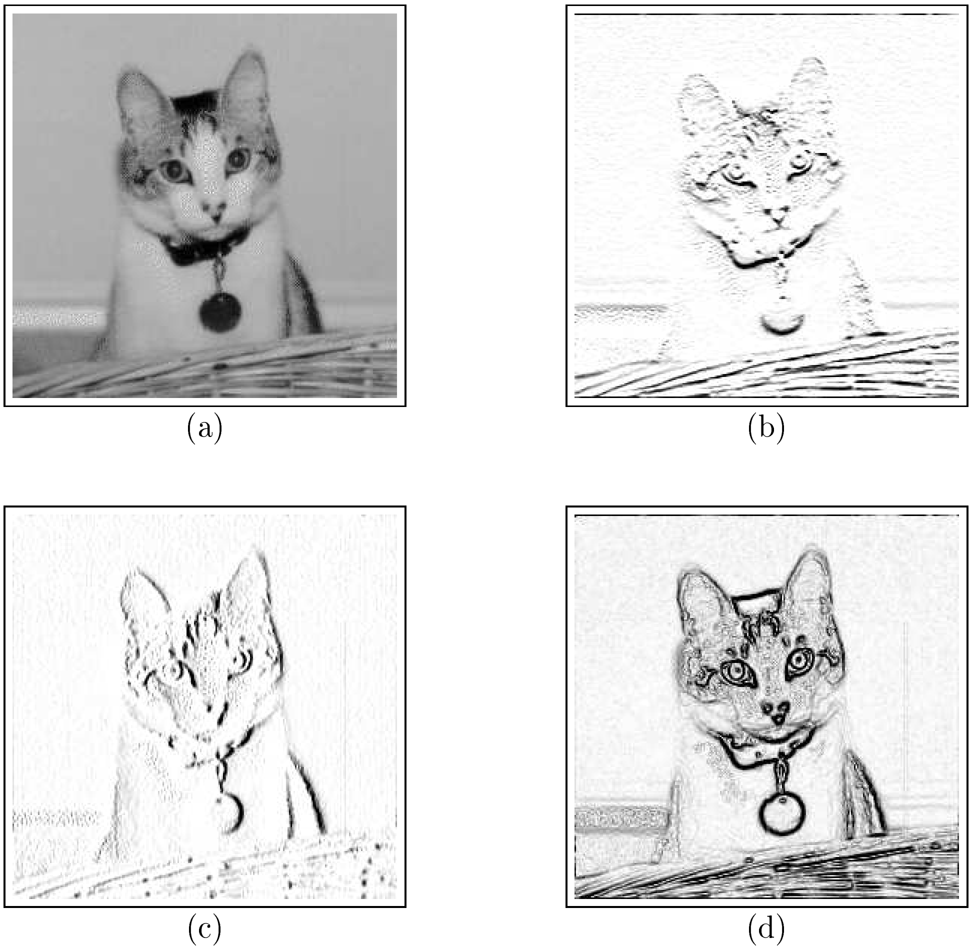
The first image *(a)* is the unprocessed image. The second image *(b)* is the original image after one smoothing operation. Some decrease in the sharpness of the image can be noticed. The images in *(c)* and *(d)* show the initial image processed 10 and 20 times, respectively, with the smoothing operator. The blurring of the image at this point is very noticeable.

A different type of operation is shown in Figure 7.10. Here, rather than blurring features in the image, features are enhanced. This set of images demonstrates the the ability of spatial masks to provide the outline of features in an image. This function is especially important for various image recognition algorithms.

The first image *(a)* in Figure 7.11 is the original unmodified image. The image in *(b)* was processed with a horizontal edge detection mask. Note that in the image, only the horizontal edges which are darker above and lighter below are prominent. This is because of the use of negative numbers by the mask. Edges which are lighter above and darker below produce large negative values which are mapped to zero.

The third image *(c)* shows the result of processing with the vertical edge detection mask. Note the prominence of vertical edges in the image. As with the horizontal edge detection function, the gradient operator can produce negative values. In this displayed image, they are again mapped to zero.

The last image *(d)* combines the horizontal and vertical edge detection operations to produce the complete two dimensional image gradient. Here both horizontal and vertical mask processing is performed, with the images being combined as the square root of the square of the pixel values. Note the distinct



(a) (b) (c) (d)

Figure 7.11: An edge detection example.

edges in this image. Figure 7.12 gives the data parallel code which implements this combined mask processing approach.

```
Horiz =  ((A * h1) + (A1 * h2) + (A2 * h3) +
         (B * h4) + (B1 * h5) + (B2 * h6) +
         (C * h7) + (C1 * h8) + (C2 * h9));

Vert =   ((A * v1) + (A1 * v2) + (A2 * v3) +
         (B * v4) + (B1 * v5) + (B2 * v6) +
         (C * v7) + (C1 * v8) + (C2 * v9));

Out =    sqrt((Horiz * Horiz) + (Vert * Vert));
```

Figure 7.12: The data parallel code for edge detection.

In Figure 7.12, the mask values for the horizontal edge detection mask are given by the values *h1* through *h9*. Similarly, the values of the vertical edge detection mask are given by the values *v1* through *v9*. The final output *Out* is given by the combination of the horizontal and vertical edge detected data. Again the vector definitions for A, B and C are omitted, as is the final mapping to values between zero and 255.

Figure 7.13 gives the circuit extracted from this code. For clarity the functions which produce the *Horiz* and *Vert* have been condensed into a single functional unit. The internal implementation of these functional units is identical to the upper portion of the original circuit in Figure 7.9. The scaling and the remapping to values between zero and 255 is not performed, however.

Again, this circuit takes in three input vectors and produces a single output. What is unique is that two image processing functions are performed in a single pass. While not possible for cascaded functions, functions which

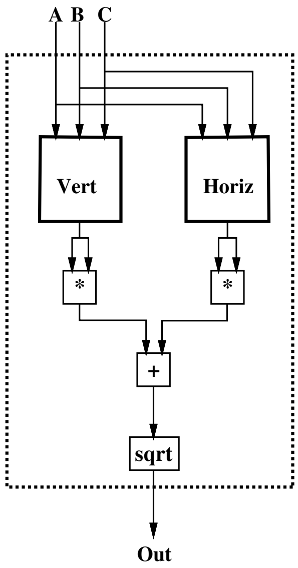


Figure 7.13: The edge detection circuit.

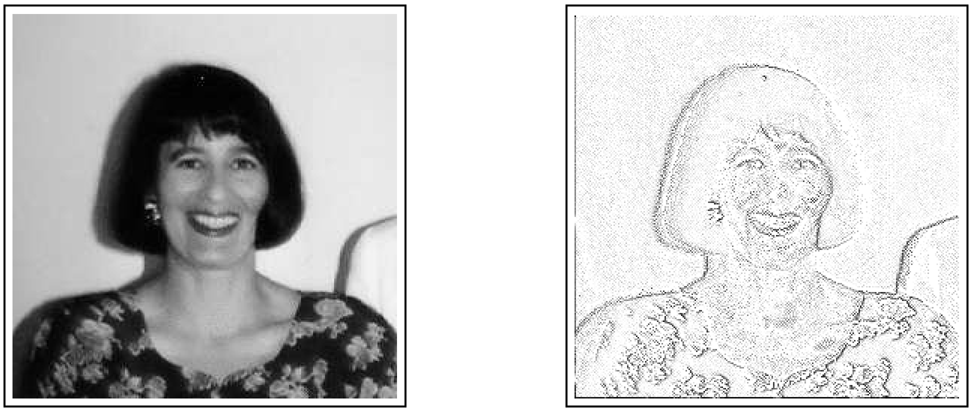
proceed in parallel, particularly those operating on the same image data, can be performed without impacting the input or output bandwidth of the circuit.

Also of consequence is the fidelity of the final output. In edge detection, some pixel values generated by the horizontal and vertical edge detection masks can be as much as four times the magnitude of the original pixels. The pixel values may also be positive or negative. If this processing is performed serially in stages, with intermediate images produced, much of the information can be lost. The reconfigurable processor permits wider internal datapaths to preserve the accuracy of the calculation. Only after all calculation has been performed will the pixel values be mapped to the valid range.

Figure 7.11 verifies this enhanced accuracy. Note that the final edge detected image in *(d)* contains more and sharper gradient information than the combination of the horizontal and vertical edge detected images in *(b)* and *(c)*.

In particular the negative valued gradients have been preserved and contribute to the calculation.

Finally, an example of processing using the Laplacian mask is shown in Figure 7.14. This mask also enhances gradients, much like the edge detection masks. Rather than taking the two dimensional first derivative, this mask takes the equivalent of a two dimensional second derivative. While this does detect edges, it also amplifies any noise in the image. Figure 7.14 verifies that both horizontal and vertical edges are indeed detected with this mask, but with more noise than in the previous example.



(a) (b)

Figure 7.14: Laplacian edge detection.

Several other spatial masks may be used to perform various image processing functions. These may be implemented by the circuits described in the section, with only the constant scaling inputs changed.

And since this also provides the functionality of a general two dimensional cellular automata with a $3x3$ neighborhood, this circuit can also be used to implement various multi-state cellular automata. Such automata are finding widespread use to simulate physical phenomena, including chemical reactions and turbulent fluid flow. It is expected that these applications will also be implemented efficiently on reconfigurable machines.

7.1.4 Performance

Table 7.1 compares the performance of four high performance architectures with the predicted performance for a reconfigurable system. The first machine in the table in the *CAM-6*, a machine dedicated to cellular automata applications. This machine is capable of producing a 256 by 256 array of cells updated at a rate of 60 Hz. This gives a total rate of approximately 4 *Million Cell Updates Per Second* or *MCUPS*. This rate is approximately the same as a *CRAY 1* supercomputer [87].

Machine	MCUPS
CAM-6	4
CRAY 1	4
CRAY 2/XMP	100
Reconf. Arch.	50

Table 7.1: Performance of cellular automata implementations.

Higher performance supercomputers, the *CRAY 2* and the *CRAY XMP*, simulate at a rate of approximately 100 MCUPS. A reconfigurable system running at 50 MHz is able to update approximately one cell per cycle. This gives an estimated rate of 50 MCUPS. This rate, while half that of the high-end supercomputers, requires substantially less hardware.

7.1.5 Other Related Work

Other work has been performed using reconfigurable hardware for cellular automata based physical simulations [91, 94, 87]. Similarly, reconfigurable logic has also been used in several recent image processing applications [14, 85, 107, 1, 115, 122].

Two factors contribute to make this application area well suited to implementation in reconfigurable logic. First, these algorithms exhibit a high degree of data parallelism. This permits large amounts of data to be processed by the reconfigurable logic, providing relatively large increases in performance.

Secondly, the calculations performed are relatively simple. Data paths in these applications are typically one to eight bits wide. The arithmetic and logical operations performed on this data are also relatively simple. Boolean operations are especially common. The simplicity of the calculations permit implementation with a very small amount of hardware.

These factors combine to make this a natural application area for reconfigurable systems. What has been demonstrated here is that these types of highly parallel algorithms can be easily specified in a high level language and translated directly into high performance pipelined circuits.

7.2 String Matching

Comparing the values in two strings is a common operation. Often it is desirable to know whether or not two strings match exactly. A more complicated, but often more important question is not whether or not two strings match exactly, but how closely they are matched.

The search for similarity in strings has many practical applications. Originally devised to detect and correct typographical errors for data entered into computers, string comparison techniques have been applied to applications involving such diverse areas as computer science, geology, chemistry and genetics [111]. All of these applications involve an attempt to match inexact data to some predefined template.

7.2.1 String Comparison

When comparing two strings for similarity, some metric must be used to gauge their similarity. One common metric is the *edit distance*. The edit distance is defined as the number of operations necessary to convert one string to another. In general, there are two operations used to convert one string to another. These are *deletion* and *insertion*. A third operation, *substitution*, can be viewed as a combination of a deletion and an insertion. These operations are performed just as one would suspect. A deletion removes a character from a string and an insertion add a character to the string. The *minimum* number of operations necessary to convert one string to another is the edit distance.

It is the notion that the edit distance must be the minimum number of edits that makes this comparison difficult. Because there are many ways to convert one string to another, some method for tallying these operations and their cumulative effect must be employed.

As a brief example, consider the strings “mail” and “male”. There are many ways to convert one to the other using insertions and deletions. The trivial approach is to delete each letter in the source string and insert each letter in the destination string. While a valid approach, and perhaps the minimum possible edit, this approach is guaranteed to take $2N$ operations, where N is the length of the string.

A second approach in this example would be to only insert and delete the last two characters. This reduces the edit distance from eight to four. Another approach is possible, that of deleting the "i" in the source string and inserting a "e" at the end. With two edits, the string has been converted from the source to the destination.

7.2.2 A Dynamic Programming Algorithm

In the early 1970s, at least nine independent discoveries of a dynamic programming algorithm for computing the minimum edit distance were published. These algorithms were published in journals in the U.S.S.R., the U.S.A., Japan, Canada and France in fields as diverse as computer science, molecular biology and speech processing [111, 128].

The algorithm for finding the minimum edit distance between two strings involves the computation and storage of values in a two dimensional table. The number of elements in this table will be the product of the lengths of the two strings.

Each element in the table is computed based on three previously computed values. Figure 7.15 gives the relative location in the table of the values used in the calculation. The values a, b, c are the previously computed values and are used to calculate the value of d.

Because the algorithm depends on the previously computed values, the calculation proceeds from the upper left corner of the array to the lower right.

$$
\begin{array}{c|ccc}
 & & & T_j \\
\hline
 & & & \vdots \\
 & & a & b \\
S_i & \cdots & c & d
\end{array}
$$

Figure 7.15: The values used to calculate d.

This propagates computed values from the upper and left portion of the array to the lower right.

Equation 7.1 gives the operation performed on a, b and c to produce a value for d. In this case, it is assumed the cost of each edit operation is unity. In the general case, a different cost may be assigned to an insertion and a deletion, depending on the matching criteria.

$$
d = min \begin{cases} \begin{cases} a & \text{if } S_i = T_j \\ a + 2 & \text{if } S_i \neq T_j \end{cases} \\ b + 1 \\ c + 1 \end{cases} \tag{7.1}
$$

This process continues until each character in the source string is compared to each character in the destination string. For a source string of length N and a destination string of length M, this produces a table of NxM values. The minimum edit distance is the last value computed, the value in the lower rightmost corner of the array.

Given the example for the strings "male" and "mail", the table in Figure 7.16 is constructed. While at first the dynamic programming algorithm may be less than intuitive, seeing a table such as the one in Figure 7.16 helps to explain the technique.

First, the first row and first column of the array are set to increasing

		m	a	i	l
	0	1	2	3	4
m	1	0	1	2	3
a	2	1	0	1	2
l	3	2	1	2	1
e	4	3	2	3	**2**

Figure 7.16: The values used to calculate d.

integral values, starting with zero. This permits the calculation to begin with the values *0* for a, *1* for b and *1* for c. In this case, since the first two characters in the string match, the value *0* is propagated diagonally downward from a.

Note that for two matching strings, this *0* is propagated all the way down the main diagonal, producing the final string edit distance of zero, as expected. Other values in the table keep track of other possible edits of the source string.

The path taken by the propagated values that eventually ends at the lower rightmost corner of the array represents the edits performed to transform the string. Note that there may be several such paths. This indicates that more than one edit sequence could produce the destination string from the source in some minimum number of edits.

7.2.3 Searching Genetic Databases

While this string matching technique has found a large number of applications, one particular application, the matching of genetic information, has led to an increased interest in this area [131].

By the mid-1980s, the technology used to identify genetic sequences had progressed to the point where an ambitious research effort known as the *Human Genome Project* was launched [124]. The goal of this project was to collect the existing genetic sequences mapped by researchers and place them in a common database. The goal was to eventually catalog the entire human genome. This database would catalog approximately 3 billion ($3x10^9$) pieces of information. This amount of information, if printed, would require approximately 200 volumes with 1000 pages each.

While a large database, the difficulty arises in that the searches are not exact. The desire to find similar, not exact, sequences requires that the algorithms such as the dynamic programming algorithm used for string comparison be employed.

Unfortunately, this algorithm has a computational complexity of $O(nm)$ for two strings of different length, or simply $O(n^2)$ for strings of the same length. This quadratic complexity is manageable for smaller strings, but makes comparison of larger strings unfeasible using traditional techniques. Vector supercomputers, parallel supercomputers, custom hardware [78] [79] [81] and reconfigurable systems [56] have been employed in the search of these genetic databases.

7.2.4 A Parallel Implementation

While the complexity of the dynamic programming algorithm for string matching is quadratic, one approach to managing this complexity is to use multiple processors. Unfortunately, the structure of the computation limits the amount of available parallelism. Each computed value d depends on three previously computed values in the preceding row and column. Because the values are computed recursively, this algorithm can neither be parallelized nor vectorized along rows or columns.

At first, it may seem that a parallel prefix operation in the form of a *min-scan()* could be employed to implement this algorithm. Unfortunately, the combination of the increment operation and the *min* operation make this calculation impossible to implement using this approach.

Fortunately, while there are dependencies across rows and columns, calculations along the lower-left to upper-right diagonals in the matrix are independent and may be calculated in parallel. In order to take advantage of these independent calculations, these diagonal vectors must be identified and defined. Rather than attempting to define these diagonals based on the square matrix, a slight transformation of the algorithm will be performed.

Figure 7.17 shows a second example of a table generated by the dynamic programming algorithm. In this case, the data in the two strings consist of the characters *a*, *c*, *g* and *t*. These represent the four nitrogenous bases, adenine, cystosine, guanine and thymine which describe a DNA sequence.

		t	g	g	a
	0	1	2	3	4
a	1	2	3	4	3
c	2	3	4	5	4
g	3	4	3	4	5
g	4	5	4	3	4

Figure 7.17: An example table.

In order to vectorize the algorithm, the table containing the data is skewed downward. This shift removes one of the data dependencies and permits the algorithm to be vectorized across rows.

Figure 7.18 gives the modified table showing the relation of the values calculated by the algorithm. Note that this representation does not alter the algorithm itself. Only the data representation is re-oriented.

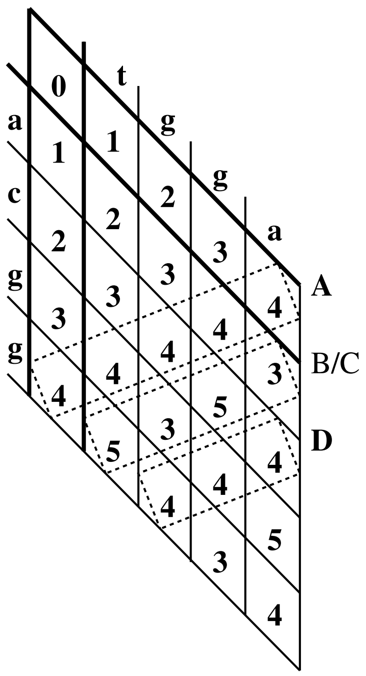


Figure 7.18: The table realigned for vectorization.

To generate this skewed table, the values of a, b, c and d are similarly reorganized. Rather than the square in the original diagram, the values now form a parallelogram as in Figure 7.19. As in the previous approach, the values are calculated from the upper left to the lower right corner of the matrix. The final minimum edit distance is again found in the lower right hand corner of the matrix.

$$\begin{array}{c|ccc} & & & T_j \\ \hline & & & \vdots \\ & & \mathrm{a} & \\ & & \mathrm{c} & \mathrm{b} \\ S_i & \cdots & & \mathrm{d} \end{array}$$

Figure 7.19: The values new used to calculate d.

7.2.5 Conditionals

The string matching algorithm is not yet implementable using the existing programming model. Up to this point, only arithmetic and logical operations on vectors have been performed. The string matching calculation requires a comparison and a conditional calculation.

Athanas describes a method for implementing the *IF* statement from the *C* language. The statement is mapped onto multiplexer or MUX based circuits [6]. This approach was used to produce combinational circuits. An extension of this method to the data-parallel programming model permits conditional code statements to be translated into pipelined circuits for vector computations.

In the conditional statement in Figure 7.20, vector x is assigned to some function $f(x)$ when the clause *cond* is *true*. In an instruction set architecture, this code represents a control structure which would generate a comparison and a branch instruction. If *cond* is *false*, the assignment statement will simply not be executed.

```
if (<cond>)
   x = f(x);
```

Figure 7.20: A conditional statement.

Translating this code statement into a digital circuit, particularly a pipelined circuit, presents some problems. First, the dataflow graph representation of the code must be extended to account for the conditionals. In addition, some technique for converting these graph structures to circuits must be specified.

If a MUX-based solution is considered, it is possible to consider the MUX a macrocell, much like the other arithmetic and logical macrocells. The MUX macrocell, however, takes three, rather than two inputs. Two of these inputs will be data inputs corresponding to the *true* and *false* cases. The third input will be a single bit for the select line.

Using this approach, two distinct alternatives must be supplied for each conditional clause. This approach introduced here will be referred to as the *dual assignment rule*. For each value assigned in the *if* clause of a conditional statement, there must also be a corresponding assignment in the *else* clause. These values are calculated in parallel and selected appropriately. In Figure 7.20, the lack of an *else* clause implies an identity assignment. The implied code is shown if Figure 7.21.

```
if (<cond>)
   x = f(x);
   else
      x = x;
```

Figure 7.21: The implied dual assignment.

This implied assignment supplies the MUX with its two necessary inputs permitting a valid dataflow graph to be constructed. A circuit for this code fragment is shown in Figure 7.22. It should be noted that the *select* input to the MUX is necessarily boolean. This input is typically formed by using a comparison operator.

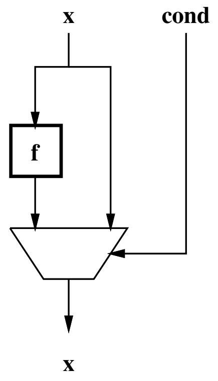


Figure 7.22: The conditional circuit.

7.2.6 The Data Parallel Implementation

With the ability to perform conditional calculation and the vectorizable representation of the string matching algorithm, data parallel code implementing this algorithm may now be written. The portion of the code which performs the calculation is a direct translation of the arithmetic formula in Equation 7.1. Figure 7.23 shows this calculation.

```
if (S == T)
   A1 = A;
   else
      A1 = A + 2;

M = min(A1, (B + 1));
D = min(M, (C + 1));
```

Figure 7.23: The data parallel code for the string matching algorithm.

There are five input vectors to the algorithm, A, B and C, and the strings T and S. A starts as the top row of the matrix. Similarly, C is the second row. B is the same data in the C vector, but shifted by one. These input values produce the single output vector D.

After the D vector is output, the process is repeated, with the new A vector being the previous C vector, and the newly calculated D vector becoming the C vector. This process continues downward in the array $((2 * n) - 1)$ times, until the minimum edit distance is generated. Figure 7.24 shows the circuit extracted from this data parallel code.

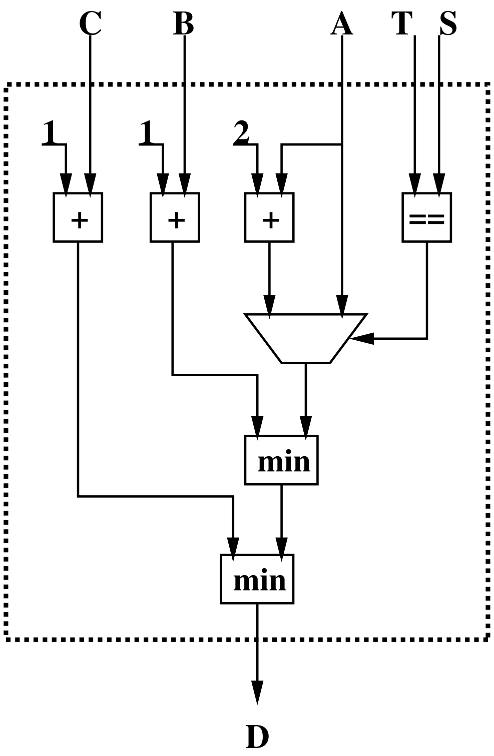


Figure 7.24: The circuit for the gene matching algorithm.

The code in Figure 7.23 gives only the arithmetic and logical vector operations in the calculation. The other control information, including the definition of the input vectors, is not included in this source code listing.

Some implementation details describing the definition of the vectors should be mentioned. First, the integer values in the top diagonal and the left edge are computed along with the other values in the algorithm. The vectors are all padded with two extra values, one at the beginning of the vector and one at the end. These values are initialized to some value greater than zero. This permits the first row and column of sequential integers to be generated on the fly. Rather than use this trick, it would be possible to have the host processor increment these values. Whether this would be the most efficient approach will be implementation dependent.

Also, the generation of the string vectors S and T is not shown in the code. Recall that each character in S must be compared to each character in T. This is accomplished by fixing the T vector, and comparing it to a "rotated" version of the S vector on each iteration. This rotation involves shifting all of the elements in the S vector one step to the right. This is easily accomplished by redefining the vector using a *stride()* operation.

7.2.7 Performance

Discounting latency and other overheads, this circuit is capable of performing one comparison per cycle. For a circuit operating at 50 MHz, this performs approximately 50 millions comparisons per second.

Hoang quotes performance levels of approximately one million comparisons per second for workstation class uniprocessors [56]. A Connection Machine with 16K processors achieves less than 6 million comparisons per second.

In an implementation of this algorithm using reconfigurable hardware Hoang achieves a maximum of 43,000 million comparisons per second. This implementation uses a relatively large 16 board reconfigurable system. A smaller system using a single board achieves 370 million comparisons per second [56].

The data parallel approach is superior to conventional uniprocessors by a large factor, and even superior to large parallel machines for this algorithm. Curiously, however, similar reconfigurable hardware is able to provide orders of magnitude higher performance. The reason for this large gap in performance has to do with the structure of the algorithm. The regular flow of data in the dynamic programming algorithm, as well as the simple operations performed on the data make this algorithm an ideal candidate for a systolic implementation [73, 74, 13]. In this systolic implementation, hundreds or even thousands of small special purpose computational units work in unison on the problem. This systolic implementation provides a high performance, low bandwidth, multiprocessor implementation of the algorithm.

Unfortunately, the data parallel approach is not geared toward exploiting this type of parallelism. For algorithms with this structure, further performance gains are possible. These gains can best be achieved using a programming model which supports the systolic framework. A second alternative is to use a circuit design methodology to produce a custom systolic array. This approach can be likened to using assembly language to increase the performance of an algorithm on a conventional serial processor. In the case of the reconfigurable processor, however, the gains may be several orders of magnitude.

7.3 The Mandelbrot Set

The algorithm to calculate the Mandelbrot set is a popular computationally intensive algorithm. The ubiquitousness of this algorithm has made it something of a benchmark for high performance systems. Much of the popularity of this algorithm can probably be attributed to the interesting bitmaps it generates.

The formula for the algorithm is very simple. A single quantity is recursively calculated using the formula in Equation 7.2.

$$z = z^2 + c \tag{7.2}$$

In this equation, z and c are both complex numbers. Initially, z is set to zero, and c is set to some initial condition. The value of z is then iteratively calculated. Since this calculation is non-linear, the value of z can be expected to either remain within fixed bounds, or diverge toward infinity.

When calculating the Mandelbrot set, the quantity of interest is not the actual value of z, but rather the number of iterations taken before the equation diverges. It is known that when the magnitude of either the real or the imaginary portion of z becomes greater than 2, the values will diverge toward infinity. The number of iterations taken to reach this point of divergence is the quantity of interest.

In most implementations of the Mandelbrot set algorithm, several initial conditions are calculated together. These points are usually equally spaced within the complex plane. The iterations counts are then plotted graphically, with individual display pixels corresponding to the points in the complex plane.

Since each calculation is independent, there is a large amount of parallelism exploitable in the algorithm.

7.3.1 Functional Decomposition

The calculation of the Mandelbrot set makes extensive use of complex arithmetic. In the complex number system, values are represented as pairs of numbers describing the real and imaginary components.

In the vector model of computation, all quantities are linear arrays of contiguous values. To perform calculations using complex values, a complex vector data type must be specified. This data type will consist of two standard vector data types grouped into a *C*-like structure.

The two complex operations used in the calculation, addition and multiplication, must also be specified in terms of existing arithmetic or logical operations. Complex addition is fairly simple. The real and imaginary parts of the vector are added, respectively. The data parallel code for this function is shown in Figure 7.25. This code uses *C++* style operator overloading.

A digital circuit can be extracted from this function. This circuit is derived from the dataflow graph of the code. Since there are two addition operations, and both are independent, the circuit is fairly simple. Figure 7.26 shows the extracted circuit from this function. Using a standard addition macrocell, a new complex addition macrocell is created. This new cell can be used in conjunction with existing macrocells. It may even be used in the construction of other, more complicated macrocells.

In a similar fashion, data parallel code can be written to multiply two

```
/* A Complex vector */
struct Complex {
          float  Re[N];
          float  Im[N];
          };

/* Complex addition */
Complex "+"(Complex A, Complex B) {
   Complex Sum;

   Sum.Re = A.Re + B.Re;
   Sum.Im = A.Im + B.Im;

   return (Sum);
   };  /* end "+" */
```

Figure 7.25: The code for complex addition.

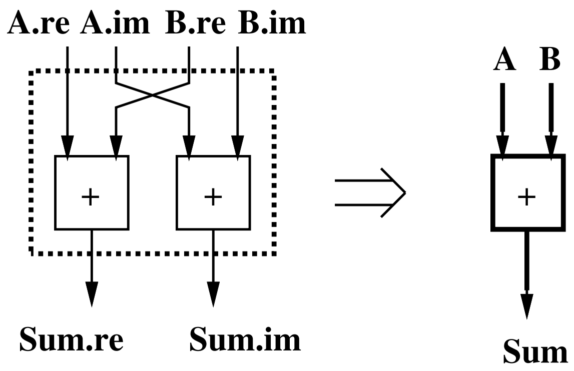


Figure 7.26: The complex addition circuit.

complex vectors. Note that the operation required is actually a square of the value of z. Rather than implement the special case of a squarer, the more general multiplication unit is implemented. The code used to implement complex multiplication is shown in Figure 7.27.

```
/* Complex multiplication */
Complex "*"(Complex A, Complex B) {
   Complex Prod;

   Prod.Re = (A.Re * B.Re) - (A.Im * B.Im);
   Prod.Im = (A.Im * B.Re) + (A.Re * B.Im);

   return (Prod);
   };  /* end "*" */
```

Figure 7.27: Code for complex multiplication.

As with the addition operation, the dataflow graph of the code may be constructed and used to produce a digital circuit implementation of the operation. In this case, four multiplications, an addition and a subtraction are performed. The circuit in Figure 7.28 is extracted from the dataflow graph of the code.

7.3.2 The Initial Condition Vector

With the necessary support for complex arithmetic in place, a data parallel description of the algorithm can be written. The first issue concerns the values in the initial condition vector, c. This vector will contain the real and imaginary values representing points in the complex plane. The simplest approach to supplying these values would be to consider this vector a static constant that is initialized before calculation begins.

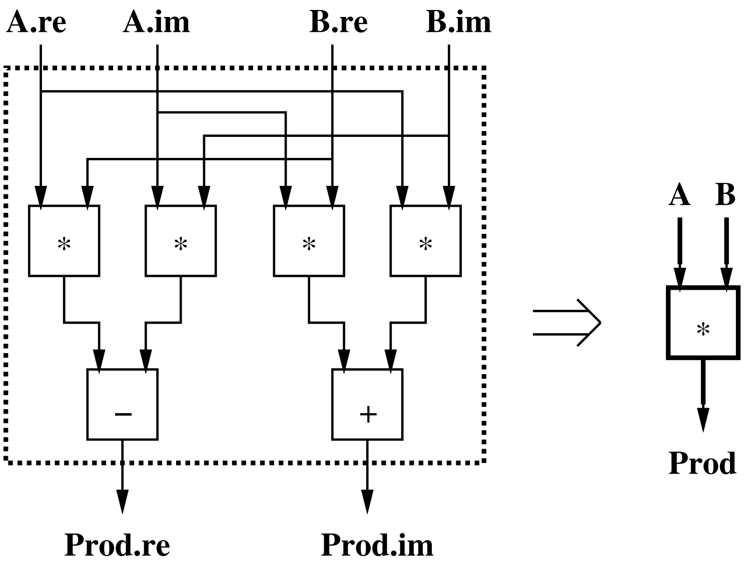


Figure 7.28: The complex multiplication circuit.

While a simple alternative, these points would still have to be calculated, perhaps off-line, by some host machine. It is desirable to calculate these values using the reconfigurable hardware.

The values to be generated are points in the complex plane. If the X-axis is considered to represent the real values, and the Y-axis imaginary values, the problem is just one of producing evenly spaced (X, Y) points in this plane.

Since the values used are real and imaginary vectors, data parallel operations on each value in the vector must be used. One approach to producing the desired vectors is:

1. Declare a vector of length $(X\_SIZE * Y\_SIZE)$
2. Generate non-negative integer points in the plane
3. Scale the values

4. Offset to the proper coordinates

The code which uses this approach to generate the c vector is shown in Figure 7.29. For simplicity, each line of code in Figure 7.29 performs a single operation. It should be noted that since only two quantities are being calculated, it is possible to merge the 12 lines of code in the example into two somewhat more complex lines of code.

First, in generating the non-negative integer values, the parallel prefix *add-scan()* operation is employed. Along with a constant initialization and a constant subtraction, this operator is used to produce vectors which range from 0 to $(X\_SIZE * Y\_SIZE)$. The modulus operator (%) is used to produce the real portion of the vector, breaking the vector into Y_SIZE segments with values running from 0 to X_SIZE.

In a similar fashion, the integer division operator (/) is used to break the imaginary vector into Y_SIZE segments containing all 0s in the first segment, 1 in the next segment, etc ...

The scaling and translation is accomplished using a constant multiplication and an addition. The technique used is the same for both real and imaginary portions of the c vector.

Employing the functional decomposition technique that was used by the complex arithmetic circuits, a circuit for producing the complex vector c can also be constructed. Figure 7.30 shows a diagram of this circuit. Note that only constant values are used as circuit inputs. These constants may be integrated directly into the circuit, rather than input as vectors.

```
/* Calculate real portion of vector */
c.re = 1;                  /* [1, 1, 1, 1, ...] */
c.re = add-scan(c.re);     /* [1, 2, 3, 4, ...] */
c.re = c.re - 1;           /* [0, 1, 2, 3, ...] */
c.re = c.re % X_SIZE;      /* [0, 1, 2, 3, ... 99, */
                           /* [0, 1, 2, 3, ... 99, */
                           /* ... */
                           /* [0, 1, 2, 3, ... 99] */

/* Calculate imaginary portion of vector */
c.im = 1;                  /* [1, 1, 1, 1, ...] */
c.im = add-scan(c.im);     /* [1, 2, 3, 4, ...] */
c.im = c.im - 1;           /* [0, 1, 2, 3, ...] */
c.im = c.im / Y_SIZE;      /* [0, 0, 0, 0, ... 0, */
                           /* [1, 1, 1, 1, ... 1, */
                           /* ... */
                           /* [99, 99, 99, ... 99] */

/* Scale real */
c.re = c.re * X_SCALE;     /* [0.00, 0.02, ... 1.98, */
                           /* ... */
                           /* [0.00, 0.02, ... 1.98] */
c.re = c.re + X_START;     /* [-1.00, -0.98, ... 0.98, */
                           /* ... */
                           /* [-1.00, -0.98, ... 0.98] */

/* Scale imaginary */
c.im = c.im * Y_SCALE;     /* [0.00, 0.00, ... 0.00, */
                           /* ... */
                           /* [1.98, 1.98, ... 1.98] */
c.im = c.im + Y_START;     /* [-1.00, -1.00, ... -1.00, */
                           /* ... */
                           /* [0.98, 0.98, ... 0.98] */
```

Figure 7.29: Code for the initial condition vector.

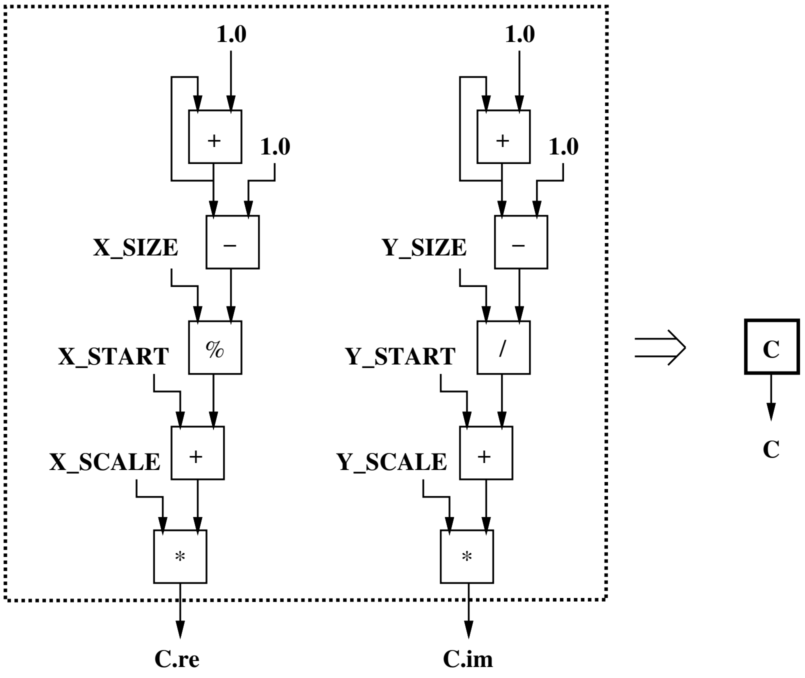


Figure 7.30: The initial condition circuit.

As with the complex arithmetic operators, this code may be packaged as a function and used as a macrocell. Note that the *c* macrocell has no inputs but outputs a new complex value with each clock cycle. This lack of inputs reduces the required bandwidth of the final circuit.

7.3.3 The Calculation

With the code and corresponding circuits for complex operators and conditional statements available, the implementation of the algorithm can proceed. Figure 7.31 shows the code used to implement the Mandelbrot set.

As in the other examples, this data parallel code is used to produce a pipelined digital circuit. Figure 7.32 shows the final circuit extracted from the dataflow graph of the code. Note that the previously defined "*c*" and complex

```
if (z < 4.0) {
   z = (z * z) + c;
   pixel = pixel + 1;
   }
```

Figure 7.31: The code to calculate the Mandelbrot set.

arithmetic operators are used in this circuit.

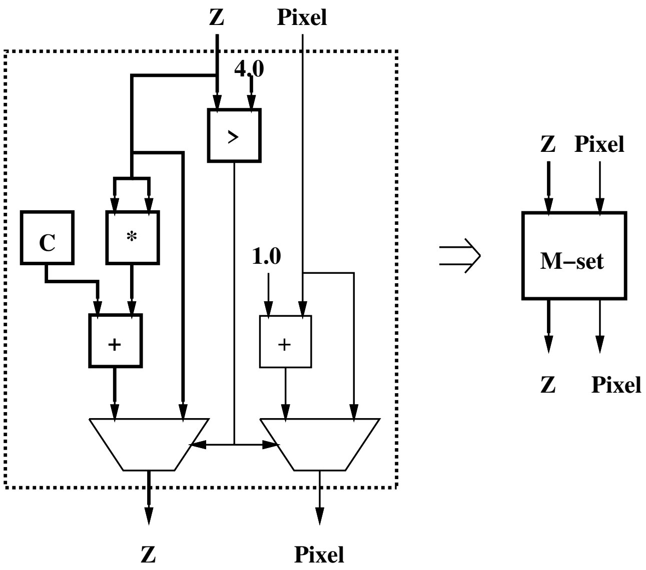


Figure 7.32: The Mandelbrot circuit.

The final circuit has very low bandwidth requirements. A complex vector z and a vector containing corresponding pixel values is input to the circuit. The updated complex vector z and pixel values are output. A total of three numeric values are input to the circuit and three numeric values output per clock cycle. During this clock cycle, approximately 20 arithmetic operations are performed.

The simulated output of this circuit is reproduced in Figure 7.33. For monochrome reproduction, the image below was produced by thresholding an 8-bit bitmap. The familiar outline of the Mandelbrot set is clearly visible.

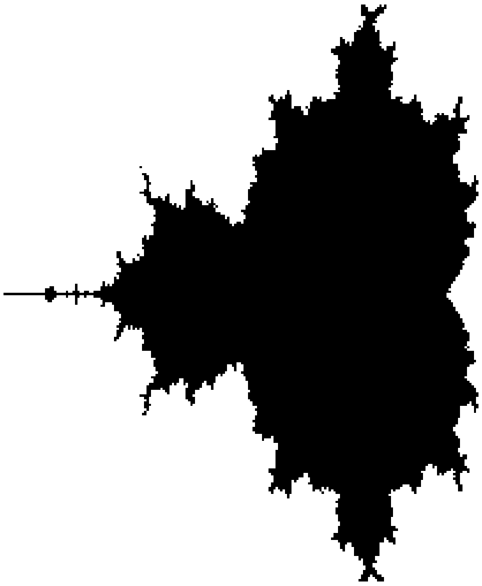


Figure 7.33: The Mandelbrot set.

7.3.4 Circuit Complexity and Performance

As with other circuits produced using this technique, one result is output per clock cycle. At 50 MHz, 50 million pixels are updated per second. Note that this technique produces successive approximations to the Mandelbrot set. In approximately 1 second, 50 iterations of a $1Kx1K$ bitmap can be processed. Including MUXes, this circuit uses 24 functional units. At 50 MHz, this corresponds to 1200 million operations per second (MOPS).

7.4 Neural Networks

Neural networks represent a biologically inspired model of computation whose history is as long as that of digital logic [89]. While these models are often used by researchers to study the activity of actual biological systems, they have also become a popular method of performing computation. The tasks performed by these networks tend to be tasks which are ill-suited to traditional algorithmic approaches. The neural network model has been used successfully in various pattern recognition tasks, including object identification and handwriting recognition.

While a useful algorithm for many computationally difficult tasks, the neural network model is itself computationally intensive. While it exhibits large amounts of parallelism, execution on general purpose hardware is still typically slow.

7.4.1 The Neural Network Model

Several different neural network models are currently in use. Lippmann gives a good overview of these models [77]. Hush and Horne update Lippman's overview and provide a large bibliography of more recent work in the area [57].

The particular model considered here was popularized by the work of Rumelhart, McClelland, Hinton and others [109, 88]. The network model is shown in Figure 7.34. This model contains three layers of neurons. These layers are referred to as the *input* layer, the *hidden* layer and the *output* layer. Data flows from the input neurons, through interconnections to the hidden neurons, then through interconnections to the output neurons. The path through the network is feed-forward and layered.

Each neuron broadcasts its output value to all neurons in the next layer. These output values pass to the next layer of neurons via *weighted* connections.

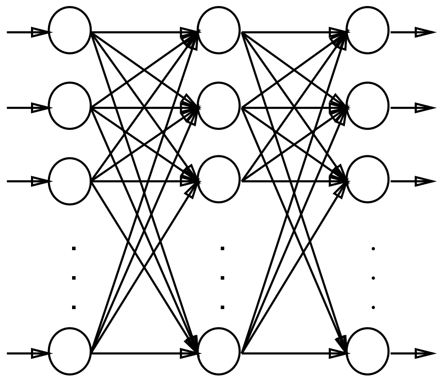


Figure 7.34: A three layer feed-forward network.

These weighted connections amplify or attenuate the output values before they are input to the neurons in the next layer. The weighted values are summed, processed by some limiting function, then output to the next layer.

The values of the weights for the interconnections define the behavior of the network. Automated techniques exist that allow networks to "learn" the values of these weights. For a set of inputs $A = (a_1, a_2, a_3, ...)$, a set of associated weights $W = (w_1, w_2, w_3, ...)$ and a limiting function $f(x)$ we can describe the behavior of a neuron with Equation 7.3.

$$output = f\left(\sum_i a_i w_i\right) \tag{7.3}$$

Despite this simple representation, calculating the output of this network is a computationally intensive problem. Each weighted interconnection in the network requires a multiplication and an addition operation. In a fully connected network with I inputs, H hidden units and O outputs, the number of interconnections is $(I \times H) \times (H \times O)$. To calculate the outputs of this network, $(I \times H^2 \times O)$ multiplications and additions must be performed as well as $(I + H + O)$ limiting functions $f(x)$. This gives an overall computational complexity of $O(n^2)$.

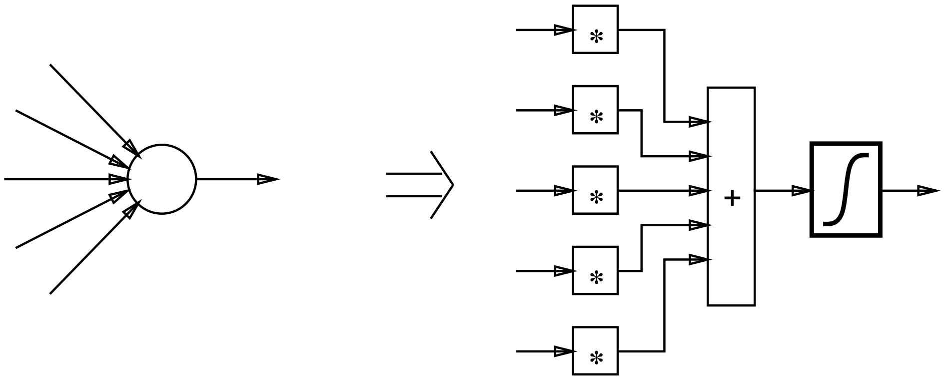


Figure 7.35: A digital representation of a neuron.

Figure 7.35 shows a direct digital hardware implementation of a neuron. For a neuron with N inputs, N multipliers, $N-1$ adders and the hardware to implement the limiting function, $f(x)$, are required. For networks containing a large number of neurons, this direct hardware implementation quickly becomes impractical.

7.4.2 A Vector Representation

The neural network model contains a large amount of parallelism. From the model, it is clear that all weighted inputs in a layer can be calculated concurrently. While this is possible, the number of hardware multipliers necessary to perform this task make this approach impractical for all but the smallest networks. This parallelism, however, may also be expressed in vector form. From this vector representation an efficient custom circuit can be extracted.

A vector representation of the network is shown in Figure 7.36. This figure shows a small network used to implement the Exclusive-OR function.

The values of the inputs, outputs and weights are grouped into vectors.

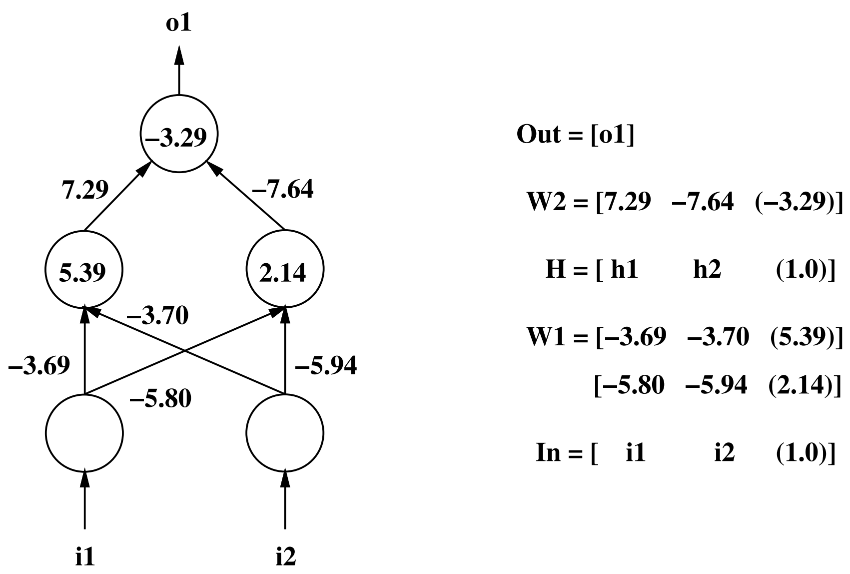


Figure 7.36: An Exclusive-OR network.

The interconnection weights between the input and hidden layers are stored in the variable *W1*. This variable is used to represent a matrix of values, but is stored as a single vector. The length of this vector is equal to the number of input units plus one, multiplied by the number of hidden neurons. The additional value associated with the input layer is for the neuron *offset*. The offset can be viewed as a weight connected to an input whose value is always '1'.

Similarly, the weights between the hidden layer and the output layer are stored in the vector *W2*. The length of this vector, like that of the *W1* vector, is the number of hidden units plus one multiplied by the number of output units.

Three other vectors store the state of the network. The input vector, *In*, supplies values to the input neurons. This vector has two elements, one for each input neuron, and is padded with an additional vector element. This additional element is used in the calculation of the neuron *bias* or *offset*. This offset may be viewed as a weight which is always connected to an input value of '1'. This last element in the input vector is always set to '1.0', and represents the offset input. This extra vector element permits the offset to be treated in the same manner as the other weighted interconnections.

In a similar fashion, the hidden layer state vector, *H*, holds the output of the neurons in the hidden layer. Like the input vector, this vector is also padded with an additional vector element containing a value of '1'. Finally, the output vector, *Out*, holds the value of the output neuron. This vector has a length equal to the number of output neurons.

Using this representation, a pairwise multiplication of the input vector *In* values with sections of the weight vector *W1* performs all of the multiplications necessary for the calculation of the first layer. A *stride()* function permits the values in the *W1* weight vector to be supplied in sections which are the same length as the *In* vector. This allows the proper pairing with the values in the weight vector *W1*. The products of these vectors are then summed and passed through the limiting function $f(x)$. The final value in this summed vector is the output of a hidden layer neuron. This process is repeated for each neuron in the hidden layer. These sums are then copied to the hidden layer state vector *H*.

After the outputs of the hidden neurons have been calculated, a similar process is used to calculate the values of the output neurons. The vector of values representing the outputs of the hidden layer, *H*, is multiplied by the segments of the vector *W2*. Again, the *stride()* function is used to provide these sections of the *W2* vector. These products are summed and passed through the function $f(x)$ to produce the values in the output vector.

The vector-based data-parallel code for this process is fairly simple. Figure 7.37 shows the code used to compute the output of the neurons in the hidden layer. This vector operation is repeated for each neuron in the layer. For clarity, declaration and initialization of the weight and input vectors are not shown in this code.

```
/* Calculate outputs of hidden neurons */
W = stride(W1, start, 1, i_len, w1_len);
T1 = f(add-scan(W * In));
```

Figure 7.37: Code to calculate the output of the hidden layer.

In the code in Figure 7.37, a vector *W* is defined using a *stride()* function. This takes the values in the weight vector *W1* which correspond to the inputs of a single neuron. This vector is multiplied by the input vector *In* and the values accumulated using an *add-scan()* function. The resulting vector is passed through the limiting function *f()* and is stored in the temporary vector *T1*. The last value in this temporary vector contains the output of the hidden layer neuron. The other vector elements in *T1* contain partial sums and are discarded. This process is repeated for each of the neurons in the hidden layer.

The code for calculating the values of the output neurons is nearly identical to the code used to calculate the outputs of the hidden neurons. If the second layer weight vector list *W2* is substituted for the first layer weight vector list *W1* and the hidden layer output vector H is substituted for the input vector *In*, the code is the same. Given the regular structure of the network model, this similarity is not surprising.

7.4.3 The Sigmoid Function

The function $f(x)$ is used to limit the values of the neuron outputs. In this network, we wish to limit the output to values between 0 and 1. McClelland and Rumelhart [88] use what they term the *logistic function* to perform this limiting. This function is given by Equation 7.4.

$$f(x) = \frac{1}{1 + e^{-x}} \tag{7.4}$$

This equation was selected because it provides the necessary limiting of the outputs while having some properties which are useful in the learning phase of the algorithm. Unfortunately, this equation contains the transcendental function e^x, which is somewhat difficult to calculate. Nordström and Svensson [97] list several functions which may be used as an approximation to the function used by McClelland and Rumelhart. These functions all have the same general characteristics. They are continuously increasing, approach 0 at $-\infty$ and 1 at $+\infty$, and have a continuous first derivative. The approximation we will use is given by the function in Equation 7.5.

$$f(x) = \frac{1}{2}\left(\frac{x}{1 + |x|} + 1\right) \tag{7.5}$$

This function is a simple polynomial which uses no transcendentals. The graph in Figure 7.38 shows both the logistic function of McClelland and Rumelhart, given by *f1(x)* and the approximation above, given by *f2(x)*. Note that the curves provide a similar limiting function. It is the general characteristics of the sigmoid, not the precise equation which is important in this case.

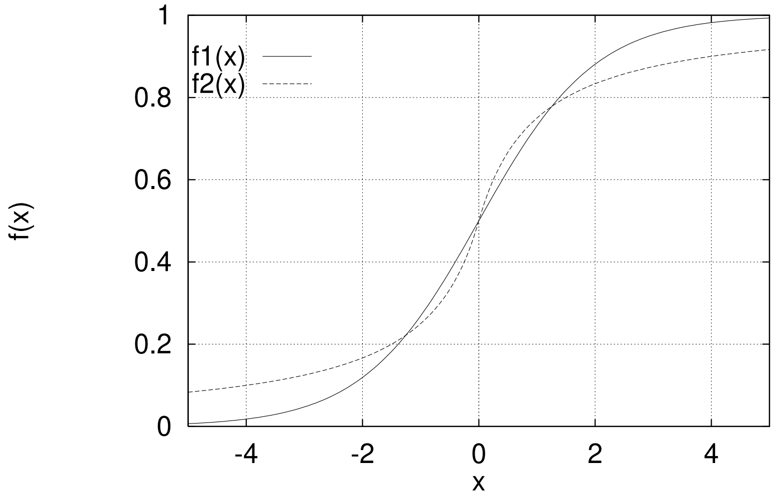


Figure 7.38: Sigmoid activation functions.

The C-like code for the function $f(x)$ is shown in Figure 7.39. It is a straightforward translation of the equation into software. Note, however, that the code is vector-oriented. For clarity the data type *Vector* is used. In this case, this data type is simply the renaming of the *C* language array of floating point values. The function $f(x)$ takes as its input parameter the data type *Vector*. The result is also a *Vector*. This specification indicates that this function performs a vector operation which can be mapped to the reconfigurable hardware.

```
/* Sigmoid activation function */
Vector
f(Vector  x) {
   return ((1.0 / 2.0) * (x / (1 + abs(x)) + 1));
   }  /* end f() */
```

Figure 7.39: Code for the sigmoid activation function.

7.4.4 Circuit Extraction

From these code fragments, circuits may be extracted which implement the neural network algorithm. As in previous examples, the circuits are extracted by creating the dataflow graph for the code. This graph is then used to configure the hardware. Since the final result of this code is a digital circuit, the sigmoid activation function circuit is treated as a macrocell, much like the predefined arithmetic and logical function macrocells used by the system.

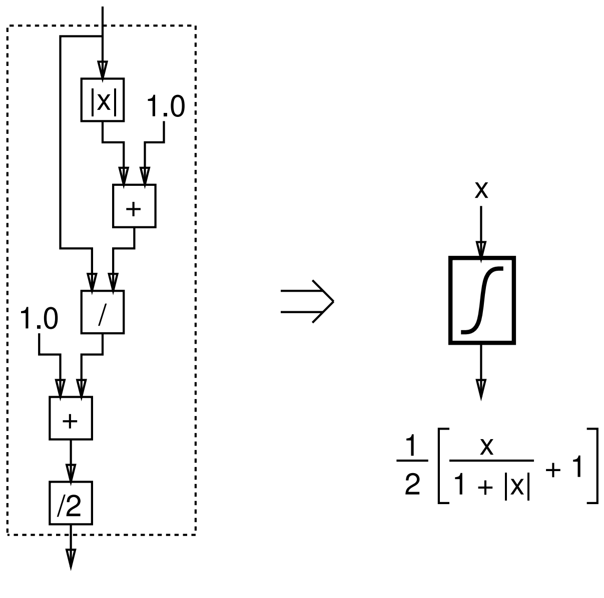


Figure 7.40: The sigmoid activation function circuit.

Figure 7.40 shows the dataflow circuit for the function $f(x)$. This circuit takes as its input a value x and returns the output $f(x)$. The functional units used by the circuit are: two adders, a divider, an absolute value and a divide-by-two circuit. Some simple optimizations have been performed on this circuit. A divide-by-two circuit, for instance, has been used rather than a full divider. Once the circuit for this function has been extracted, it may be used as a macrocell, much like the other macrocells in the circuit.

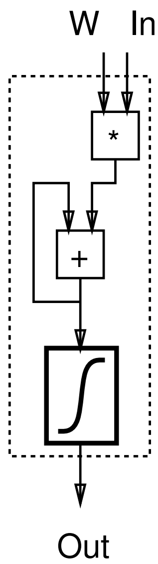


Figure 7.41: The neural network circuit.

Once the circuit for the sigmoid activation $f(x)$ has been extracted, the code for the neuron output calculation may be converted into a circuit. In this case, the circuit is very simple. As Figure 7.41 illustrates, the weight and the input vectors are multiplied, with the results being accumulated by an *add_scan()* macrocell. The result is then passed to the sigmoid activation function for output limiting.

This circuit processes one set of vectors for each neuron in the network. Because the *add_scan()* macrocell performs an accumulate function, it will be necessary to clear the output of this macrocell to zero before each vector operation begins. This can be accomplished either with a reset signal, or by writing directly to the accumulate register in the *add_scan()* macrocell. It is expected that a reset signal will be more efficient.

7.4.5 Performance

This implementation of this neural network algorithm is well suited to a reconfigurable logic based machine. The uniform nature of the model permits a single circuit to be configured and used to calculate the outputs of the network. Additionally, the bandwidth requirement of the circuit is small. Two vectors are input and a single output vector is produced. Only two input ports and one output port are required.

One unusual feature of this circuit is the calculation performed by the sigmoid function. While it is only necessary to take the sigmoid of the final sum of the weighted inputs, this circuit takes the sigmoid of each of the partial sums. This would be extremely wasteful on an instruction set architecture, but in this case, there is no penalty for performing these extra calculations. In fact, to do otherwise would require that more than one circuit be used in the calculation. This would require a potentially expensive reconfiguration phase in the algorithm. The ability to perform all of the calculations in a single pass with a single circuit is valuable, even though some computed values are never used.

The circuit produced by this code contains seven functional units. These units are cascaded in an essentially linear pipeline. In estimating the performance of this circuit, two assumptions are made. First, it is again assumed that the bandwidth of the memory system is sufficient. Two values must be supplied to the circuit input and one read from the output per clock cycle. Second, it is assumed that the time taken to fill the pipeline is negligible. For larger networks this is a valid assumption. With these assumptions, one weighted interconnection calculation is performed per clock cycle.

Since the metric typically used to measure performance of neural networks is *connections per second* or *CPS*, it is a simple matter to estimate the performance of this circuit. Since one connection is processed per cycle, the performance of the circuit in *CPS* is equal to the circuit clock speed.

As a comparison, the CRAY-2 can simulate this network at approximately 50 MCPS [97]. This would correspond to a clock speed of 50 MHz for the reconfigurable architecture. Similarly, a 10-processor Warp system has been benchmarked at 17 MCPS [104].

Nordström and Svensson [97] give benchmarks for other architectures, some of which calculate over 1000 MCPS. These machines, however are usually special purpose parallel architectures. One system, *GANGLION*, which makes use of reconfigurable logic to implement a fixed size network, can compute approximately 4480 MCPS.

While these special purpose architectures provide substantial increases over this reconfigurable logic approach as well as supercomputers, they are typically inflexible. But this does indicate that not all of the parallelism in the algorithm is currently exploited. Other approaches using reconfigurable hardware may provide substantially more performance, at a cost of some additional complexity.

7.5 The Fourier Transform

A popular method for producing spectral information about a signal is the Fourier transform. This transform converts a signal in the time domain to one in the frequency domain. Here, instead of representing a signal by an amplitude which varies over time, the signal is represented as a series of sinusoidal frequency components, each with a phase and magnitude. This representation is sometimes of value for its own sake, and at other times it provides an efficient means of processing and filtering the signal.

The Fourier transform and its implementations have a somewhat colorful history. The concept of representing signals in the frequency domain goes back at least as far as the Babylonians. A brief historical account, with references can be found in Oppenhheim, Willsky and Young [98].

While a potentially powerful technique for processing signals, calculation of the Fourier transform can be computationally intensive. Various techniques, including sophisticated custom hardware [120] have been employed to calculate the Fourier transform. In this section, the Fourier transform will be implemented for reconfigurable hardware. This implementation will be analyzed and compared to other implementations.

7.5.1 The Discrete Fourier Transform

While the Fourier transform was originally described in terms of continuous functions, it is also possible to perform a Fourier transform on digitized data. Here, a set of numeric values represent the amplitude of the signal at evenly spaced intervals. Given a set of N values, it is possible to compute the Fourier series which represents the digitized signal in the frequency domain. The result of this calculation is two sets of N values, representing the amplitude and phase of the components in the frequency domain. This transformation is commonly referred to as the *Discrete Fourier Transform* or *DFT* to distinguish it from its continuous counterpart [20, 106].

Equation 7.6 gives the equation for computing the discrete Fourier transform. The original signal is represented by the N sampled values in h_0 through h_{N-1}. Each of these sampled values is multiplied by a complex value and summed to produce a single component of the frequency domain representation, H_n.

$$H_n = \sum_{k=0}^{N-1} h_k e^{2\pi ikn/N} \tag{7.6}$$

From this equation we see that each of the N values of the original signal h contribute to the calculation of each value H in the frequency domain. The direct implementation of this equation is clearly of $O(N^2)$ complexity.

Equation 7.7 shows Equation 7.6 with the real and imaginary portions of H_n computed separately. The real portion $Re(H_n)$ gives the magnitude of the Fourier components, while the imaginary portion, $Im(H_n)$ gives the phase.

$$
\begin{aligned}
Re(H_n) &= \sum_{k=0}^{N-1} (Re(h_k)cos(2\pi kn/N) + Im(h_k)sin(2\pi kn/N)) \qquad (7.7) \\
Im(H_n) &= \sum_{k=0}^{N-1} (Im(h_k)cos(2\pi kn/N) - Re(h_k)sin(2\pi kn/N))
\end{aligned}
$$

From this representation it is a simple matter to produce a data parallel version of the algorithm. Figure 7.42 gives one approach to this calculation. This data parallel implementation of the DFT calculates a single component n of the DFT. This calculation must be repeated N times to produce all N components of the DFT.

The first portion of the algorithm generates a vector containing the integers from zero to $(N - 1)$. In a serial version of this algorithm, these values for K would typically be found in the control loop which keeps track of the number of terms computed. As with other data parallel calculations, this control structure information is not directly available to the calculation. The integer values for K are instead produced by the first three lines in the algorithm. Note that these three lines could have easily been condensed into a single line of source code of the form K = *add-scan(1) - 1*. It is shown in its expanded form with comments for clarity.

Once the K values are produced, the constant *w0* is calculated. K and *w0* are then multiplied and the *sin()* and *cos()* taken. This code assumes that library implementations of the *sin()* and *cos()* functions already exist. These implementations may consist either of custom designed circuitry or arithmetic sequences in data parallel code.

Once the transcendental functions are computed, the real and imaginary

```
K = 1; /* K = 1, 1, 1, ..., 1 */
K = add-scan(K); /* K = 1, 2, 3, ..., N */
K = K - 1; /* K = 0, 1, 2, ..., (N-1) */

w0 = (2 * PI * n) / N;
Sin = sin(K * w0);
Cos = cos(K * w0);

Dft_re = ((In_re * Cos) + (In_im * Sin));
Dft_im = ((In_im * Cos) - (In_re * Sin));

Dft_re = add-scan(Dft_re) / N;
Dft_im = add-scan(Dft_im) / N;
```

Figure 7.42: The data parallel code for the DFT.

portions of the DFT are calculated. These are summed using the *add-scan()* operation and scaled by a factor of N. The vector calculation of length N produces a single value in the DFT of the signal. This procedure must be repeated N times to produce the complete transform.

From the data parallel code in Figure 7.42, a circuit can be extracted. Figure 7.43 shows the extracted circuit. Note that two vectors are input producing two output vectors. The output vectors, however, contains the summed values. Only the final sum is likely to be of interest.

This implementation was tested against the data in Figure 7.44. Thirty-two samples of the function $\cos(2\pi n/8)$ taken at intervals of one are used as the input. The impulses in the figure represent the sampled values while the sinusoidal envelope gives the sampled function.

Theory predicts that the Fourier transform of a cosine function will produce two positive symmetrical impulses. The results of the DFT calculation

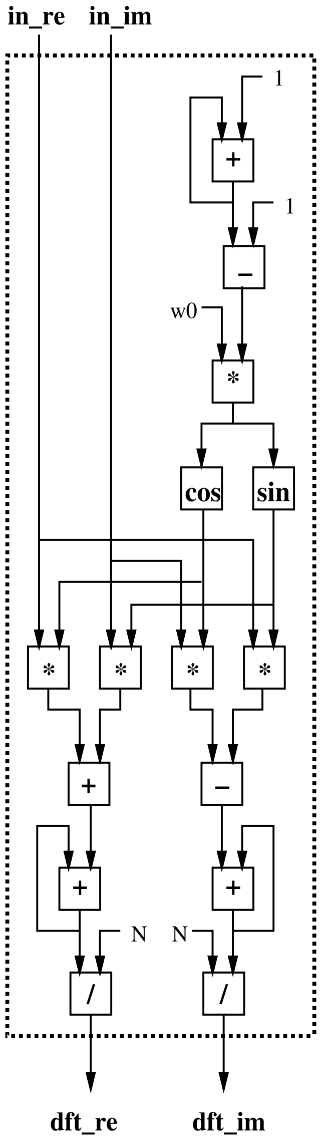


Figure 7.43: The circuit for the DFT.

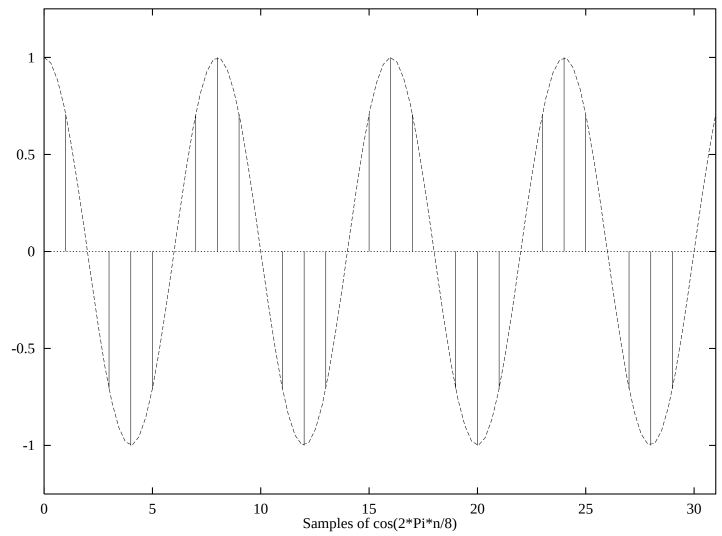


Figure 7.44: The 32 samples of the function $\cos(2\pi n/8)$.

in Figure 7.45 verify this expectation.

7.5.2 The Fast Fourier Transform

While the representation of a digitized signal provided by the Fourier transform has many powerful uses, generating this frequency domain representation from the sampled time domain data is computationally intensive. The calculation involves all N samples to compute each of the N transform values. This gives the calculation an overall complexity of $O(n^2)$. Additionally, the computation involves repeated computation of sine and cosine values, which are often themselves computationally intensive.

Fortunately, a more efficient method for computing the DFT exists. This method is commonly known as the *Fast Fourier Transform* or *FFT*. The modern version of this algorithm was published by Cooley and Tukey in 1965, although similar methods were reported earlier in the century. The general

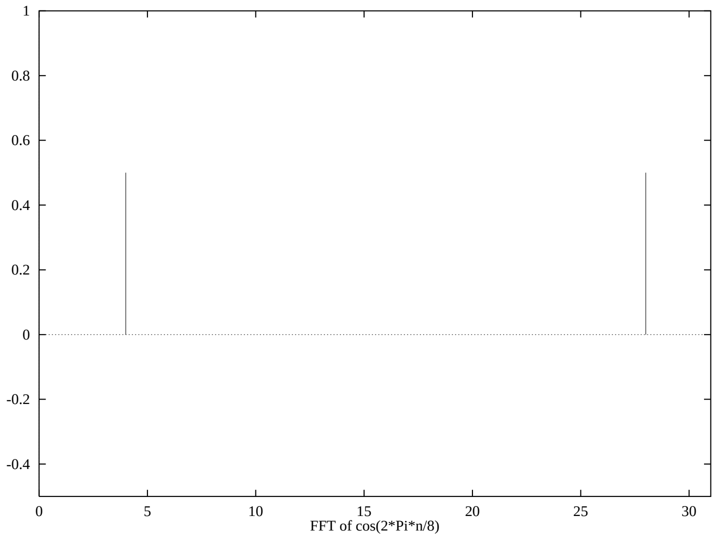


Figure 7.45: The DFT of the function $\cos(2\pi n/8)$.

technique goes back at least as far as Gauss in the early 19th century.

This technique is based on restructuring the computation to take advantage of previously computed values. One way to view the restructured calculation is that of breaking the computation of a single DFT of length N into two DFT calculations of length $N/2$. Equation 7.8 gives the method used to decompose the DFT into these two calculations.

$$\begin{aligned} H_n &= \sum_{k=0}^{N-1} h_k e^{2\pi ikn/N} \\ &= \sum_{k=0}^{(N/2)-1} h_{2k} e^{2\pi ikn/(N/2)} + W^n \sum_{k=0}^{(N/2)-1} h_{2k+1} e^{2\pi ikn/(N/2)} \\ &= H_n^e + W^n H_n^o \end{aligned} \tag{7.8}$$

This equation reveals that the two components used to produce the DFT are composed of the even and odd values of the original data, respectively.

These two smaller DFT calculations are summed, with the odd portion of the calculation multiplied by the scale factor W^n. This reduces the complexity of the calculation from $O(n^2)$ to $2 * O((n/2)^2)$.

This process can be carried further, with each of the two smaller calculations split into even and odd portions, reducing the complexity to $4 * O((n/4)^2)$. This process may be continued until the data can no longer be divided in two. The final DFT of a single value h_n is simply the value of h_n. For values of N which are an even power of two, this produces an overall complexity of $O(n \log_2(n))$.

While this is not a significant improvement for small values of n, for larger values, the improvement can be substantial. For $N = 16$, the FFT is roughly four times faster than the standard DFT. For $N = 1024$, the difference is a factor of 100. Larger N produce even larger gains in efficiency.

In Equation 7.8, the value W^n is specified as a scale factor for the odd portion of the FFT. These values, sometimes referred to as *twiddle factors* take the complex value given in Equation 7.9. Note that the computation of W^n contains all of the transcendentals used in the calculation.

$$W^n = e^{2\pi i/N} \tag{7.9}$$

This method of computing the FFT is often given the graphical representation shown in Figure 7.46. Here, a small FFT of eight values is computed. Note that even and odd values are paired. Also note that there are three distinct stages of the calculation, providing the logarithmic component of the complexity.

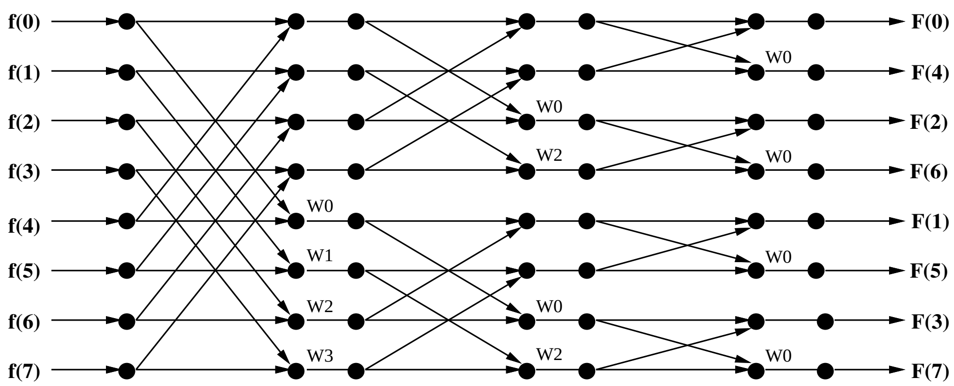


Figure 7.46: A standard representation of the FFT.

From this representation, a basic *cell* can be viewed as the basic component of the FFT calculation. This cell is shown graphically in Figure 7.47. This cell takes two complex values, a and b as its input. These input values are used to produce the two complex output values c and d. The value of c is simply the sum of a and b. The value of d is the difference of a and b, multiplied by the appropriate value of W^n.

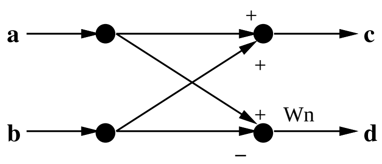


Figure 7.47: The basic FFT cell.

Curiously, while most descriptions of the FFT rely on a basic FFT cell such as that in Figure 7.47, the typical figure such as the one in Figure 7.46 does not clearly distinguish these cells, except in the last stage of the calculation. All other such cells are stretched and interleaved in some fashion.

As an alternative to this representation, Figure 7.48 rearranges the elements of the diagram to provide distinct cells. This rearrangement causes the previously interleaved data paths to be independent units, but causes the values passed between stages to be interleaved.

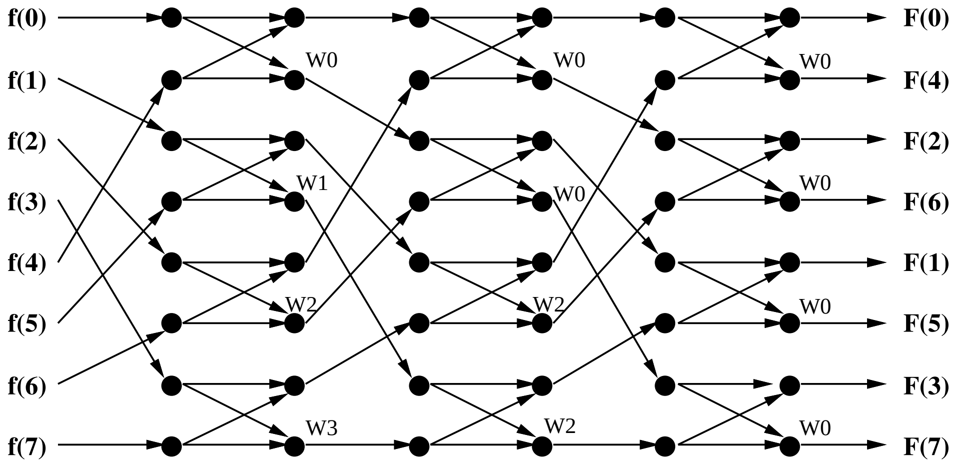


Figure 7.48: An alternate representation of the FFT.

Fortunately, the interleaving between stages is the same for each stage in the calculation. The form of this interleaving is that of a *perfect shuffle*. The perfect shuffle is so named because it produces data interleaved in a manner similar to that of a deck of playing cards being shuffled. The perfect shuffle is known to have some interesting computational properties and has been studied in other contexts [117].

From this representation, the implementation of the basic cell is fairly simple, but the data patterns necessary to produce the desired results may require further exploration. Figure 7.49 shows the way in which vectors of data are accessed to feed the inputs and produce the outputs from the basic FFT cell. Assuming the input values are available in a single vector, producing the vector inputs A and B is fairly simple. The input vector is split in half, with the first $N/2$ elements comprising the A vector and the second $N/2$ elements comprising the B vector.

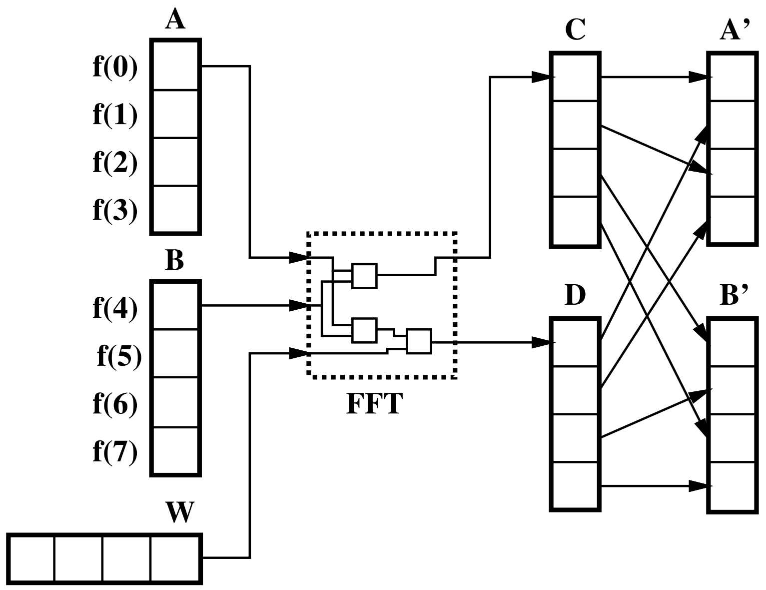


Figure 7.49: Vectorizing the FFT.

These inputs, along with the appropriate values for W^n produce two output vectors C and D of length $N/2$. Before these vectors can be used by the subsequent stages of the calculation they must be reordered. If C and D are stored in contiguous memory locations, they may be viewed again as a single vector of length N. A perfect shuffle of this vector produces a new vector of length N which may be split into two new vectors A' and B' and fed into the next stage of the calculation.

Fortunately, the perfect shuffle can be easily implemented using the *stride()* function. Here a mixed valued stride of $((N/2) + (1/2))$ starting at the first element in the vector produces the desired result. It should be emphasized that for a system which has direct hardware support for mixed valued striding, this operation takes no resources. The *stride()* function simply describes the access pattern used. Data are not necessarily physically manipulated by this operation.

With the basic cell and the data access pattern specified, the algorithm can now be implemented using data parallel code. The implementation of the basic cell for complex numbers can be taken directly from the diagram for the cell in Figure 7.47.

Figure 7.50 shows the data parallel code for the basic cell. Note that since this calculation involves complex values, both real and imaginary values are computed. For the calculation of the vector C, this is simply two additions. For the subtraction then multiplication by W^n in the calculation of D, the multiplication of two complex values produces a more complicated expression.

```
C_re = A_re + B_re;
C_im = A_im + B_im;

D_re = (W_re * (A_re - B_re)) - (W_im * (A_im - B_im));
D_im = (W_re * (A_im - B_im)) + (W_im * (A_re - B_re));
```

Figure 7.50: The data parallel code for the FFT.

The code in Figure 7.50 is executed on the vectors of length N for the required $\log_2(N)$ iterations to produce the final DFT values. From this data parallel code, the circuit in Figure 7.51 is extracted.

While this is the bulk of the algorithm, the generation of the values for the W vector have not yet been addressed. The vector W is also a complex quantity with real and imaginary components. Equation 7.10 shows the expanded version with transcendental functions sine and cosine.

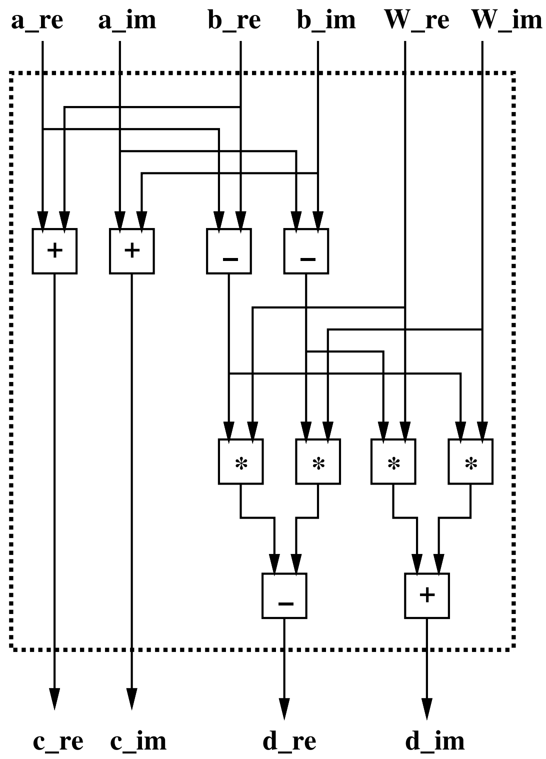


Figure 7.51: The extracted FFT circuit.

$$W^n = \cos(2\pi n/N) - i\sin(2\pi n/N) \tag{7.10}$$

The vector W contains $N/2$ values which are reused throughout the calculation. This vector should be precomputed at the beginning of the calculation. While the reconfigurable hardware may easily be employed in this computation, this may or may not be the best approach. Because of the small size of this vector it is possible that the host may be profitably employed in this portion of the computation.

Also note that the vector W changes with each stage of the computation. Unfortunately, no simple access method for producing this new W vector is possible. While the new vector of length $N/2$ can be generated using two *stride()* functions, this will require some physical rearranging of data. Again, since this operation is fairly simple, it may be desirable to employ the host in this manipulation.

Finally, as with the standard FFT calculation, the data is produced in a peculiar order. This order is often referred to as *bit reversed* order. Here, the actual index of the result is determined by reversing the bits of the vector index of the values produced. For example, in Figure 7.48 the second transformed value in location "1" is denoted *F(4)*. The index "4", or binary (100) is determined by reversing the bits in the binary value of its actual location, 1, or binary (001).

Unfortunately, these is no obvious data access pattern that would permit striding to be used to reorder the data. Again, it is likely that this function would be best performed by the host processor. Note that the solution involves the swapping of pairs of elements in the vector and is a fairly simple process.

The complete algorithm for the FFT using this approach has been implemented and simulated. Using the data from the DFT example in Figure 7.44, a result identical to Figure 7.45 is produced. Because the values are identical, they are not reproduced here.

7.5.3 The FFT in 2D

While the Discrete Fourier Transform is defined to act on a vector of digitized waveform data, it is not strictly limited to one dimensional signal data. The DFT may also be applied to image processing tasks. The use of a Fourier representation for digitized images presents many interesting possibilities for image enhancement. The image processing functions which use a spectral representation are often difficult to perform using a standard image representation.

To produce the two dimensional DFT of a digitized image, the DFT of each row of pixels in the image must be taken. The result of these DFT operations is then processed by taking the DFT of these values by columns, rather than by rows. (For a more detailed analysis, as well as some interesting processing techniques based on digital images represented in the frequency domain, the reader is referred to Gonzalez and Wintz [43].)

For an NxN image, $2N$ transforms are required. If the FFT is employed the complexity of the complete two dimensional image FFT calculation is $O(2N^2 \log_2(N))$. For even moderately sized images, this represents a significant amount of calculation.

The data parallel implementation of the FFT is used to compute the two dimensional FFT of a simple test image. Figure 7.52 shows a $256x256$ pixel image where each pixel was represented by eight bits, giving 256 distinct shades of grey. The processed image shows the spectral pattern produced by this synthetic image.

It should be noted that this spectral image is displayed in a somewhat unconventional manner. First, the image was translated so that the "interesting" portion of the image is moved to the center. This translation can be viewed as dividing the image into four quadrants and swapping diagonal quadrants. This places the pixels which were originally in the corners at the center of the image.

Second, the values of the pixels are scaled in a logarithmic manner. This is because the values produces by the FFT tend to be logarithmically distributed. If it were not for this scaling, the image would appear more as a small white dot in the center of the frame. Finally, the image represents only the real portion of the processed image. The phase information in the imaginary portion of the processed image, while important, does not contain information which is as meaningful when viewed as an image.

Figure 7.53 contains a more realistic image and its Fourier transform. Again the transform image is a scaled version of the real portion of the transformed image.

This two dimensional image DFT has some computational features which should be mentioned. First, since all of the transforms are of a fixed length N, the W vector need only be calculated once. The versions of the W vector used

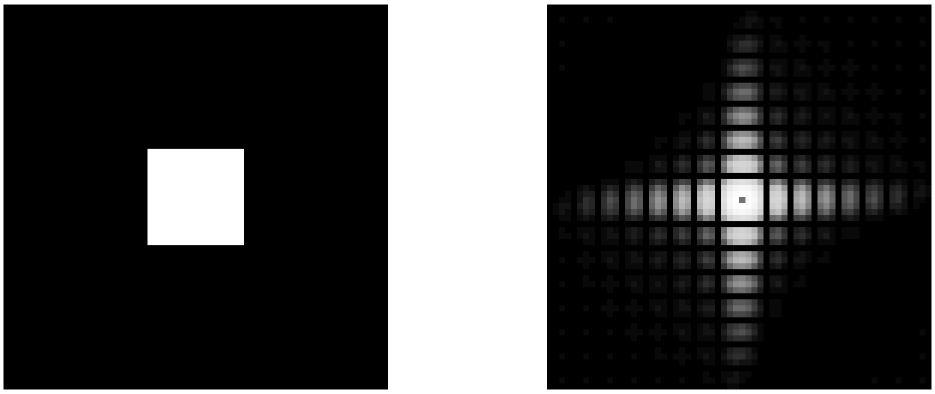


Figure 7.52: A square image and its 2D FFT.

Figure 7.53: A more complex image and its 2D FFT.

in each of the $\log_2(N)$ stages of the calculation may also be precomputed and stored separately for repeated use.

Also, the rearranging of the output data via the bit reversal method can be postponed until after the complete image is processed. Since each row in the image will be ordered in the same manner, the calculation in the second dimension, while performing the computations in a bit reversed order, still performs all of the necessary calculations.

Finally, some mention should be made of the method for transforming the frequency domain version of a signal or image back into the standard digitally sampled representation. While this is not necessary for many applications, others which use the DFT as an intermediate step in processing a signal will require some means of returning the signal to its original representation.

The inverse transform is very similar to the forward transform. In fact, the inverse transform is nothing but the forward transform, with two additional processing steps. These two additional processing steps are the taking of the complex conjugate of each point and dividing the result by N. This procedure returns data in the frequency domain to data in the sampled domain. This inverse transform may be easily implemented by modifying the FFT code.

7.5.4 Performance

From Figure 7.49, we see that each stage in the FFT processes vectors of length $N/2$. These vectors are processed for $\log_2(N)$ stages. This gives a total of $(N/2)\log_2(N)$ data elements processed.

Assuming that throughput is maintained such that one data element per cycle is processed, the time to calculate a FFT of length N is given by Equation 7.11, where f is the clock frequency of the system in Hertz (Hz).

$$T_{fft} = (N/2f) * \log_2(N) \tag{7.11}$$

From this analysis, the reconfigurable hardware should be able to process a 1024 point FFT in approximately 0.1 milliseconds, assuming a 50 MHz clock. Sohie and Chen [116] give performance benchmarks for modern digital signal processors. These processors typically perform an identical calculation in one to two milliseconds. This predicted order of magnitude increase in performance is especially significant when one considers that the processors executing the FFT benchmarks are not standard microprocessors, but digital signal processors which are optimized for operations such as the FFT.

This order of magnitude increase in performance should allow a two dimensional Fourier transform to be calculated at rates approaching that of standard video. Assuming a video rate of 30 frames per second, the reconfigurable system is able to produce the two dimensional Fourier transform of a $256x256$ pixel video signal in real time while executing at a system clock speed of just over 15 MHz. A $512x512$ signal can be processed in real time if a system clock speed of just of 70 MHz can be achieved. The ability to process video data in this manner should permit filtering and enhancement of video images not normally available using standard techniques.

7.6 The Livermore FORTRAN Kernels

So far, all of the algorithms examined have a common theme. All exhibit relatively large amounts of data parallelism and are easily vectorized. All of the algorithms examined so far, in fact, have been implemented at some time using custom hardware. For reconfigurable architectures to be truly useful, they must be able to perform general purpose computations such as those commonly found on instruction set architectures. To better examine the feasibility of general purpose computing, codes from existing applications, particularly those from high performance machines, should be examined.

One such collection of codes is the *Livermore FORTRAN Kernels* or *LFK* [90] [33]. The LFK suite is selected for implementation on reconfigurable hardware for several reasons. First, the LFK is a widely used tool to measure CPU performance. This permits comparison of results to a wide range of existing architectures.

Second, the LFK are composed of a number of tests which include a wide range of computational structures. While some of these structures are used to measure the peak performance of a system, others are constructed specifically to limit performance. This permits an examination of architectural weaknesses as well as strengths.

Finally, the LFK are relatively compact and self contained. This allows their simulation on models of proposed hardware. Performance information gained from such simulations is valuable in guiding the design.

7.6.1 The Kernel Code

The LFK are a collection of 24 relatively small fragments of code. Each of these code fragments contains a CPU intensive loop, giving the test suite its informal name, "the Livermore Loops". The LFK were originally developed in 1970 to test the code generated by compilers. Over time, these codes have become a benchmarking tool for new supercomputer systems.

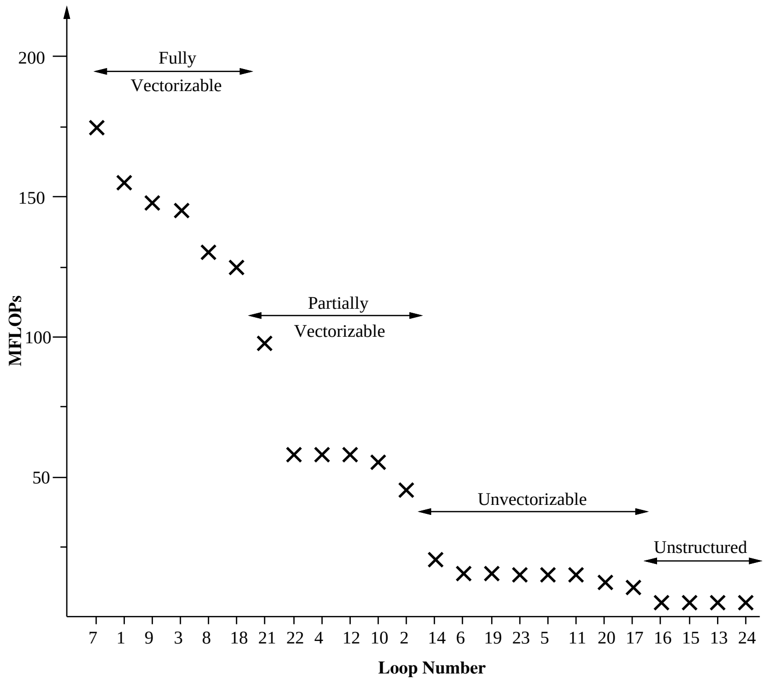


Figure 7.54: Performance sorted by MFLOPs for a CRAY X-MP.

As the LFK have evolved into a benchmarking tool, new loops have been added to exercise specific features of both compilers and hardware. The number of loops has grown from the initial 12 to the current 24.

The kernels in this study were converted by hand from the original FORTRAN to a data parallel version of the *C* language. Most of the transformations performed in translating the source code are fairly simple, and could conceivably be performed by a sufficiently intelligent compiler.

To verify the accuracy of the translated kernels, the data parallel C code was simulated. The results from this simulation were verified against results from a standard C version of the LFK [35].

It should be noted that the LFK are specified for high accuracy floating point arithmetic. While some work is being done on the implementation of floating point arithmetic in reconfigurable logic [32], it is understood that using the technology available today, a very large RPU would be required to implement these functions. While numeric accuracy is important, it is the computational structures in the LFK which are of primary interest. It is these structures, not numeric accuracy, which have the greatest impact on performance.

Figure 7.54 plots the performance of the 24 Livermore Fortran Kernels run on a *CRAY X-MP* using the *CFT77 3.0* compiler [103]. The numbers are listed in MFLOPs and are sorted by performance. From this sorted graph of the LFK, four performance ranges can be identified. These are:

- Fully vectorizable
- Partially vectorizable
- Unvectorizable
- Unstructured

In general, kernels in each of these regions present different computational challenges. These will be discussed in more detail as the kernels are implemented. For brevity, only representative kernels from each region are discussed. Kernels were chosen primarily for their simplicity in illustrating the particular computational structures.

7.6.2 Fully Vectorizable Loops

Kernels in the fully vectorizable category typically perform the highest on vector supercomputers. These kernels are characterized by being easily vectorized as well as providing enough work to occupy multiple functional units.

Loop 1: Hydrodynamic Code

Loop 1 is a fragment from a hydrodynamic simulation. The original FORTRAN code for this loop is shown in Fig. 7.55. This loop is easily vectorizable and can make concurrent use of several functional units.

```
      Do  1 k = 1,n
1         X(k) = Q + (Y(k) * ((R * Z(k+10)) + (T * Z(k+11))))
```

Figure 7.55: The original FORTRAN code for Loop 1.

The translation of this algorithm to data parallel form is shown in Figure 7.56. Because of the structure and simplicity of this loop, the similarities between the FORTRAN code, the algorithm and the data parallel code are clear.

```
Z10 = delta(Z, 10);
Z11 = delta(Z10, 1);
X = q + (Y * ((r * Z10) + (t * Z11)));
```

Figure 7.56: The data parallel code for Loop 1.

Figure 7.57 shows the RPU circuit extracted from the dataflow graph of this code. This circuit makes use of 5 functional units, and has a latency of 5 functional units.

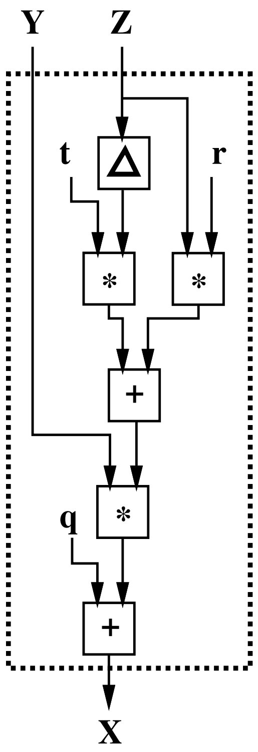


Figure 7.57: The configured circuit for Loop 1.

Two aspects of this circuit may require further explanation. First, only a single *delta* functional unit is employed, in spite of the use of two such operators in the code. Because the initial vector has a fixed index offest, the memory system is assumed to begin its access of the vector Z at this offset. This reduces the size of the circuit as well as eliminating data which are never used in the calculation.

The second interesting feature of this circuit is the use of the constants q, r and t. One alternative would be to use RPU bandwidth and make them input vectors. Instead, they are embedded in the circuit as constant inputs to the functional units. These constant inputs may also allow further opportunities for circuit optimizations.

Estimating performance of this circuit is fairly simple. Assuming sufficient memory bandwidth and a clock speed of 50 MHz, the processor will produce one result per clock cycle, neglecting latency. Since all functional units are kept busy on each cycle, approximately 250 million operations per second are performed. If the functional units all perform floating point operations, this corresponds to 250 MFLOPs. Even at this modest clock speed, this exceeds the rate of computation of the CRAY X-MP.

Loop 3 - Inner Product

The second fully vectorizable loop is an inner product calculation. This is a multiply-accumulate function found in many applications, including matrix arithmetic. Because of the widespread use of this type of calculation, most supercomputers are especially efficient at its execution. Figure 7.58 gives the original FORTRAN code for this kernel.

```
      Do  3 k = 1,n
3         Q = Q + (Z(k) * X(k))
```

Figure 7.58: The original FORTRAN code for Loop 3.

The data parallel version of the code is shown in Figure 7.59. The accumulate operation used to produce Q is implemented as a parallel prefix *scan* operator. This produces a vector Q containing the accumulated values, with the last element containing the final sum.

```
Q = add-scan(Z * X);
```

Figure 7.59: The data parallel code for Loop 3.

The circuit extracted from this data parallel code is shown in Figure 7.59. This circuit is fairly simple, containing only two functional units. The memory bandwidth required is also fairly modest. Two vector inputs and a vector output are required.

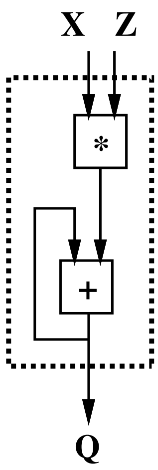


Figure 7.60: The configured circuit for Loop 3.

At a rate of 50 MHz, neglecting overhead, this circuit performs 100 million operation per second. This is somewhat less than the CRAY X-MP. The very small number of functional units indicates very little exploitable parallelism. Hence the modest performance.

This test, however, is intended to gauge the efficiency of an accumulation operation. A vector architecture with no direct support for this operation would have to resort to slower and more complex schemes to perform the accumulation. The availability of custom operations like *add-scan()* and their efficient implementation in reconfigurable logic provide respectable levels of performance for this common operation.

7.6.3 Partially Vectorizable Loops

The next group of kernels perform at a level somewhat below that of the fully vectorizable kernels. These loops are referred to as *partially vectorizable loops.* In these kernels, the ability to use the vector units fully is reduced. While these loops do not have the performance levels of the fully vectorizable loops, their levels of performance are still substantial, but only a fraction of the peak performance achieved by the fully vectorizable loops.

Loop 12 - First Difference

Loop 12 is a first difference calculation. The original FORTRAN for this calculation is shown in Figure 7.61. Despite its relatively low performance, this loop is fairly simple and is easily translated to data parallel code.

```
        Do  12 k = 1,n
12          X(k) = Y(k+1) - Y(k)
```

Figure 7.61: The original FORTRAN code for Loop 12.

Figure 7.62 gives the translated data parallel code for the first difference calculation. The use of the *delta()* function provides the offset version of the vector Y, saving input bandwidth.

```
Y1 = delta(Y, 1);
X = Y - Y1;
```

Figure 7.62: The data parallel code for Loop 12.

The circuit extracted from this code is shown in Figure 7.63. This is perhaps the simplest circuit produced by the LFK. While the algorithm contains a large amount of data parallelism permitting vectorizations, there is only a single arithmetic functional unit used by the calculation. At a clock rate of 50 MHz, neglecting overheads, this calculation proceeds at a rate of 50 million operation per second. This is similar to the rate achieved by the CRAY X-MP.

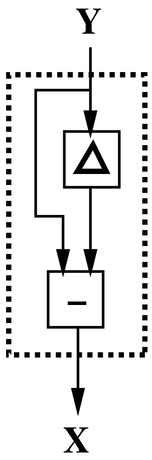


Figure 7.63: The configured circuit for Loop 12.

The performance of this loop is reduced because of a lack of work for the functional units. Assuming available memory bandwidth, it is possible to further accelerate this algorithm. N configurations identical to that shown in Figure 7.63 could be replicated within the RPU. The input vector Y could be split into N parts, each performing a first difference independently. This approach can increase performance by a factor of N, at the cost of increased bandwidth.

Loop 22 - Planckian Distribution

Loop 22 is from a Planckian distribution program. Here, the computation rate is slowed by conditional execution. Figure 7.64 gives the original FORTRAN implementation. Some vector processors provide special hardware for this situation. This hardware produces a bit vector called a *vector mask* from the conditional statement. This bit vector is used to selectively perform operations later in the computation. A similar technique to vector masking is used by the reconfigurable hardware implementation.

```
      Do  22 k = 1,n
          Y(k) = 20.0d0
          if (U(k) .lt.  20.0d0 * V(k)) Y(k) = U(k) / V(k)
          W(k) = X(k) / (dexp(Y(k)) - 1.0d0)
   22 Continue
```

Figure 7.64: The original FORTRAN code for Loop 22.

Despite the conditional statement, the data parallel code is again a fairly simple translation from the original FORTRAN. Figure 7.65 gives the data parallel code for this loop.

```
if (U < (V * 20.0))
   Y = (U / V);
   else
      Y = 20.0;

W = X / (exp(Y) - 1.0);
```

Figure 7.65: The data parallel code for Loop 22.

In this implementation, the conditional statement provides two alternate values for the elements in Y, depending on the result of the conditional statement. This permits a parallel computation of the two values, with the proper result being selected.

Figure 7.66 gives the circuit extracted from the data parallel code. The comparison operator (<) takes two arithmetic inputs and produces a single bit output. This output is used as a select line to a multiplexer. This multiplexer selects the appropriate value of Y, which will be used later in the calculation. This approach can be viewed as building a vector mask "on the fly".

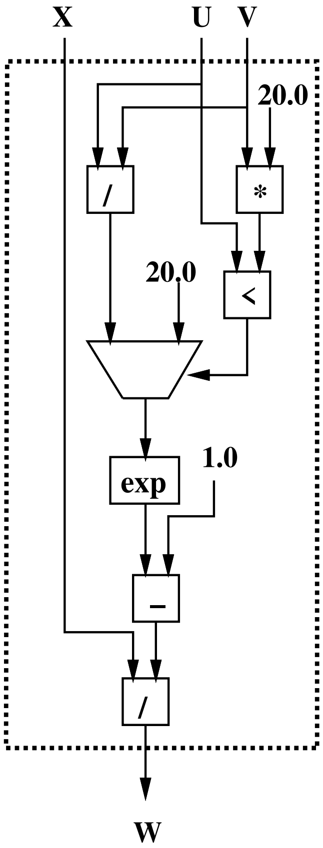


Figure 7.66: The configured circuit for Loop 22.

While this is a more complex loop, only 5 functional units, not including the comparison or the the multiplexer, are used. This assumes that the *exp()*

function is counted as a single functional unit. Depending on the implementation, this operator may be composed of other simpler arithmetic and logical operations.

Assuming a clock speed of 50 MHz, this implementation achieves approximately 250 million operations per second. This is almost four times the rate of the CRAY X-MP reference machine. This increase is attributed to the ability to efficiently perform conditional operations.

7.6.4 Unvectorizable Loops

The kernels in this performance range are typically unvectorizable and are unable to make extensive use of vector hardware. Since they are not able to make use of the vector processing facilities that helped enhance performance in the previous loops, their performance is not only considerably lower, but also more uniform. These algorithms are typically forced to use the non-vector portion of the CPU, thus testing the performance of this portion of the architecture.

Most of these loops are unvectorizable because of data dependencies introduced by recurrence equations. While difficult or impossible to vectorize using traditional fixed instruction architectures, the use of structures such as parallel prefix *scan* operators open up new possibilities for these functions.

Loop 5 - Tridiagonal Elimination

Loop 5 is a fragment of code used in tridiagonal elimination. The original FORTRAN code as shown in Figure 7.67 contains a data dependency in X that prohibits vectorization. This kernel typifies a class of equations known as first order linear recurrence equations. Several approaches to parallelizing this class of equations have been proposed [69] [67] [37].

```
       Do  5 k = 1,n
5          X(i) = Z(i) * (Y(i) - X(i-1))
```

Figure 7.67: The original FORTRAN code for Loop 5.

Figure 7.68 gives the dataflow graph for this equation. While simple recurrences can be implemented with a single parallel prefix operation, this recurrence is more complicated. The feedback path through two functional units eliminates the possibility of a simple pipelined approach.

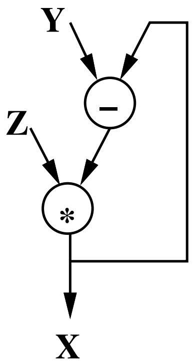


Figure 7.68: The dataflow graph for Loop 5.

The approach demonstrated here makes use of the fact that the computed values are actually independent if previously computed values are substituted into the subsequent equations. The first four values of X produced by this substitution are shown in Equation 7.12.

$$\begin{aligned} X_1 &= Z_1 * (Y_1 - X_0) \qquad (7.12) \\ &= Z_1 Y_1 - Z_1 X_0 \end{aligned}$$

$$\begin{aligned}
X_2 &= Z_2 * (Y_2 - X_1) \\
&= Z_2Y_2 - Z_2X_1 \\
&= Z_2Y_2 - Z_2Z_1Y_1 + Z_2Z_1X_0 \\
X_3 &= Z_3 * (Y_3 - X_2) \\
&= Z_3Y_3 - Z_3Z_2Y_2 + Z_3Z_2Z_1Y_1 - Z_3Z_2Z_1X_0 \\
X_4 &= Z_4Y_4 - Z_4Z_3Y_3 + Z_4Z_3Z_2Y_2 - Z_4Z_3Z_2Z_1Y_1 + Z_4Z_3Z_2Z_1X_0
\end{aligned}$$

From this form, it is clear that all values of X_i may actually be calculated in parallel, given the initial condition X_0. Unfortunately, a naive implementation of this fully parallel approach results in an $O(N^3)$ complexity for the algorithm.

While the direct implementation of the expanded calculation results in an undesirably high complexity, it is clear that there is still much parallelism in the representation. What is desirable is a compact mathematical representation for this expansion which can be easily manipulated. Given the regular structure of the elments in the calculation, a standard mathematical series representation of each value X_n should be useful. Equation 7.13 gives the series representation for X_n.

$$X_n = \sum_{i=1}^{n} \left(\left(- \prod_{j=i}^{n} -Z_j \right) Y_i \right) + X_0 \prod_{i=1}^{n} -Z_j \tag{7.13}$$

This concise representation exposes the parallelism in the equation, while maintaining a form which permits easy manipulation.

Table 7.2 shows the mathematical notation for series calculation and the corresponding data parallel programming language construct. Other series calculation expressions and corresponding *scan* operators also exist. Binary series expressions and their corresponding boolean *scan* operations are one class of examples. For the purposes of this derivation, however, only the sum and product series are necessary.

Expression	Language Construct
$Y_n = \sum_{i=1}^{n} X_i$	Y = add-scan(X)
$Y_n = \prod_{i=1}^{n} X_i$	Y = mult-scan(X)

Table 7.2: Mathematical expressions and their language constructs.

Using the correspondence between the standard mathematical operations for series calculation and their data parallel constructs, mathematical representations of algorithms involving series calculations can be directly transformed into data parallel algorithms containing only vector and parallel prefix operations.

While in a form of sums and products, the parallel prefix operators strictly correspond to series with fixed indices. Before making use of the representation in Equation 7.13, some transformations must be performed. First, the product portion of the equation must be modified to change the starting index to a constant value. This is most easily accomplished by multiplying by unity in the form of a ratio of the missing product terms, as shown in Equation 7.14.

$$X_n = \sum_{i=1}^{n} \left(\left(\frac{\prod_{k=1}^{i-1} -Z_k}{\prod_{k=1}^{i-1} -Z_k} \cdot - \prod_{j=i}^{n} -Z_j \right) Y_i \right) + X_0 \prod_{i=1}^{n} -Z_j \tag{7.14}$$

By merging the two products in the numerator into a single product and moving this common expression outside of the summation, the reduced series in Equation 7.15 is derived.

$$X_n = \prod_{j=1}^{n} -Z_j \left(\sum_{i=1}^{n} \frac{-Y_i}{\prod_{k=1}^{i-1} -Z_k} + X_0 \right) \tag{7.15}$$

Finally, the index of the product in the denominator must be brought up from $i-1$ to i. This is most easily accomplished by multiplying by unity in the form of $-Z_i/-Z_i$. This results in the final form of the expression as shown in Equation 7.16.

$$X_i = \prod_{i=1}^{n} -Z_i \left(\sum_{j=1}^{n} \frac{Z_j Y_j}{\prod_{k=1}^{j} -Z_k} + X_0 \right) \tag{7.16}$$

To verify that this form of the equation is the same as the original expansion, the value of X_4 is calculated using this equation and returned to its original form. Equation 7.17 demonstrates that the transformed series produces the same values as the original recurrence equation.

$$X_4 \;=\; Z_4 Z_3 Z_2 Z_1 \left[\frac{Z_4 Y_4}{Z_4 Z_3 Z_2 Z_1} + \frac{Z_3 Y_3}{-Z_3 Z_2 Z_1} + \frac{Z_2 Y_2}{Z_2 Z_1} + \frac{Z_1 Y_1}{-Z_1} + X_0 \right] \tag{7.17}$$

$$\begin{aligned} &= \frac{Z_4Z_3Z_2Z_1 \cdot Z_4Y_4}{Z_4Z_3Z_2Z_1} + \frac{Z_4Z_3Z_2Z_1 \cdot Z_3Y_3}{-Z_3Z_2Z_1} + \frac{Z_4Z_3Z_2Z_1 \cdot Z_2Y_2}{Z_2Z_1} + \\ &\quad \frac{Z_4Z_3Z_2Z_1 \cdot Z_1Y_1}{-Z_1} + Z_4Z_3Z_2Z_1X_0 \\ &= Z_4Y_4 - Z_4Z_3Y_3 + Z_4Z_3Z_2Y_2 - Z_4Z_3Z_2Z_1Y_1 + Z_4Z_3Z_2Z_1X_0 \end{aligned}$$

The expression in Equation 7.16 may now be converted to data parallel code by simply substituting the appropriate scan operations for the sums and products. Vector operations are used for the multiplication, division and addition operations. Figure 7.69 contains the data parallel code for the recurrence equation.

```
X = mul-scan(-Z) * (add-scan((Z * Y) /
    mul-scan(-Z)) + x0)
```

Figure 7.69: The data parallel code for Loop 5.

From this data parallel code the circuit in Figure 7.70 can be extracted. The ability to use non-standard operators such as *scans* has permitted a pipelined version of this kernel, greatly improving performance.

This circuit uses 7 functional units and two vector inputs and a single vector output. At 50 MHz, this circuit calculates 350 million operations per second. While the re-casting of the algorithm has added these extra functional units, thereby boosting the number of operations, the throughput of this circuit is still superior to other implementations, including those on supercomputers.

There are, however, some drawbacks to this approach. One is numerical stability. The parallel version of the algorithm requires that the product of all elements in the vector Z must be multiplied. This product must be representable by the hardware.

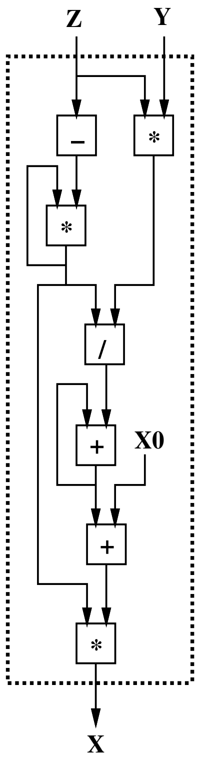


Figure 7.70: The configured circuit for loop 5.

Additionally, the introduction of the division operator produces the possibility of a divide by zero error. As long as Z is non-zero, this possibility is eliminated. Mapping all zero elements of Z to very small values near zero is one solution. Another is detecting the zero values and re-setting the entire circuit when they occur. Because a zero value in Z forces the output to zero, it can be viewed as a reset condition, similar to starting a new vector calculation.

Finally, the author knows of no automated method for translating recurrence equations to data parallel code. It is hoped that the performance gains made available by this technique will spur advances in this area.

Loop 11 - First Sum

Loop 11 is a first sum calculation. As in loop 5, a data dependency in the form of a recurrence is responsible for the low performance. Figure 7.71 give the original FORTRAN code for this procedure.

```
      X(1) = Y(1)
      Do  11 k = 2,n
11        X(k) = X(k-1) + Y(k)
```

Figure 7.71: The original FORTRAN code for Loop 11.

Unlike the recurrence equation in loop 5, the first sum in this kernel is very simple. It is essentially the definition of the *add-scan()* operator. Figure 7.72 gives the data parallel code for this kernel.

```
X = add-scan(Y);
```

Figure 7.72: The data parallel code for Loop 11.

The circuit extracted from this code, as shown in Figure 7.73 is trivial. A single *add-scan* operator is used. A single vector input Y is used to produce a single vector output X.

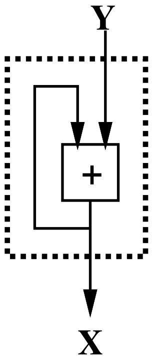


Figure 7.73: The configured circuit for Loop 11.

While providing a vector solution for this algorithm, the first sum suffers a similar performance limitation to loop 12, the first difference kernel. Since only a single functional unit is used, the number of operations at 50 MHz is only 50 million per second. Unlike the first difference, the dependency eliminates the possibility of replicating the circuit to perform vector calculations in parallel. Even this low rate of calculation, however, still exceeds supercomputer levels of performance.

7.6.5 Unstructured Loops

These kernels are the lowest in performance on the CRAY X-MP reference machine. As with the unvectorizable loops, they are unable to take advantage of the special vector hardware. Additionally, these kernels contain structures that further reduce performance, even for the non-vector portion of the processor.

For lack of a better term, these loops will be referred to as *unstructured.*

They are characterized primarily by the presence of unstructured control, usually in the form of *goto* statements, as well as complicated array indexing schemes.

Some of these loops actually exhibit a large amount of parallelism. It is often the way in which the algorithm is expressed, rather than any limitation in the underling algorithm, that reduces performance. For these reasons, some of these loops are better test of FORTRAN compiler optimizers than the underlying processor architecture.

Loop 24 - First Minimum

Loop 24 is selected as a representative of the unstructured loops because it has a deceptively simple implementation, while having the lowest performance of all 24 loops on a CRAY X-MP. The original FORTRAN code for this kernel is given in Figure 7.74.

```
        max24 = 1
        Do  24 k = 2,n
 24         if (X(k) .lt.  X(max24)) max24 = k
```

Figure 7.74: The original FORTRAN code for Loop 24.

An attempt to translate this algorithm into data parallel code reveals some of its limitations. First, this code uses a conditional operator, which interferes with vectorization. Next, it performs an operation involving only two scalar quantities. Finally, X is indexed by a scalar quantity which changes unpredictably. All of these factors combine to dramatically reduce the performance of this kernel.

The goal of this loop, however, is to find the location of the minimum value in the vector X. Constructing a data parallel solution will require more than a simple translation from the original FORTRAN specification of the algorithm.

```
Min = min-scan(X);
Min1 = delta(Min, 1);
Diff = Min1 < Min;

Index = add-scan(1);
M = max-scan(Index * Diff);
```

Figure 7.75: The data parallel code for Loop 24.

Figure 7.75 gives the data parallel code for this algorithm. The structures used require some explanation. First, since we are attempting to find the minimum value in the array, a *min-scan()* function is employed. This produces a vector *Min* containing the smallest value found up to that point.

This minimum value vector would be useful, except for the case of multiple identical minima. This algorithm requires the first occurrence of this minimum. The less-than ($<$) operator is used to compare neighboring elements in the *Min* vector. This locates the places where a new minimum has been found.

The *add-scan(1)* is used to produce a sequence of integers $1, 2, 3, \ldots$. Since there is no looping control structure in the data parallel code, this is necessary to produce the index value.

The index values created by this *add-scan()* operator are then multiplied by the results of the comparison. When a new minimum is found, this value is

"1", passing the index value. Otherwise the result of the comparison is zero, passing a zero as the result of the multiplication.

This produces a stream containing the index of the new minima indices and zeros. A *max-scan* operator will propagate the index of the first occurrence of the vector minima to the vector M. The last element in M contains the index of the vector minimum. Figure 7.76 give the dataflow graph extracted from this data parallel code.

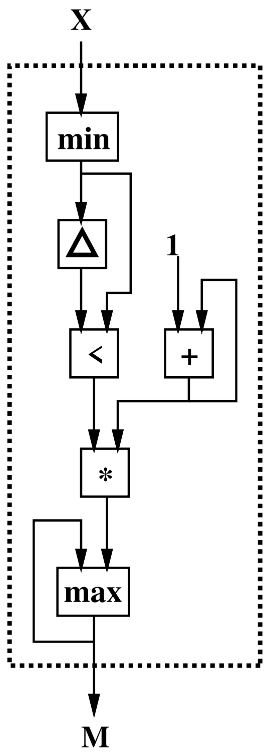


Figure 7.76: The configured circuit for Loop 24.

While a substantial modification of the original algorithm, this version is fully pipelinable and executes at approximately 200 million operations per second. While much of this figure is due to the additional functional units, this implementation still produces the desired result in approximately N clock cycles for a vector of length N.

A final note on unstructured algorithms. Many may not be suitable for reconfigurable machines. Unstructured access to vector data is a problem. Vector indexing of the form $X[Y[n]]$ is particularly difficult. Without special hardware support in the memory system, this type of calculation will almost certainly involve the host.

7.6.6 Performance

Table 7.3 gives an analysis of the performance of the seven kernels implemented. The estimates for the performance of the reconfigurable machine is based on an 50 MHz clock. This table is intended to illustrate the relatively high level of performance of the reconfigurable architecture. Too much emphasis should not, however, be placed on direct numerical performance comparisons.

Recall that most algorithms have been modified to take advantage of structures such as *scan* operators that are available to the reconfigurable system. Other algorithms, such as the recurrence equation in Loop 5, have been substantially transformed. These transformations make comparisons especially difficult. The goal here is merely to show that the reconfigurable architecture can perform the sorts of tasks usually performed by supercomputers at relatively high level of performance.

Perhaps not surprising, the algorithms that fared well on supercomputers also fared well on the reconfigurable machine. What is more surprising is the high levels of performance achieved by some of the loops which performed poorly on the CRAY X-MP, the supercomputer reference machine.

Loop Num.	Vector Inputs	Vector Outputs	Latency	Func. Units	Estimated MFLOPS	CRAY X-MP MFLOPS
1	2	1	5	5	250	160
3	2	1	2	2	100	138
12	1	1	2	1	50	63
22	3	1	5	5	250	68
5	2	1	6	7	450	14
11	1	1	1	1	50	14
24	1	1	5	5	200	3

Table 7.3: The LFK performance parameters.

While many of the algorithms are easily implementable and exhibit very high performance, some structures are still problematic. First, simple recurrences can be implemented efficiently using scan circuits. Currently, however, no simple algorithm exists for translating more complex recurrences into these circuits. Secondly, unstructured algorithms, particularly those which make use of indirect array indexing, are not well suited to reconfigurable logic. Using the host to vectorize these types of array accesses before they are submitted to the RPU may be a solution for some algorithms.

The implementation of these selected portions of the LFK demonstrates the feasibility of general purpose supercomputing using reconfigurable logic. While not a solution to all problems, the results for a large class of common computational structures is promising. Furthermore, it should be noted that the algorithms in the LFK are taken from real applications, written for traditional architectures. It is possible that new classes of algorithms which exploit the unique features of reconfigurable logic will provide even higher levels of performance for a larger class of problems.

Chapter 8

Summary

In this dissertation, an approach to high performance computing based on reconfigurable logic has been described. At the device level, a cellular reconfigurable logic architecture has been described. This device architecture uses a hexagonal array of three-input / three-output cells to support the configuration of pipelined arithmetic and logic circuits. By interfacing these devices to a dedicated memory system and a host processor, a high performance computing system may be constructed.

Perhaps more importantly, an existing software methodology has been used to program this system. By using a data parallel programming language, portions of the code containing large amounts of data parallelism are easily identified. The data flow graph of these portions of the code may then be mapped directly onto the reconfigurable hardware for execution.

Two additions to this software model have also been introduced. First, the use of parallel prefix or *scan* operators allows various accumulation operations to be performed efficiently. While *scan* operators have been shown to be powerful programming constructs on other high performance architectures, their implementation has suffered from inefficiency. In the approach demonstrated in this dissertation, *scan* functions are implemented with the performance and complexity on par with other standard arithmetic and logical operators.

The second software construct introduced as an addition to the data parallel programming model is *mixed valued striding*. This construct is a simple extension of traditional vector striding. Rather than allowing only integral strides, mixed values are permitted. This allows richer access patterns to data stored in arrays.

Since mixed valued striding is a direct extension of existing striding techniques, hardware support may be provided at a small incremental cost. With direct hardware support for this technique, longer data streams can be produced, reducing the overhead usually associated with calculations involving shorter vectors. It is believed that this technique is also be of value to other vector processing architectures, including digital signal processors.

Finally, several algorithms have been implemented and simulated. These include popular algorithms such as the Fourier transform and neural networks, as well as portions of the Livermore FORTRAN Kernels. These examples demonstrate that computationally intensive algorithms may be specified using existing software methodologies and executed at supercomputer levels of performance. Several of the Livermore FORTRAN kernels also indicate that computational structures which are problematic for other high performance architectures fare well on fine grained reconfigurable hardware. Table 8.1 summarizes some of the architectural parameters required by the algorithms simulated in this study. Similar results for the Livermore FORTRAN Kernels may be found in Table 7.3 in the previous chapter.

	Input Ports	Output Ports	Functional Units
Cellular Automata	3	1	20
String Matching	5	1	7
Mandelbrot Set	3	3	24
Neural Network	2	1	7
DFT	2	2	15
FFT	6	4	10

Table 8.1: System requirements of the algorithms.

As with other high performance architectures, memory bandwidth is an issue. One of the strengths of this architecture is the ability to perform multiple operations on input data before having to store a result to memory. In the case of the selected Livermore FORTRAN Kernels, a single vector output is generated from one to three input vectors. The bandwidth requirements are somewhat higher for the other algorithms in this study, but all may be implemented using standard design techniques.

In all cases studied, the performance of the reconfigurable hardware was at least comparable with modern supercomputers. In some cases, the performance was considerably higher. What makes these results particularly significant is that the hardware necessary to produce a reconfigurable system of the type described in this dissertation should be several orders of magnitude smaller and less costly than existing supercomputers.

While it is difficult to obtain a precise estimate of the cost of such a reconfigurable system, some rough estimates can be made. Currently, reconfigurable logic tends to be approximately 10 to 100 times less dense than custom logic.

An algorithm which can be implemented in a single custom logic device will require an RPU with 10 to 100 devices using similar fabrication technologies. It is expected that an RPU may be constructed on a single, standard printed circuit board without using any exotic packaging technology. The memory system would be incrementally more expensive than that of a typical workstation.

If it is assumed that the reconfigurable logic devices used to construct the RPU are high-volume commodity parts, the cost of a high performance reconfigurable system should be somewhat more than a workstation. It is also believed that this approach to computing will benefit more from advances in device technology than traditional processors.

8.1 Future Directions

This research has concentrated on the use of reconfigurable logic for general purpose computing. In particular, the use of high level languages has been addressed. Perhaps because this is such a relatively new field of study, many areas of future research present themselves.

First, the cellular architecture of reconfigurable logic arrays provides a platform for work in wafer scale technology and fault tolerance. When large cellular arrays are fabricated, faulty cells may be bypassed using various techniques. The possibility of using software techniques alone to produce functionally correct circuits in the presence of faulty cells is especially interesting. This should permits larger reconfigurable logic devices to be produced at a lower cost.

Another future direction of research involves the construction of the macrocells used by the system. Implementation of macrocells, particularly arithmetic circuits, presents some unique challenges. Unlike typical hardware design environments, the cellular array presents a constrained set of resources. Both logic and routing are fixed and finite quantities. This quantization, particularly the quantization of routing resources, provides a new framework for circuit design.

Another potential area of research is multiprocessor support. While a uniprocessor model of computation is presented here, the use of multiple reconfigurable logic coprocessors in conjunction with a multiprocessor system is possible. While some research in this area is currently being performed, many issues, especially software issues, are still unresolved.

From an architectural perspective, this approach to computation using reconfigurable logic makes use of some of the dataflow principles explored in the 1980s [24, 38, 4]. Unlike the dynamic dataflow architectures proposed for this model of computation, the approach taken here is more akin to static dataflow or data driven architectures [25, 71, 70]. The ability to efficiently support such a model of computation represents a step away from so-called von Neumann model of computation and its associated control structures [8].

Because of the use of dataflow principles, other less popular programming languages may turn out to be well suited to reconfigurable logic based machines. Dataflow and functional programming languages in particular seem to be promising.

The inherent flexibility of reconfigurable logic permits the hardware to be mapped to the algorithm, rather than the algorithm to the hardware. The resulting efficiencies have been demonstrated to produce high levels of performance from relatively small systems. As the circuit density of reconfigurable logic devices continues to increase, it will become easier to build larger systems. The ability to program these machines using high level languages should help to make these systems competitive with existing high performance computers.

Bibliography

[1] A. Abbott, P. M. Athanas, L. Chen, and R. L. Elliot, "Finding lines and building pyramids with spalsh 2," in *IEEE Workshop on FPGAs for Custom Computing Machines* (D. A. Buell and K. L. Pocek, eds.), (Los Alamitos, CA), pp. 155–163, IEEE Computer Society Press, April 1994.

[2] A. V. Aho, R. Sethi, and J. D. Ullman, *Compilers: Principles, Techniques and Tools*. Addison-Wesley Publishing Company, 1986.

[3] Algotronix, Ltd., *CAL1024 Datasheet*, 1990.

[4] Arvind, D. E. Culler, and G. K. Maa, "Assessing the benefits of fine-grain parallelism in dataflow programs," in *Supercomputing '88*, pp. 60–69, 1988.

[5] P. M. Athanas, "An adaptive machine architecture and compiler for dynamic processor reconfiguration," Tech. Rep. LEMS-101, Brown University, Division of Engineering, February 1992.

[6] P. M. Athanas, "A functional reconfigurable architecture and compiler," Tech. Rep. LEMS-100, Brown University, Division of Engineering, February 1992.

[7] P. M. Athanas and H. F. Silverman, "Processor reconfiguration through instruction-set metamorphosis," *IEEE Computer*, vol. 26, pp. 11–18,

March 1993.

[8] J. Backus, "Can programming be liberated from the von Neumann style? a functional style and its algebra of programs," *Communications of the ACM*, vol. 21, pp. 613–641, August 1978. 1977 ACM Turing Award Lecture.

[9] P. Bertin, D. Roncin, and J. Vuillemin, "Introduction to programmable active memories," Tech. Rep. 3, DEC Paris Research Laboratory, 1989.

[10] P. Bertin, D. Roncin, and J. Vuillemin, "Programmable active memories: A performance assessment," Tech. Rep. 24, DEC Paris Research Laboratory, 1993.

[11] G. Blelloch, "Scans as primitive parallel operations," in *Proceedings of the 1987 International Conference on Parallel Processing*, pp. 355–3672, 1987.

[12] G. E. Blelloch, *Vector Models for Data-Parallel Computing.* Cambridge, MA: The MIT Press, 1990.

[13] S. Borkar, R. Cohn, G. Cox, S. Gleason, T. Gross, H. T. Kung, M. Lam, B. Moore, C. Peterson, J. Pieper, L. Rankin, P. Tseng, J. Sutton, J. Urbanski, and J. Webb, "iWarp: An integrated solution to high-speed parallel computing," in *Supercomputing '88*, pp. 330–339, 1988.

[14] B. Box, "Field programmable gate array based reconfigurable preprocessor," in *IEEE Workshop on FPGAs for Custom Computing Machines*

(D. A. Buell and K. L. Pocek, eds.), (Los Alamitos, CA), pp. 40–48, IEEE Computer Society Press, April 1994.

[15] D. A. Buell and K. L. Pocek, eds., *IEEE Workshop on FPGAs for Custom Computing Machines*, (Los Alamitos, CA), IEEE Computer Society Press, April 1993.

[16] D. A. Buell and K. L. Pocek, eds., *IEEE Workshop on FPGAs for Custom Computing Machines*, (Los Alamitos, CA), IEEE Computer Society Press, April 1994.

[17] W. S. Carter, K. Duong, R. Freeman, H.-C. Hseih, J. Y. Ja, J. E. Mahoney, L. T. Ngo, and S. L. Sze, "A user programmable reconfigurable logic array," in *IEEE 1986 Custom Integrated Circuits Conference*, pp. 233–235, 1986.

[18] S. Casselman, "Virtual computing and the virtual computer," in *IEEE Workshop on FPGAs for Custom Computing Machines* (D. A. Buell and K. L. Pocek, eds.), (Los Alamitos, CA), pp. 43–48, IEEE Computer Society Press, April 1993.

[19] P. K. Chan, M. D. F. Schlag, and M. Martin, "BORG: A reconfigurable prototyping board using field-programmable gate arrays," in *First International ACM/SIGDA Workshop on Field Programmable Gate Arrays*, pp. 47–51, 1992.

[20] S. C. Chapra and R. P. Canale, *Numerical Methods for Engineers*. McGraw–Hill Book Company, second ed., 1988.

[21] C. E. Cox and W. E. Blanz, “GANGLION – a fast field-programmable gate array implementation of a connectionist classifier,” *IEEE Journal of Solid-State Circuits*, vol. 27, pp. 288–299, March 1992.

[22] S. A. Cuccaro and C. F. Reese, “The CM-2X: A hybrid CM-2 / xilinx prototype,” in *IEEE Workshop on FPGAs for Custom Computing Machines* (D. A. Buell and K. L. Pocek, eds.), (Los Alamitos, CA), pp. 121–130, IEEE Computer Society Press, April 1993.

[23] D. E. V. den Bout, “The anyboard: Programming and enhancements,” in *IEEE Workshop on FPGAs for Custom Computing Machines* (D. A. Buell and K. L. Pocek, eds.), (Los Alamitos, CA), pp. 68–77, IEEE Computer Society Press, April 1993.

[24] J. B. Dennis, “Data flow supercomputers,” *IEEE Computer*, vol. 13, pp. 48–56, November 1980.

[25] J. B. Dennis and G. G. Rong, “Maximum pipelining of array operations on static data flow machine,” in *International Conference on Parallel Processing*, pp. 331–334, August 1983.

[26] C. Ebeling, G. Borriello, S. A. Hauck, D. Song, and E. A. Walkup, “TRIPTYCH: a new FPGA architecture,” in *FPGAs* (W. Moore and W. Luk, eds.), pp. 75–90, Abingdon, England: Abingdon EE&CS Books, 1991.

[27] M. A. Ellis and B. Stroustrup, *The Annotated C++ Reference Manual.* The Addison-Wesley Publishing Company, 1990.

[28] G. Estrin, "Organization of computer systems – the fixed plus variable structure computer," in *Proceedings of the Western Joint Computer Conference*, pp. 33–40, May 1960.

[29] G. Estrin, B. Bussell, R. Turn, and J. Bibb, "Parallel processing in a restructurable computer system," *IEEE Transactions on Electronic Computers*, vol. EC-12, pp. 747–755, December 1963.

[30] G. Estrin and R. Turn, "Automatic assignment of computations in a variable structure computer system," *IEEE Transactions on Electronic Computers*, vol. EC-12, pp. 755–773, December 1963.

[31] G. Estrin and C. R. Viswanathan, "Organization of a "fixed-plus-variable" structure computer for eigenvalues and eigenvectors of real symmetric matricies," *Journal of the ACM*, vol. 9, pp. 41–60, January 1962.

[32] B. Fagin and C. Renard, "Field programmable gate arrays and floating point arithmetic," *IEEE Transactions on Very Large Scale Integrated (VLSI) Systems*, vol. 2, pp. 365–367, September 1994.

[33] J. T. Feo, "An analysis of the computational and parallel complexity of the livermore loops," *Parallel Computing*, vol. 7, pp. 163–185, June 1988.

[34] H. Fleisher and L. I. Maissel, "An introduction to array logic," *IBM Journal of Research and Development*, pp. 98–109, March 1975.

[35] M. Fouts, "The Livermore Loops in C." NASA Ames Research Center memo, 1994.

[36] F. Furtek, G. Stone, and I. Jones, "Labyrinth: A homogeneous computational medium," in *IEEE Custom Integrated Circuits Conference*, pp. 31.1.1–31.1.4, 1990.

[37] D. D. Gajski, "An algorithm for solving linear recurrence systems on parallel and pipelined machines," *IEEE Transactions on Computers*, vol. C-30, pp. 190–206, March 1981.

[38] D. D. Gajski, D. A. Padua, D. J. Kuck, and R. H. Kuhn, "A second opinion on data flow machines and languages," *IEEE Computer*, vol. 15, pp. 58–69, February 1982.

[39] G. R. Gao, "Algorithmic aspects of balancing techniques for pipelined data flow code generation," *Journal of Parallel and Distributed Computing*, vol. 6, pp. 39–61, 1989.

[40] M. Gokhale, W. Holmes, A. Kosper, D. Kunze, D. Lopresti, S. Lucas, R. Minnich, and P. Olsen, "SPLASH: A reconfigurable linear logic array," in *International Conference on Parallel Processing*, pp. I–526–I–532, 1990.

[41] M. Gokhale, W. Holmes, A. Kosper, S. Lucas, R. Minnich, and D. Sweely, "Building and using a highly parallel programmable logic array," *IEEE Computer*, pp. 81–89, January 1991.

[42] M. Gokhale and R. Minnich, "FGPA computing in data parallel C," in *IEEE Workshop on FPGAs for Custom Computing Machines* (D. A. Buell and K. L. Pocek, eds.), (Los Alamitos, CA), pp. 94–101, IEEE Computer Society Press, April 1993.

[43] R. C. Gonzalez and P. Wintz, *Digital Image Processing.* Reading, Massachusetts: Addison-Wesley Publishing Company, second ed., 1987.

[44] H. Grünbacher and R. W. Hartenstein, eds., *Field-Programmable Gate Arrays: Architecture and Tools for Rapid Prototyping.* Berlin, Germany: Springer-Verlag, September 1992. Selected papers from the Second International Workshop on Field Programmable Logic and Applications. Published in the *Lecture Notes in Computer Science* series, Volume 705.

[45] S. A. Guccione, "List of FPGA-based computing machines." World Wide Web page http://www.utexas.edu/˜ guccione/HW_list.html, 1994.

[46] S. A. Guccione and M. J. Gonzalez, "A data-parallel programming model for reconfigurable architectures," in *IEEE Workshop on FPGAs for Custom Computing Machines* (D. A. Buell and K. L. Pocek, eds.), (Los Alamitos, CA), pp. 79–87, IEEE Computer Society Press, April 1993.

[47] S. A. Guccione and M. J. Gonzalez, "A neural network implementation using reconfigurable architectures," in *More FPGAs* (W. Moore and W. Luk, eds.), pp. 443–451, Abingdon, England: Abingdon EE&CS Books, 1993.

[48] H. A. Gutowitz, ed., *Cellular Automata: Theory and Experiment.* Cambridge, Massachusetts: MIT Press, 1991.

[49] R. W. Hartenstein, A. G. Hirschbiel, M. Reidmüller, K. Schmidt, and M. Weber, "A novel ASIC design approach based on a new machine paradigm," *IEEE Journal of Solid-State Circuits*, vol. 26, pp. 975–989, July 1991.

[50] R. W. Hartenstein, R. Kress, and H. Reinig, "A reconfigurable data-driven ALU for xputers," in *IEEE Workshop on FPGAs for Custom Computing Machines* (D. A. Buell and K. L. Pocek, eds.), (Los Alamitos, CA), pp. 139–146, IEEE Computer Society Press, April 1994.

[51] R. W. Hartenstein and M. Z. Servít, eds., *Field-Programmable Logic: Architectures, Synthesis and Applications.* Berlin, Germany: Springer-Verlag, September 1994. Selected papers from the Fourth International Workshop on Field Programmable Logic and Applications. Published in the *Lecture Notes in Computer Science* series, Volume 849.

[52] P. J. Hatcher and M. J. Quinn, *Data–Parallel Programming on MIMD Computers.* Cambridge, MA: The MIT Press, 1991.

[53] J. L. Hennessy and D. A. Patterson, *Computer Architecture: A Quantitative Approach.* Morgan Kaufmann Publishers, Inc., 1990.

[54] W. D. Hillis, *The Connection Machine.* Cambridge, MA: The MIT Press, 1985.

[55] W. D. Hillis and J. Guy L. Steele, "Data parallel algorithms," *Communications of the ACM*, vol. 29, pp. 1170–1183, December 1986.

[56] D. T. Hoang, "Searching genetic databases on splash 2," in *IEEE Workshop on FPGAs for Custom Computing Machines* (D. A. Buell and K. L. Pocek, eds.), (Los Alamitos, CA), pp. 185–191, IEEE Computer Society Press, April 1993.

[57] D. R. Hush and B. G. Horne, "Progress in supervised neural networks: What's new since Lippmann," *IEEE Signal Processing Magazine*, pp. 8–39, January 1993.

[58] K. Hwang and F. A. Briggs, *Computer Architecture and Parallel Processing.* McGraw–Hill Book Company, 1984.

[59] K. Hwang and Z. Xu, "Multipipeline networking for compound vector processing," *IEEE Transactions on Computers*, vol. 37, pp. 33–47, January 1988.

[60] C. Iseli and E. Sanchez, "Spyder: A reconfigurable VLIW processor using FPGAs," in *IEEE Workshop on FPGAs for Custom Computing Machines* (D. A. Buell and K. L. Pocek, eds.), (Los Alamitos, CA), pp. 17–24, IEEE Computer Society Press, April 1993.

[61] W. H. Kautz, "Cellular logic-in-memory arrays," *IEEE Transactions on Computers*, vol. C-18, pp. 719–727, August 1970.

[62] W. H. Kautz, K. N. Levitt, , and A. Waksman, "Cellular interconnection arrays," *IEEE Transactions on Electronic Computers*, vol. C-17, pp. 443–451, May 1968.

[63] T. Kean and G. Feng, "Configurable logic: An approach to rapid implementation of ASIC's," Tech. Rep. CSR-234-87, University of Edinburgh, Department of Computer Science, June 1987.

[64] T. Kean and J. Gray, "Configurable hardware: Two case studies of micro-grain computation," in *Systolic Array Processors* (J. McCanny and E. S. Jr., eds.), pp. 310–319, Prentice Hall, 1989.

[65] T. A. Kean, *Configurable Logic: A Dynamically Programmable Cellular Architecture and its VLSI Implementation.* PhD thesis, University of Edinburgh, Department of Computer Science, January 1989.

[66] B. W. Kernighan and D. M. Ritchie, *The C Programming Language.* Prentice Hall Publishing Company, second edition ed., 1988.

[67] P. M. Kogge, "Parallel solution of recurrence problems," *IBM Journal of Research and Development*, vol. 18, pp. 138–148, March 1974.

[68] P. M. Kogge, *The Architecture of Pipelined Computers.* McGraw–Hill, 1981.

[69] P. M. Kogge and H. S. Stone, "A parallel algorithm for the efficient solution of a general class of recurrence equations," *IEEE Transactions on Computers*, vol. C-22, pp. 786–793, August 1973.

[70] I. Koren, B. Mendelson, I. Peled, and G. M. Silberman, "A data-driven VLSI array for arbitrary algorithms," *IEEE Computer*, pp. 30–43, October 1988.

[71] I. Koren and G. M. Silberman, "A direct mapping of algorithms onto VLSI processing arrays based on the data flow approach," in *Proceedings of the International Conference on Parallel Processing*, pp. 335–337, 1983.

[72] C. P. Kruskal, L. Rudolph, and M. Snir, "The power of parallel prefix," *IEEE Transactions on Computers*, vol. C-34, pp. 965–968, October 1985.

[73] H. T. Kung, "Why systolic architectures?," *IEEE Computer*, vol. 15, pp. 37–46, January 1982.

[74] S.-Y. Kung, "On computing with systolic/wavefront array processors," *Proceedings of the IEEE*, vol. 72, pp. 867–884, July 1984.

[75] R. E. Ladner and M. J. Fischer, "Parallel prefix computation," *Journal of the ACM*, vol. 27, pp. 831–838, October 1980.

[76] G. J. Lipovski and A. Tripathi, "A reconfigurable varistructure array processor," in *Proceedings of the 1977 International Conference on Parallel Processing*, pp. 165–174, IEEE Press, 1977.

[77] R. P. Lippmann, "Introduction to computing with neural nets," *IEEE Acoustics, Speech and Signal Processing*, pp. 4–22, April 1987.

[78] R. Lipton and D. Lopresti, "A systolic array for rapid string comparison," in *1985 Chapel Hill Conference on Very Large Scale Integration* (H. Fuchs, ed.), pp. 363–376, Computer Science Press, 1985.

[79] R. Lipton and D. Lopresti, "Comparing long strings on a short systolic array," in *Systolic Arrays* (W. Moore, A. McCabe, and R. Urquhart, eds.), pp. 181–190, Adam Hilger, 1986.

[80] J. C. Logue, N. F. Brickman, F. Howley, J. W. Jones, and W. W. Wu, "Hardware implementation of a small system in programmable logic arrays," *IBM Journal of Research and Development*, pp. 110–119, March 1975.

[81] D. P. Lopresti, "P-NAC: A systolic array for comparing nucleic acid sequences," *IEEE Computer*, pp. 98–99, July 1987.

[82] W. Luk, D. Ferguson, and I. Page, "Structured hardware compilation of parallel programs," in *More FPGAs* (W. Moore and W. Luk, eds.), pp. 213–224, Abingdon, England: Abingdon EE&CS Books, 1993.

[83] W. Luk, V. Lok, and I. Page, "Hardware acceleration of divide-and-conquer paradigms: A case study," in *IEEE Workshop on FPGAs for Custom Computing Machines* (D. A. Buell and K. L. Pocek, eds.), (Los Alamitos, CA), pp. 192–201, IEEE Computer Society Press, April 1993.

[84] W. Luk and I. Page, "Parameterising designs for FPGAs," in *FPGAs* (W. Moore and W. Luk, eds.), pp. 284–296, Abingdon, England: Abingdon EE&CS Books, 1991.

[85] W. Luk, T. Wu, and I. Page, "Hardware-software codesign of multidimensional programs," in *IEEE Workshop on FPGAs for Custom Computing Machines* (D. A. Buell and K. L. Pocek, eds.), (Los Alamitos, CA), pp. 82–90, IEEE Computer Society Press, April 1994.

[86] G.-K. Ma and F. J. Taylor, "Multiplier policies for digital signal processing," *IEEE ASSP Magazine*, vol. 7, pp. 6–20, January 1990.

[87] P. Marchal and E. Sanchez, "CAFCA: (compact accelerator for cellular automata) the metamorphosable machine," in *IEEE Workshop on FPGAs for Custom Computing Machines* (D. A. Buell and K. L. Pocek, eds.), (Los Alamitos, CA), pp. 66–71, IEEE Computer Society Press, April 1994.

[88] J. L. McClelland and D. E. Rumelhart, *Explorations in Parallel Distributed Processing: A Handbook of Models, Programs and Exercises.*

Cambridge, Massachusetts: MIT Press, 1988.

[89] W. McCulloch and W. Pitts, "A logical calculus of the idea immanent in nervous activity," *Bulletin of Mathematical Biophysics*, vol. 5, pp. 115–153, 1943.

[90] F. H. McMahon, "The Livermore Fortran kernels: A computer test of the numerical performance range," Tech. Rep. UCRL-53745, Lawrence Livermore National Laboratory, December 1986.

[91] G. Milne, P. Cockshott, G. McCaskill, and P. Barrie, "Realizing massively concurrent systems on the SPACE machine," in *IEEE Workshop on FPGAs for Custom Computing Machines* (D. A. Buell and K. L. Pocek, eds.), (Los Alamitos, CA), pp. 26–32, IEEE Computer Society Press, April 1993.

[92] R. C. Minnick, "Cutpoint cellular logic," *IEEE Transactions on Electronic Computers*, vol. EC-13, pp. 685–698, December 1964.

[93] R. C. Minnick, "A survey of microcellular research," *Journal of the ACM*, vol. 14, no. 2, pp. 203–241, 1967.

[94] S. Monaghan, T. O'Brien, and P. Noakes, "Use of FFGAs in computational physics," in *FPGAs* (W. Moore and W. Luk, eds.), pp. 363–372, Abingdon, England: Abingdon EE&CS Books, 1991.

[95] W. Moore and W. Luk, eds., *FPGAs*. Abingdon, England: Abingdon EE&CS Books, 1991. edited from the 1991 International Workshop on Field Programmable Logic and Applications.

[96] W. Moore and W. Luk, eds., *More FPGAs.* Abingdon, England: Abingdon EE&CS Books, 1993. edited from the 1993 International Workshop on Field Programmable Logic and Applications.

[97] Nordström, T. and Svensson, B., "Using and designing massively parallel computers for artificial neural networks," *Journal of Parallel and Distributed Processing*, vol. 14, pp. 260–285, March 1992.

[98] A. V. Oppenheim, A. S. Willsky, and I. T. Young, *Signals and Systems.* Prentice–Hall, 1983.

[99] H. L. Owen, U. R. Khan, and J. L. A. Hughes, "FPGA-based emulator architectures," in *More FPGAs* (W. Moore and W. Luk, eds.), pp. 398–409, Abingdon, England: Abingdon EE&CS Books, 1993.

[100] D. A. Padua and M. J. Wolf, "Advanced compiler optimizations for supercomputers," *Communications of the ACM*, vol. 29, pp. 1184–1201, December 1986.

[101] I. Page and W. Luk, "Compiling occam into FPGAs," in *FPGAs* (W. Moore and W. Luk, eds.), pp. 271–283, Abingdon, England: Abingdon EE&CS Books, 1991.

[102] S. S. Patil and T. A. Welch, "A programmable logic approach for VLSI," *IEEE Transactions on Computers*, vol. c-28, September 1979.

[103] W. Pfeiffer, A. Alagar, A. Kamrath, R. H. Leary, and J. Rogers, "Benchmarking and optimization of scientific codes on the CRAY X-MP, CRAY-2 and SCS-40 vector computers," *The Journal of Supercomputing*, vol. 4, pp. 131–152, June 1990.

[104] D. A. Pomerleau, G. L. Gusciora, D. S. Touretzky, and H. T. Kung, "Neural network simulation at warp speed: How we got 17 million connections per second," in *IEEE International Conference on Neural Networks*, pp. 143–150, 1988.

[105] W. Poundstone, *The Recursive Universe.* William Morrow and Company, 1985.

[106] W. T. Press, S. A. Teukolsky, W. T. Vetterling, and B. P. Flannery, *Numerical recipes in C.* Cambridge University Press, second ed., 1992.

[107] G. Quénot, I. Kraljić, J. Sérot, and B. Zavidovique, "A reconfigurable compute engine for real-time vision automata prototyping," in *IEEE Workshop on FPGAs for Custom Computing Machines* (D. A. Buell and K. L. Pocek, eds.), (Los Alamitos, CA), pp. 91–100, IEEE Computer Society Press, April 1994.

[108] F. Raimbault, D. Lavenier, S. Rubini, and B. Pottier, "Fine grain parallelism on a MIMD machine using FPGAs," in *IEEE Workshop on FPGAs for Custom Computing Machines* (D. A. Buell and K. L. Pocek, eds.), (Los Alamitos, CA), pp. 2–8, IEEE Computer Society Press, April 1993.

[109] D. E. Rumelhart, G. E. Hinton, and R. J. Williams, "Learning internal representations by error propagation," in *Parallel Distributed Processing: Explorations in The Microstructure of Cognition* (D. E. Rumelhart and J. L. McClelland, eds.), vol. 1, pp. 318–362, Cambridge, Massachusetts: MIT Press, 1986.

[110] G. W. Sabot, *The Paralation Model: Architecture Independent Parallel Processing*. Cambridge, Massachusetts: MIT Press, 1988.

[111] D. Sankoff and J. B. Kruskal, eds., *Time Warps, String Edits, and Macromolecules: The Theory and Practice of Sequence Comparison*. Reading, Massachusetts: Addison-Wesley Publishing Company, Inc., 1983.

[112] M. Shand, "Measuring system performance with reprogrammable hardware," Tech. Rep. 19, DEC Paris Research Laboratory, 1992.

[113] M. Shand, P. Bertin, and J. Vuillemin, "Hardware speedups in long integer multiplication," *Computer Architecture News*, vol. 19, pp. 106–113, March 1991.

[114] R. G. Shoup, *Programmable Cellular Logic Arrays*. PhD thesis, Carnegie-Mellon University, Computer Science Department, March 1970.

[115] R. G. Shoup, "Parameterized convolution filtering in an FPGA," in *More FPGAs* (W. Moore and W. Luk, eds.), pp. 274–280, Abingdon, England: Abingdon EE&CS Books, 1993.

[116] G. R. Sohie and W. Chen, *Implementation of Fast Fourier Transforms on Motorola's Digital Signal Processors*. Motorola, Inc., 1993.

[117] H. S. Stone, "Parallel processing with a perfect shuffle," *IEEE Transactions on Computers*, vol. C-20, pp. 153–161, February 1971.

[118] H. S. Stone, *High-Performance Computer Architecture*. Addison-Wesley Publishing Company, 1990.

[119] E. E. Swartzlander, Jr., ed., *Computer Arithmetic*, vol. 1. Los Alamitos, California: IEEE Computer Society Press, 1990.

[120] E. E. Swartzlander, Jr. and G. Hallnor, "High speed FFT processor implementation," in *VLSI Signal Processing*, pp. 27–34, IEEE Press, 1984.

[121] T. Toffoli and N. Margolus, eds., *Cellular Automata Machines.* Cambridge, Massachusetts: The MIT Press, 1987.

[122] K. W. Tse, C. H. Leung, and K. F. Cheng, "Implementation of preprocessing and feature extraction of chinese characters with FPGAs," in *More FPGAs* (W. Moore and W. Luk, eds.), pp. 307–314, Abingdon, England: Abingdon EE&CS Books, 1993.

[123] J. D. Ullman, *Computational Aspects of VLSI.* Rockville, Maryland: Computer Science Press, Inc., 1984.

[124] United States Department of Energy, Washington, D.C. 20585, *Human Genome Program Report*, June 1992.

[125] D. E. Van den Bout, J. H. Morris, D. Thomae, S. Labrozzi, S. Wingo, and D. Hallman, "AnyBoard: An FPGA-based reconfigurable system," *IEEE Design and Test of Computers*, vol. 9, pp. 21–30, September 1992.

[126] J. Varghese, M. Butts, and J. Batcheller, "An efficient logic emulation system," *IEEE Transactions on Very Large Scale Integrated (VLSI) Systems*, vol. 1, pp. 171–174, June 1993.

[127] J. Viitanen, T. Korpiharju, and H. Kiminkinen, "Mapping algorithms onto the TUT cellular array processor," in *International Conference on Application Specific Array Processors*, pp. 235–246, 1990.

[128] R. A. Wagner and M. J. Fischer, "The string-to-string correction problem," *Journal of the ACM*, vol. 21, no. 1, pp. 168–173, 1974.

[129] S. E. Wahlstrom, "Programmable logic arrays – cheaper by the millions," *Electronics*, vol. 40, pp. 90–95, December 11 1967.

[130] M. Wazlowski, L. Agarwal, T. Lee, A. Smith, E. Lam, P. Athanas, H. Silverman, and S. Ghosh, "PRISM-II compiler and architecture," in *IEEE Workshop on FPGAs for Custom Computing Machines* (D. A. Buell and K. L. Pocek, eds.), (Los Alamitos, CA), pp. 9–16, IEEE Computer Society Press, April 1993.

[131] W. J. Wilbur and D. J. Lipman, "Rapid similarity searches of nucleic acid and protein data banks," *Proceedings of the National Academy of Science (USA)*, vol. 80, pp. 726–730, February 198.

[132] A. Wolfe and J. P. Shen, "Flexible processors: A promising application-specific processor design approach," in *Proceedings of the 21st Annual Workshop on Microprogramming and Microarchitecture*, pp. 30–39, IEEE Press, 1988.

[133] Xilinx, Inc., *The Programmable Gate Array Data Book*, 1991.

Vita

Steven Anthony Guccione was born in New Orleans, Louisiana on November 30, 1962, son of Eugene Steven Guccione and Joan McPherson Guccione. He received his diploma from Archbishop Rummel High School in Metairie, Louisiana in 1980. He received the degree of Bachelor of Science in Electrical and Computer Engineering from Boston University in 1984 and the degree of Master of Science in Electrical Engineering from University of Minnesota in 1989. Mr. Guccione has been employed by Texas Instruments, Honeywell, Advanced Micro Devices, I.B.M. and several smaller companies.

Permanent address: 4205 Dauphine Drive
Austin, Texas 78727

This dissertation was typeset with LaTeX[‡] by the author.

[‡] LaTeX is a document preparation system developed by Leslie Lamport as a special version of Donald Knuth's TeX Program.